УДК 004.43

МЕТОДЫ РАСШИРЕНИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ (Часть 2)

***В. Д. Михеева*[1],**
старший инженер по программному обеспечению,
Российское отделение компании «Интел»

Приводится обзор методов расширения современных языков программирования, определенных автором и использованных для построения классификации расширений по способам интеграции и исполнения кода расширений. Рассматривается метод расширения языков программирования новыми конструкциями, методы исполнения расширений, а также приводится пример предметно-ориентированного расширения языка общего назначения средствами таблично-ориентированного программирования, реализованного автором на основе средств программирования системы эфемеридных расчетов в астрономии.

Ключевые слова *— предметно-ориентированный язык программирования, расширение языка программирования, инструментальные средства программирования, таблично-ориентированное программирование.*

Метод 4. Расширение новыми языковыми конструкциями

Все три рассмотренных метода [1] внесения расширенных возможностей в БЯ не изменяют имеющихся языковых конструкций, а лишь отличаются способом интеграции основной программы с внешней реализацией расширений. Рассматриваемый здесь четвертый способ представляет собой наиболее тесную интеграцию выразительных средств БЯ с конструкциями расширений. А именно, имеется в виду буквальное совмещение программного кода на двух языках в одной программе, что удобнее пользователю (программисту) по ряду причин. Во-первых, появляется возможность выражать специализированную функциональность с помощью наиболее подходящих языковых конструкций и, во-вторых, облегчается поиск ошибок в процессе разработки приложений, поскольку синтаксический и семантический контроль конструкций расширений наряду с контролем конструкций БЯ может быть выполнен компилятором на этапе трансляции.

Обычно при таком совмещении текст на специализированном языке выделяется в программном коде на БЯ с помощью окружающих маркирующих конструкций, легко выделяемых лексически, иногда даже на этапе *препроцессирования* текста программы (этапе предварительного анализа текста программы до синтаксического разбора). В других случаях появление специализированных конструкций однозначно определяется из контекста и такого выделения не требуется — тогда в одной программе буквально появляется «смесь» языковых конструкций с внешне ничем не выделенными границами.

Совмещение языков
с явным выделением кода расширений.

Типичным примером совмещения разных языков с использованием маркирующих конструкций для явного выделения кода расширения можно считать ассемблерные вставки в код C/C++. Рассмотрим пример такого кода — программу на C для компилятора GNU (GNU C compiler — GCC) с использованием кода встроенной функции на ассемблере (пример 7) [2]. Эта программа вычисляет наибольший общий делитель двух заданных чисел с помощью алгоритма Евклида, реализованного в виде встроенной функции на ассемблере в целях оптимизации производительности исполняемого кода.

Пример 7. Программа на C со встроенным кодом на ассемблере.

```
1 #include <stdio.h>
2
3 int gcd( int a, int b ) {
4   int result ;
```

[1] Научный руководитель — кандидат физ.-мат. наук, заведующий лабораторией астрономического программирования Института прикладной астрономии РАН *Ф. А. Новиков*. Окончание. Начало в № 4.

```
5   /* Compute Greatest Common Divisor using Euclid's Algorithm */
6   asm volatile ( "movl %1, %%eax;"
7           "movl %2, %%ebx;"
8           "CONTD: cmpl $0, %%ebx;"
9           "je DONE;"
10          "xorl %%edx, %%edx;"
11          "idivl %%ebx;"
12          "movl %%ebx, %%eax;"
13          "movl %%edx, %%ebx;"
14          "jmp CONTD;"
15          "DONE: movl %%eax, %0;" : "=g" (result) : "g" (a), "g" (b)
16  );
17
18  return result ;
19 }
20
21 int main() {
22  int first, second ;
23  printf( "Enter two integers : " ) ;
24  scanf( "%d%d", &first, &second );
25
26  printf( "GCD of %d & %d is %d\n", first, second, gcd(first,
second) ) ;
27
28  return 0 ;
29 }
```

В примере 7 представлена основная C-функция main, вызывающая вспомогательную C-функцию gcd, в теле которой присутствует сегмент встроенного ассемблерного кода (строки 6–16), реализующий тело алгоритма Евклида на уровне инструкций микропроцессора. Маркером начала и конца сегмента данного языкового расширения является синтаксическая конструкция asm (...), внутри которой располагаются строки кода на языке ассемблер для микропроцессора семейства Intel x86. Заметим, что спецификатор volatile здесь не является маркером, а только указывает компилятору, что данный код не подлежит автоматической оптимизации. Существует множество стилей написания подобных ассемблерных вставок. В данном примере применена конструкция в стиле AT&T, соответствующая стандарту [3] и поддерживающаяся компилятором GCC. Компиляторами C/C++ компании Microsoft для таких расширений используется другая нотация — в стиле Intel.

Можно привести другой пример, в настоящее время еще экзотический, поскольку речь идет об инновационной разработке компании Intel — об архитектуре Intel, поддерживающей интеграцию с различными акселераторами. Это архитектура IA с технологией расширения Exoskeleton Sequencer (EXO) и специально разработанные средства программирования с поддержкой языковых расширений — C for Heterogeneous Integration (CHI) [4]. «Интегрированная среда программирования CHI предоставляет разработчикам приложений возможность встраивать специализированный ассемблерный код для акселераторов или код на предметно-ориентированном языке в исходный код на традиционном языке C/C++»[2] [4, p. 185]. В примере 8 [4] представлен фрагмент кода для CHI, содержащий встроенный код на предметно-ориентированном языке DPL (Datastream Programming Language), специально разработанном для программирования реконфигурируемого акселератора SCC-DPE [4, p. 190].

Пример 8. Программа на C со встроенным кодом на языке DPL.

```
1 float Vin[4];
2 float Vout[4];
3
4 void *in_desc = (void *)chi_alloc_buffer_desc
5       (DPE_INPUT_BUFFER, Vin, 4, 1);
6 void *out_desc = (void *)chi_alloc_buffer_desc
7       (DPE_OUTPUT_BUFFER, Vout, 4, 1);
8
9 #pragma omp parallel target(dpe) shared(Vin,Vout)
10 descriptor(in_desc,out_desc)
11 {
12   __dpl {
13     configuration[1] cfgMult( vector val[1], vector coeff[1] )
14     {
15       result bs( mull(val, coeff), 13 );
16     }
17     flow[4] multiFlow( vector vec[4], vector coeffs[4])
18     {
19       vector ret[4]; result out;
20       selector[iter : 4] sel[1] = {{ iter }};
21       selector[iter : 4] selRev[1] = {{ 3 - iter }};
22       ret[sel] = cfgMult(vec[sel], coeffs[selRev]);
23     }
24     vector cf[4] = { 0.5 + I * 0.0 };
25     program dlMain()
26     {
27       Vout = multiFlow(Vin, cf);
28     }
29   }
30 }
```

В примере 8 фрагмент кода на языке DPL (строки 12–29) выделен в программе на языке C с помощью синтаксической конструкции __dpl { ... }, по структуре аналогичной конструкции встраивания ассемблерного кода в стиле Microsoft/Intel __asm { ... }. Хотя способ оформления языковых расширений сходен, разница примеров 7 и 8 состоит в том, что в последнем случае в основную программу встраивается фрагмент кода на другом языке высокого уровня, а не команды ассемблера.

Среди разнообразия современных средств программирования можно найти еще немало примеров совмещения различных языков программирования в одной программе с явным синтаксическим разделением кода на разных языках. Например, совмещение кода на языке разметки гипертекста HTML и описания функций на языке сценариев, таких как JavaScript, VBScript.

---

[2] *Пер. авт.*

Совмещение языков
без явного выделения кода расширений.

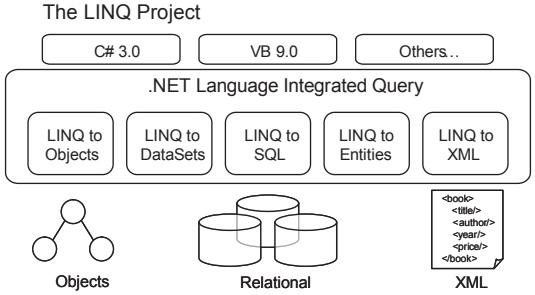
Теперь рассмотрим введение в язык расширений без маркировки на примере проекта LINQ компании Microsoft. Технология LINQ (.NET Language-Integrated Query) — это средство расширения языков на платформе .NET для интеграции с языком описания запросов [5, 6]. Эти расширения поддерживаются в языках C# 3.0 и VB 9.0 компании Microsoft, а также, возможно, появятся и в других языках на платформе .NET. В докладе одного из авторов технологии LINQ [7] рассказано, что LINQ — это абстракция, реализованная несколькими видами API, позволяющая в программе на платформе .NET единым образом оперировать с разного рода данными, выполняя традиционные для БД операции (такие как запрос, изменение и преобразование данных и т. п.). Такими данными могут быть любые .NET объекты в памяти программы, коллекции объектов, массивы данных, реляционные БД и документы XML (рисунок [7]).

Рассмотрим пример 9 [8], созданный на основе образца кода LINQ.

Пример 9. Программа на C# с конструкциями LINQ.

```
1 public void SimpleQuery()
2 {
3  Northwind db = new Northwind(...);
4  db.Log = Console.Out;
5  var query = from customer in db.Customers
6          where customer.City == "Paris"
7          select customer;
8
9  foreach (var Customer in query)  << Query Executes here
10  {
11   Console.WriteLine(Customer.CompanyName);
12  }
13 }
```

В примере 9 представлен код процедуры на языке C#, включающий конструкцию на языке описания запросов (строки 5–7, начиная с ключевого слова from), близком по синтаксису к SQL. Здесь среди конструкций C# употребляется конструкция совсем другого рода — описание запроса к БД, и заметим, без каких бы то ни было окружающих ее маркирующих символов. В этом случае конструкция расширения БЯ автоматически определяется компилятором исходя из анализа структуры выражения и по контексту (без синтаксических «подсказок» с помощью маркеров).



■ *Обзор технологии LINQ*

Примечательно, что в LINQ возможна также альтернативная запись запроса — через интерфейс API [7] (согласно первому методу интеграции расширений [1]). Поэтому строки 5–7 примера 9 (декларация переменной query, содержащей описание запроса) могут быть переписаны так, как показано в примере 10, и их исполнение приведет к получению такого же результата (т. е. обе формы записи эквивалентны по смыслу).

Пример 10. Программа на C# с конструкциями LINQ.

```
5  var query = db.Customers
6          .Where(customer => customer.City == "Paris")
7          .Select(customer => customer);
```

Методы Where и Select реализованы с помощью механизма C# введения так называемых *методов расширения* — возможности добавлять новые методы к уже описанным типам объектов. А логические выражения в скобках в вызовах методов представляют собой встроенные функции, называемые *лямбда-выражениями*.

В реализации LINQ форма описания запроса примера 9 преобразуется компилятором в конструкцию вида, описанного в примере 10, т. е. с использованием средств БЯ и применением интерфейса API (методов Where и Select), с помощью которого реализовано данное расширение.

Подводя итог, следует отметить, что для реализации описанного в этом разделе метода расширения БЯ новыми языковыми конструкциями в обоих рассмотренных вариантах необходима разработка специальных средств программирования, поддерживающих новый синтаксис и семантику расширенного языка. А это может быть достаточно трудоемкой задачей. Однако у данного метода есть неоспоримые преимущества по сравнению с представленными в первой части статьи — это максимальное удобство при программировании и возможность наиболее эффективного исполнения таких расширений.

Методы исполнения кода расширений

Помимо нескольких способов интеграции кода расширений в БЯ, можно также выделить несколько способов реализации описанных расширениями действий во время исполнения основной программы. По мнению автора, классификация расширений современных языков программирования в соответствии с основными методами исполнения их кода представляет собой список следующих категорий, перечисленных в порядке возрастания сложности реализации.

1. Программный код расширений (обычно библиотеки функций, иногда семейство классов) реализован на БЯ.

2. Расширения в виде внешних библиотек функций реализованы на другом языке программирования и, в некоторых случаях, встроены в систему исполнения.

3. Программная интерпретация расширенных возможностей во время исполнения основной программы с помощью специальной программы – интерпретатора.

4. Аппаратная интерпретация расширенных возможностей, например с помощью сопроцессора.

Первый метод исполнения расширений (для первой категории расширений) является наиболее простым с точки зрения его реализации. Он может применяться в сочетании с первым или вторым методами интеграции расширений [1]. Второй метод обычно применяется в сочетании также с первым методом интеграции расширений. Третий метод подходит для реализации расширений третьего [1] и четвертого методов интеграции расширений. Четвертый метод применяется в сочетании с первым или четвертым методами интеграции расширений.

Четвертый метод исполнения кода расширений является наиболее сложным с точки зрения его реализации, поскольку требует наличия и интеграции дополнительных специализированных аппаратных возможностей у основной вычислительной машины. Но благодаря такому способу достигается наибольшая производительность при исполнении кода расширений (хотя этот эффект может несколько ослабляться накладными расходами на передачу данных сопроцессора). Например, аппаратная интерпретация расширений применяется в системе программирования EXO-CHI для архитектуры IA с технологией интеграции с различными акселераторами (о которой говорилось при описании четвертого метода интеграции расширений). В частности, в качестве средства программирования микропроцессора архитектуры IA, интегрированного с акселератором DPE, разработана интеграция языка C со специализированным языком DPL [9]. Язык DPL является предметно-ориентированным, он предназначен для описания параллельных вычислений на акселераторе процессов обработки сигналов DPE [10].

Пример предметно-ориентированного расширения языка общего назначения средствами таблично-ориентированного программирования

Рассмотрим предметно-ориентированное расширение языка общего назначения средствами решения задач с данными в табличной форме, настраиваемое на предметную область[3]. Оно имеет рабочую настройку на предметную область эфемеридной астрономии и применяется для автоматизации вычислений в этой области [15].

Возникновение задачи разработки этого расширения связано со следующими историческими предпосылками, обусловливающими актуальность этой задачи. Специализированная система ЭРА, включающая средства программирования на предметно-ориентированном языке СЛОН (слежение и обработка наблюдений), более 20 лет успешно применяется в Институте прикладной астрономии (ИПА) РАН для автоматизации вычислительных задач эфемеридной астрономии. Ее создание и развитие продолжалось с середины 80-х гг. и нашло широкое применение в практической работе института. Ключевой идеей, положенной в основу системы ЭРА, является предложение Г. А. Красинского об использовании таблиц и алгебры таблиц как основных элементов программирования для решения задач эфемеридной астрономии в форме так называемых табличных операторов [12]. Используемая в системе ЭРА методика применения табличного подхода к обработке данных получила в дальнейшем название таблично-ориентированного программирования. Другой примечательной особенностью системы ЭРА является возможность настроить ее на выбранную предметную область [13]. И, наконец, высокое качество вычислительных моделей, положенных в основу функционального предметного наполнения, его полнота и регулярное обновление позволили системе ЭРА стать безусловным лидером в сфере программного обеспечения эфемеридной астрономии.

В процессе многолетнего опыта эксплуатации системы ЭРА назрела потребность в расширении средств, предоставляемых специализированным языком СЛОН [12, 14], средствами, доступными в традиционном языке программирования (а именно, Object Pascal [16]), что, согласно исследованиям [17, p. 15], вполне типично для предметно-ориентированных языков вообще. Эта потребность явилась побудительным мотивом к разработке нового языка Дельта на основе двух языков — СЛОН и Object Pascal, с объединением их функциональных возможностей путем расширения языка Object Pascal табличными операторами языка СЛОН. Выбор языка Object Pascal в качестве прототипа требующихся средств программирования общего назначения обусловлен несколькими причинами, в частности, тем, что Object Pascal тоже применяется для разработки вспомогательного функционального наполнения

[3] Это расширение реализовано автором под руководством канд. физ.-мат. наук *В. И. Скрипниченко* [11] на базе специализированной системы ЭРА (эфемеридных расчетов в астрономии) [12–14].

системы ЭРА, а отдельные конструкции языков Object Pascal и СЛОН схожи [18].

Рассмотрим выбор методов, примененных в реализации данного расширения. Простейший способ интеграции кода расширений на основе языка СЛОН с помощью библиотеки специализированных функций API в данном случае является неприемлемым — он неэффективен с точки зрения удобства программирования и продуктивности разработки с помощью таких расширений. Данный тезис проиллюстрирован в примере 2 [1] БД. Чтобы добиться желаемой эффективности целевого инструмента программирования при разработке языка Дельта, требуется более высокий уровень абстракции, чем уровень API, — а именно такой, как в исходном предметно-ориентированном языке СЛОН. Поэтому при проектировании языка Дельта выбран метод интеграции расширений в основной язык программирования (Object Pascal) в виде новых языковых конструкций — табличных операторов языка СЛОН в исходном виде, т. е. метод 4. При этом новые конструкции явно обозначаются в тексте на языке Object Pascal с помощью маркирующих конструкций, что позволяет выделять эти расширения на этапе предварительной трансляции (препроцессирования).

Теперь рассмотрим выбор метода исполнения данного расширения. В системе ЭРА [12–14] для исполнения программ на языке СЛОН, состоящих из табличных операторов, применяется программная интерпретация с помощью специальной программы, названной *процессором языка СЛОН* (СЛОН-процессором). Эта программа выполняет трансляцию табличных операторов в промежуточное представление, а затем его интерпретирует.

В отличие от способа, принятого для языка СЛОН, в целях исполнения программного кода на языке Дельта было решено сначала транслировать табличные операторы в промежуточный код на языке Object Pascal, а затем компилировать и компоновать всю программу с помощью имеющихся инструментов в системе Borland Delphi. В результате такого двойного преобразования получается исполняемый код. Таким образом, в системе Дельта применяется «двухпроходная» трансляция. Этот подход позволяет разделить способы исполнения отдельных частей табличного оператора — табличного выражения и использованных в нем блоков действий, а также предметно-ориентированных функций. Благодаря этому в реализации языка Дельта удалось ограничиться интерпретацией лишь части табличных операторов — табличных выражений — конструкций, не имеющих прямых аналогов в языке реализации Object Pascal (на этапе предварительной трансляции они преобразуются в строковые параметры функции интерпретации согласно методу 3 [1] интеграции расширений). Остальная часть табличных операторов программы на языке Дельта (блоки действий, включая вызовы предметно-ориентированных функций и процедур), наряду с основным текстом программы на Object Pascal, компилируется в исполняемый код с помощью двухпроходной трансляции. На первой фазе трансляции выполняется преобразование в промежуточный код на Object Pascal с вызовами функций API интерпретатора (согласно методу 1 [1] интеграции расширений), а на второй — преобразование в целевой исполняемый код.

Таким образом, в результате анализа разнообразных методов расширения языков программирования сделан обоснованный выбор следующих методов, подходящих для реализации языка Дельта [15].

- Для интеграции расширений в БЯ используются новые языковые конструкции (метод 4 интеграции расширений) на уровне разработки исходного кода приложений Дельта и вызовы функций API реализации расширений со специальными строковыми параметрами (методы 1 и 3 [1]) на уровне представления промежуточного кода на Object Pascal.
- Для исполнения кода расширений используется сочетание программной интерпретации с предварительной трансляцией в код на БЯ программирования (методы 1 и 3 исполнения расширений).

Среди методов интеграции расширений метод расширения языков программирования новыми конструкциями является наилучшим с точки зрения качеств получаемых средств программирования, но в то же время и наиболее сложным в реализации по сравнению с применением других методов. С другой стороны, в различных случаях оптимальным может оказаться выбор любого из первых трех рассмотренных наиболее простых и часто встречающихся методов интеграции расширений. Из-за относительной простоты реализации данные методы обладают некоторыми общими недостатками, связанными с необходимостью «подстраивать» расширенные возможности под существующую систему программирования на БЯ, что в результате приводит к значительным ограничениям в плане удобства программирования и автоматического контроля со стороны инструментальных средств программирования. Вместе с тем их неоспоримыми преимуществами являются возможность использовать имеющийся инструментарий и минимальность дополнительной разработки для поддержки соответствующих расширений. Благодаря этому данные методы являются востребованными, а в определенных ситуациях их применение может оказаться наиболее удачным решением.

Выбор метода исполнения кода расширений в общем случае во многом определяется условия-

ми конкретного проекта: составом имеющихся инструментальных средств программирования на БЯ; требованиями к производительности исполнения кода на расширенном языке; возможностями целевой аппаратной платформы по исполнению специализированных функций; наличием временных и человеческих ресурсов, необходимых для доработки базовых инструментальных средств программирования.

Заключение

В приведенном обзоре систематизированы сведения о разновидностях расширений в современных языках программирования, о методах интеграции и исполнения кода расширений. Этот материал дает представление о существующих расширенных возможностях в известных языках программирования, позволяющее сориентироваться в их многообразии и выбрать самое подходящее средство для решения конкретных специализированных задач наиболее удобным и эффективным способом с использованием расширений языка программирования. Эти сведения будут также полезны при выборе оптимальных методов для реализации собственных расширений в случае возникновения такой потребности.

Литература

1. **Михеева В. Д.** Методы расширения языков программирования. Ч. 1 // Информационно-управляющие системы. 2010. № 4. С. 46–52.
2. **Using Inline Assembly in C/C++:** revision 14.10.2006 // The Code Project (a community of Software development and Design developers). http://www.codeproject.com/KB/cpp/edujini_inline_asm.aspx (дата обращения: 10.09.2009).
3. **International standard:** ISO/IEC 9899:1990, Information technology —Programming Languages — C; ISO/IEC JTC1/SC22/WG14 — The international standardization working group for C. http://www.open-std.org/JTC1/SC22/WG14/ (дата обращения: 10.09.2009).
4. **Wang P. et al.** Accelerator Exoskeleton — Tera-scale Computing // Intel Technology Journal. Aug. 2007. Vol. 11. Is. 03. P. 185–196.
5. **Box D., Hejlsberg A.** LINQ: .NET Language-Integrated Query // MSDN Library. Feb. 2007. http://msdn.microsoft.com/ru-ru/library/bb308959(en-us).aspx (дата обращения: 14.07.2009).
6. **Введение в LINQ** // Библиотека MSDN. Нояб. 2007. http://msdn.microsoft.com/ru-ru/library/bb397897.aspx (дата обращения: 14.07.2009).
7. **Hejlsberg A.** DEV223: LINQ Overview // Microsoft Tech Ed Developers, 2006.
8. **Calvert Ch.** LINQ and Deferred Execution // MSDN Blogs. Charlie Calvert's Community Blog. http://blogs.msdn.com/charlie/archive/2007/12/09/deferred-execution.aspx (дата обращения: 10.08.2009).
9. **Mironov S., Pavlov V., Yakoushkin S., Ivanov V.** DPL: Domain specific programming language and tools // XI Intern. Symp. on Problems of Redundancy in Information and Control Systems, St.-Petersburg, July 2007. С. 251–255.
10. **Hoffman J., Ilitzky D., Chun A., Chapyzhenka A.** Overview of the Scalable Communications Core // IEEE Computer Society Annual Symp. on VLSI (ISVLSI '07), 9–11 May 2007, Porto Alegre, Brazil. P. 3–8. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4208886 (дата обращения: 21.03.2010).
11. **Михеева В. Д., Новиков Ф. А., Скрипниченко В. И.** Дельта-язык и система программирования для решения прикладных задач с табличными данными // Научно-технические ведомости СПбГПУ. 2007. № 4. Т. 2. С. 57–60.
12. **Krasinsky G. A., Novikov F. A., Skripnichenko V. I.** Problem Oriented Language for Ephemeris Astronomy and its Realization in System ERA // Cel. Mech. 1989. Vol. 45. P. 219–229.
13. **Новиков Ф. А.** Архитектура системы «ЭРА» — табличный подход к обработке данных // Препринт ИПА АН СССР / Л., 1990. № 16. — 32 с.
14. **Krasinsky G. A., Vasiljev M. V.** ERA-7. Knowledge Base and Programming System for Dynamical Astronomy: manual / IAA RAS. St.-Petersburg, 2001. — 232 p.
15. **Михеева В. Д.** Разработка предметно-ориентированных приложений с помощью инструментальных средств Дельта // Сообщения ИПА РАН / СПб., 2008. № 179. — 32 с.
16. **Object Pascal Language Guide.** 2001 // Borland Software Corporation. 100 Enterprise Way, Scotts Valley, CA 95066-3249. http://www.borland.com, http://docs.embarcadero.com/products/rad_studio/cbuilder6/EN/CB6_ObjPascalLangGuide_EN.pdf (дата обращения: 21.03.2010).
17. **Horowitz E.** Fundamentals of Programming Languages. Second Edition. — USA: Computer Science Press, 1984. — 446 p.
18. **Михеева В. Д., Скрипниченко В. И.** Расширение языка Object Pascal (Delphi) таблично-ориентированными средствами решения задач эфемеридной астрономии // Сообщения ИПА РАН/ СПб., 2006. № 168. — 20 с.