Обложка

**Аннотация**

Работа освещает вопросы реализации математической подсистемы ПА10, пакета моделирования динамики технических систем.

В исследовательской части дипломного проекта дан обзор существующих систем моделирования, их языков описания моделей. Рассмотрены достоинства и недостатки существующих вариантов, и предложено альтернативное решение, реализованное в ПА10. Также описаны основные возможности библиотеки DMAN, низкоуровневого ядра математического подсистемы пакета.

В конструкторской части описана разработанная архитектура подсистемы. Объяснены основные архитектурные решения. Отдельное внимание в этой части уделено анализу производительности разработанного ядра.

Технологическая часть посвящена вопросам применимости платформы Microsoft .NET для разработки научных и инженерных приложений.

[1 Введение 5](#_Toc106088094)

[2 Научно-исследовательская часть 7](#_Toc106088095)

[2.1 Моделирование динамики технических объектов 7](#_Toc106088096)

[2.2 Обобщенные формальные схемы 7](#_Toc106088097)

[2.3 Компонентный подход 9](#_Toc106088098)

[2.4 Аналогии 10](#_Toc106088099)

[2.5 Объектно-ориентированное проектирование 11](#_Toc106088100)

[2.6 Языки описания моделей 12](#_Toc106088101)

[2.6.1 ПА7 и Па9 13](#_Toc106088102)

[2.6.2 PSpice 16](#_Toc106088103)

[2.6.3 Verilog-A 17](#_Toc106088104)

[2.6.4 Анализ языков 18](#_Toc106088105)

[2.7 Вывод 20](#_Toc106088106)

[3 Конструкторская часть 21](#_Toc106088107)

[3.1 Библиотека DMAN 21](#_Toc106088108)

[3.1.1 Работа с библиотекой 22](#_Toc106088109)

[3.1.2 Интеграция 25](#_Toc106088110)

[3.2 Математическое ядро 27](#_Toc106088111)

[3.2.1 Классы-компоненты 28](#_Toc106088112)

[3.2.2 Классы, управляющие вычислением и выполняющие его 31](#_Toc106088113)

[3.2.3 Процесс моделирования 32](#_Toc106088114)

[3.3 Компоненты 33](#_Toc106088115)

[3.3.1 Базовые элементы ЭЕС 34](#_Toc106088116)

[3.3.2 Базовые классы компонентов 35](#_Toc106088117)

[3.3.3 Создание элементов 38](#_Toc106088118)

[3.3.4 Создание многополюсников 39](#_Toc106088119)

[3.3.5 Исключительные ситуации в компонентах 40](#_Toc106088120)

[3.4 Вопросы производительности 41](#_Toc106088121)

[3.5 Выводы 43](#_Toc106088122)

[4 Технологическая часть 44](#_Toc106088123)

[4.1 Применение C# в проектах для научных вычислений 44](#_Toc106088124)

[4.1.1 Производительность и языки 45](#_Toc106088125)

[4.1.2 MSIL и портируемость 45](#_Toc106088126)

[4.1.3 Автоматическое управление памятью 46](#_Toc106088127)

[4.1.4 Ориентация на объекты 47](#_Toc106088128)

[4.1.5 Высокоточные операции с плавающей точкой 48](#_Toc106088129)

[4.1.6 Многомерные массивы 49](#_Toc106088130)

[4.1.7 Небезопасный подход 50](#_Toc106088131)

[4.1.8 Эталонные тесты 52](#_Toc106088132)

[4.2 Выводы 53](#_Toc106088133)

[Приложение 1 Пример кода сборки моделируемой схемы из базовых компонентов 54](#_Toc106088134)

# Введение

Разрабатываемый программный комплекс ПА10 представляет собой систему моделирования динамических процессов в технических объектах, предназначенную для анализа процессов в различных физических системах во временной и частотной области.

ПА10 относится к группе систем инженерного анализа проектных решений на сосредоточенном уровне, наряду с такими системами как MSC.ADAMS, отечественные разработки серии ПА. Моделирование на сосредоточенном уровне связано с решением систем алгебро-дифференциальных уравнений и позволяет получить точные результаты за приемлемое время без использования больших вычислительных ресурсов. Поэтому этот вид моделирования находит все большее применение при проектировании изделий промышленности.

Основная задача, которая стояла при разработке новой системы из серии ПА, это повышение «дружественности» системы пользователю. Причем под термином «дружественность» мною понимается не только качество пользовательского интерфейса. Этим понятием я определяю все составляющие, способные повысить скорость разработки всего проекта, начиная от создания математической модели, заканчивая ее тестированием и подготовкой документации.

По сути, в ПА10 была предпринята попытка объединить все лучшее из более ранних систем этой серии: гибкость в разработке моделей системы ПА7, качественный оконный интерфейс ПА9.

Кроме заимствования лучших идей из систем более ранних поколений, в ПА10 мы попытались реализовать объектно-ориентированный (ОО) подход к проектированию математических моделей объектов. То, что уже давно стало стандартом де-факто в разработке ПО, по целому ряду причин с трудом приживается пока в области САПР, хотя между проектированием в любой САПР системе и программированием можно найти много общего.

Виной тому сложность реализации такого подхода при создании самой системы. Известно много случаев, когда разработчик, заявив такой функционал в новой версии, в результате отказывался от него, из-за высокой сложности реализации.

А между тем объектно-ориентированное проектирование предоставляет пользователю богатейшие возможности в вопросах повторного использования моделей, что немаловажно при промышленном использовании системы.

Особо хочу отметить основной предмет описания этого дипломного проекта – математическое ядро ПА10, основанное на библиотеке DMAN, и реализующее поддержку ОО модели проектирования, что позволило в значительной степени расширить потенциал системы. Использование же DMAN-а предоставляет пользователю целый ряд дополнительных возможностей, отсутствующих в других пакетах моделирования.

В качестве основной платформы при разработке ПА10 была использована Microsoft.NET, как одна из наиболее мощных современных технологий. В качестве языка разработки ПА10 выбран основной язык .NET - C#.

# Научно-исследовательская часть

## Моделирование динамики технических объектов

Системы инженерного анализа проектных решений – Computer Aided Engineering (CAE) Systems, предназначены для решения задач моделирования и анализа объектов производства. Основная задача подобных систем – создание адекватной модели проектируемого технического объекта (ТО) с целью снижения временных затрат и, соответственно, стоимости, разработки проектного решения изделия.

Основную группу CAE систем составляют программы, проводящие моделирование и анализ ТО на уровне дифференциальных уравнений в частных производных. Широко известны таки системы, как MSC.Nastran, ANSYS и так далее, которые выполняют моделирование с очень высокой степенью точности и достоверности, но требуют значительных затрат времени и вычислительных ресурсов для моделирования.

Однако, применение перечисленных систем далеко не всегда обоснованно, как по временным затратам, так и из-за сложности описания задачи. В подобных условиях все более активно начинают использоваться системы моделирования и анализа технических объектов на уровне обыкновенных дифференциальных уравнений, такие как MSC.ADAMS, обеспечивающие как быстроту, так и точность оценки принимаемых проектных решений.

Рынок подобных систем непрерывно рос на 8 – 9 % в год даже во время экономического кризиса (при общем уменьшении рынка CAD/CAE/CAM систем) и предполагается, что этот показатель возрастет до 14 % к 2007 году (по данным Daratech).

## Обобщенные формальные схемы

Основная масса систем моделирования динамики ТО – это узкоспециализированные в одной предметной области программные продукты. PSPICE специализируется в электронике, ADAMS и DADS в механике и гидравлике, МВТУ предназначен для моделирования систем управления, в первую очередь, ядерными реакторами и так далее.

Однако в настоящее время очень часто встают задачи комплексного моделирования сложных технических систем, состоящих из физически разнородных технических объектов. Для подобных задач требуются универсальные программы моделирования поведения ТО разной физической природы на основе метода аналогий переменных и уравнений. Именно к таким системам и относятся программы из серии ПА – ПА6, ПА7, ПА9 и разрабатываемая ПА10.

При математическом моделировании динамических процессов состояние любого технического объекта (ТО) в конкретный момент времени можно описать множеством некоторых физических величин, имеющих как постоянные значения, так и переменные значения в течение заданного времени моделирования. Обозначим множество переменных модели ТО вектором . Математической моделью динамических процессов в ТО в общем случае будет некоторая система ДАУ, замкнутая относительно этих переменных:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| , | | | | (‑) |
| Где |  | - | вектор переменных модели | |
|  |  | - | независимая переменная (например, время) | |

Множество математических систем и библиотек способны решать системы, аналогичные (2‑1).

Однако на практике все оказывается гораздо сложнее.

Во-первых, зачастую записать всю систему целиком, в силу ее сложности, оказывается непросто, а иногда и практически невозможно.

Во-вторых, подобный подход совершенно негибок. Любое изменение в ТО может привести к тому, что всю систему придется написать заново.

Эту проблема в комплексах серии ПА решается использованием схемных математических моделей технических объектов. Любой ТО можно представить как множество элементов, связанных между собой множеством связей, поэтому его можно отобразить схемной моделью в виде некоторой обобщенной формальной схемы (ФС), состоящей из набора элементов, связанных между собой в некоторых узлах этой схемы.

Подробнее с описанным выше подходом можно ознакомиться в работе **…**

## Компонентный подход

Практически любое техническое устройство обладает некоторой логичной для понимания человека структурой. Эту структуру могут создавать как отдельные подсистемы устройства (электрическая, механическая и так далее), так и непосредственно его узлы.

В независимости от принципа формирования структуры, наиболее логичным для проектировщика является такое же разделение математической модели на составляющие ее части. Это позволяет разработчику сосредоточиться на создании отдельных блоков (будем называть их компонентами) единого устройства, произвести их создание (задание уравнений, описывающих поведение блока) и отладку, и лишь потом реализовывать их совместную работу в рамках проектируемого ТО.

Все это объясняет популярность применения в системах моделирования компонентного подхода к моделированию. Практически все современные системы исповедуют такую стратегию проектирования. Более того, они идут дальше, предоставляя пользователю библиотеки моделей уже спроектированных узлов и устройств, позволяя инженеру использовать их при разработке своих ТО. Очень часто пользователю остается лишь собрать в визуальном редакторе свою собственную схему из готовых устройств, содержащихся в библиотеке.

Компонент обладает набором внешних переменных, которые используются для связи компонентов друг с другом в единую схему.

В разных системах нашли применение разные методологии использования компонентов. В первую очередь это отражается в различных вариантах их функционирования.

1. Ориентированные блоки. Такие компоненты разделяют свои переменные на входные, изменяющиеся вне компонента, и выходные, изменяющиеся внутри компонента, и являющиеся, по сути, результатом его работы. При сборке схемы устройства, выходы одних блоков подключаются ко входам других. Таким образом задаются причинно-следственные связи между компонентами. Примером системы, где нашел применение такой подход, является, например, Simulink Matlab.
2. Неориентированные блоки. Деления на входные и выходные переменные не производится. Значения переменных могут изменяться как самим блоком, так и вне его. Причинно-следственной связи в явном виде подобные схемы не задают. Примером такого подхода являются такие системы как PSPICE, серия ПА.

Второй метод является более естественным в вопросах приближения модели устройства к реальному его функционированию, однако его реализация более сложна для разработчика самой системы проектирования.

## Аналогии

Из главы 2.3.1 мы можем сформулировать следующие особенности компонента, описывающего некоторое техническое устройство (или его часть):

1. Компонент – это независимый элемент, взаимодействующий с другими компонентами через «входы-выходы» или «контакты».
2. Каждый компонент имеет как внешние переменные, влияющие на его состояние и являющиеся результатом его деятельности, так и внутренние, использующиеся для вычисления внешних.
3. Каждый компонент, помимо своих переменных несет некоторую вычислительную нагрузку, которая заключается в изменении его переменных, в соответствии с его математической моделью.

Эти пункты наводят нас на проведение аналогии между классами объектно-ориентированного языка программирования и компонентами в системах моделирования.

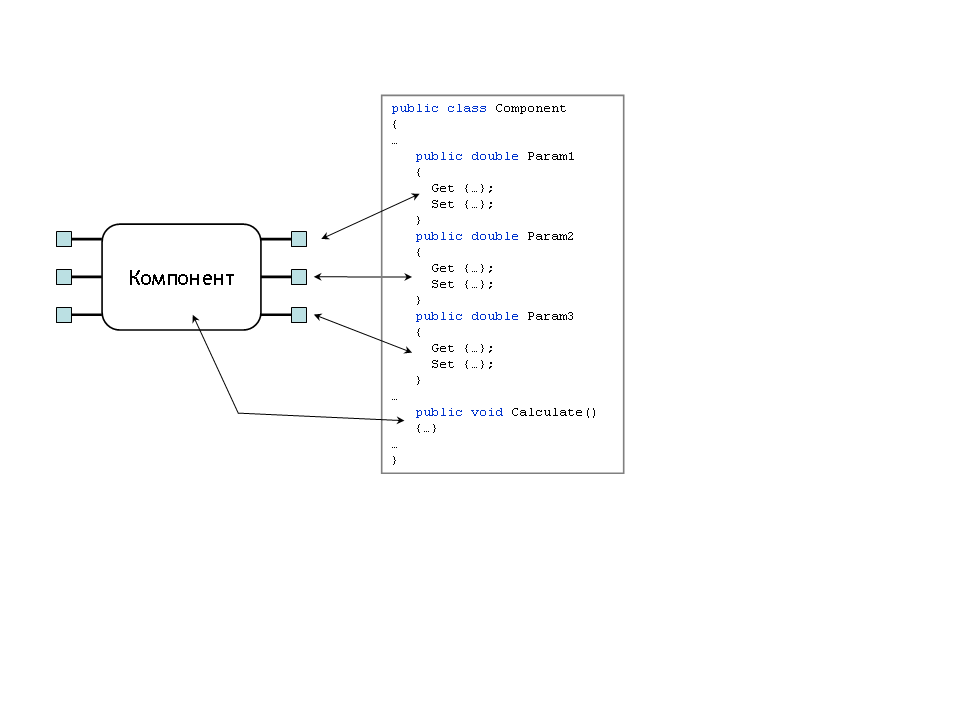


рис. ‑ Аналогии класса и компонента

Класс, также как и компонент несет некое поведение, реализованное в его методах, и взаимодействует с внешними для него классами посредством своих свойств (рис. 2‑1). Точно также мы имеем возможность перенести на классы оба типа компонентов, описанных выше.

Стоит, однако, заметить, что классы в языках программирования гораздо более гибкие понятия, чем определенные нами компоненты. И основное различие – принципы объектно-ориентированного программированиям (ООП), реализующиеся на классах.

Попробуем, основываясь на проведенных аналогиях, расширить понятие компонента до возможностей, предоставляемых нам классами при программировании.

## Объектно-ориентированное проектирование

Объектно-ориентированные подходы в программировании основываются на таких понятиях, как объект, инкапсуляция, наследование, полиморфизм, и подробно освещены в большом числе литературы, например, в **…**

Рассмотрим применимость этим подходов к моделированию технических объектов.

***Объект*** представляет из себя некую сущность, которая инкапсулирует (содержит) в себе данные и методы, взаимодействуя с внешней средой через некоторый интерфейс. Каждый объект представляет собой экземпляр класса. Мы уже отметили схожесть классов и компонентов, поэтому в нашем случае будем считать объектом каждый конкретный экземпляр компонента, добавленного в моделируемую схему.

При ***наследовании*** все элементы базового класса включаются в производный. При этом изменения, сделанные в структуре базового, автоматически отражаются и в производном классе. Целью наследования является модификация базового класса: дополнение его новыми элементами и/или переопределение существующих. Перенеся подобную возможность на процесс компонентного моделирования, мы получим возможность создания новых компонентов, на основе старых, но с частично расширенной или, наоборот, узкоспециализированной функциональностью. Например, на основе базовой модели транзистора возможно будет создать множество его конкретных модификаций с заданными параметрами. При этом изменения, вносимые в базовый компонент, будут отражаться и на дочерние.

Наследование также открывает нам дорогу к ***полиморфизму***. Полиморфизмом в ООП называется возможность замещения объекта одного класса объектом другого. При этом замещающий класс должен быть наследником замещаемого, либо реализовывать общий с ним интерфейс. В случае проектирования технических устройств такая функциональность дает нам потенциальную возможность легкой модификации схем и компонентов, разработку общих архитектур объектов на основе базовых компонентов, с возможностью последующего уточнения с помощью компонентов-наследников.

Все описанные выше возможности объектно-ориентированного проектирования способны дать в руки инженера дополнительные серьезные инструменты, которые, при грамотном использовании, способны значительно упростить как сам процесс, так и результат проектирования, а также понизить временные и финансовые затраты.

## Языки описания моделей

Реализация принципов объектно-ориентированного моделирования невозможна без поддержки этих возможностей непосредственно языком описания моделей, использующимся системой.

Рассмотрим некоторые системы моделирования и применяемые в них языки.

### ПА7 и Па9

Система Па9 является на текущий момент наиболее современной версией пакета ПА. От более ранних версий (в том числе и ПА7) ее выгодно отличает наличие визуального пользовательского интерфейса, в том числе и на этапе создания схемы устройства.

Пакет был разработан на языке Java, благодаря чему стало возможным его применение на различных операционных системах.

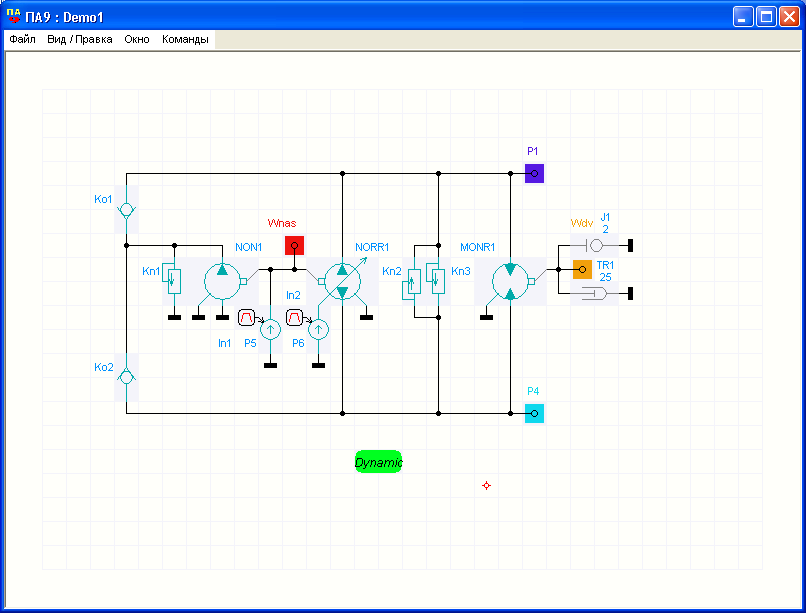


рис. ‑ РА-9

Язык описания моделей в Па9 основан на родном для системы языке Java. После разработки модели происходит ее конвертация в Java, с последующей компиляцией.

Отметим, что в ПА7 применялась схожая схема, с той лишь разницей, что в основе языка лежал другой язык программирования – С.

Рассмотрим некоторые возможности языка ПА9 на простейшем примере модели резистора.

$MODEL:Res;

#NODES: B,C;

#Param: R=50;

#PRC:Iter;

I$[0] = (F$2 - F$1)/P$R;

I$[1] = -I$[0];

Как мы видим, пользователь производит описание имени модели (раздел $Model), ее узлов (#Nodes), внешних параметров(#Param), а также непосредственно логику работы компонента (обработчики встроенных событий #PRC).

После конвертации в Java модель принимает вид:

public class MM$Res extends PA9$Element {

static String \_NName[] = {"B","C"};

static String \_PName[] = {"R"};

static double \_PDefault[] = {50};

public MM$Res() {

NAME = "Res";

NNe = 2;

NPl = 1; NPh = 1;

NName=\_NName; PName=\_PName; PDefault=\_PDefault; }

public void ELM\_Iter() {

$I[0] = ($F[2] - $F[1])/$P[0];

$I[1] = -$I[0];

}

Также, существует возможность сборки компонента на основе базовых.

Например, модель диода из **…**:

$MODEL: MyDiod;

#NODES: E,B,<Z>;

#PARAM:Is = 1e-9,A=38.7,Vs=1.5,Ra=0.1,Rb=10,Cd=1e-12;

#PRC:Iter;

double v = F$B-F$Z;

if(v <= P$Vs)

{

A$IST[0]= P$Is\*(exp(P$A\*v) - 1);

IST.ChanParam();

}

else

{

double k= P$Is\*P$A\*exp(P$A\*P$Vs);

double b = P$Is\*(exp(P$A \* P$Vs) \* (1 - P$A\* P$Vs) -1);

A$IST[0]= k\*v + b;

IST.ChanParam();

}

#TOP:

R1: R E Z = P$Ra;

IST: <I> Z B = 0;

COne: <C> Z B = P$Cd;

RTwo: R Z B = P$Rb;

Соответствующий код на Java:

public class MM$MyDiod extends PA9$Element {

static String \_NAMEo[] = {"R","R"};

static String \_IDo[] = {"R1","RTwo"};

static int \_MFo[][] = {{0,1,3},{0,3,2}};

double \_PARo[][] = { {0},

{0} };

static int \_IDp[] = {\_I\_,\_C\_};

static int \_MFp[][] = {{3,2},{3,2}};

static int \_MFAp[][] = {{},{}};

static String \_NName[] = {"E","B"};

static String \_PName[] = {"Is","A","Vs","Ra","Rb","Cd"};

static double \_PDefault[] = {1e-9,38.7,1.5,0.1,10,1e-12};

public MM$MyDiod() {

NAME = "MyDiod";

NNe = 2;

NPl = 6; NPh = 6;

NAMEo=\_NAMEo; IDo=\_IDo; MFo=\_MFo; PARo=\_PARo;

IDp=\_IDp; MFp=\_MFp; MFAp=\_MFAp;

NName=\_NName; PName=\_PName; PDefault=\_PDefault; }

public void ELM\_ChanParam() {

/\* R1:R \*/

PARo[0][0] = $P[3];

/\* RTwo:R \*/

PARo[1][0] = $P[4];

}

public void ELM\_Iter() {

double v = $F[2]-$F[3];

if(v <= $P[2])

{

$Vp[0][0]= $P[0]\*(exp($P[1]\*v) - 1);

IST.ChanParam();

}

else

{

double k= $P[0]\*$P[1]\*exp($P[1]\*$P[2]);

double b = $P[0]\*(exp($P[1] \* $P[2]) \* (1 - $P[1]\* $P[2]) -1);

$Vp[0][0]= k\*v + b;

IST.ChanParam();

}

/\* R1:R \*/

/\* IST:<I> \*/

$Vp[0][0] = 0;

/\* COne:<C> \*/

$Vp[1][0] = $P[5];

/\* RTwo:R \*/

}

}

### PSpice

PSpice - популярное во всем мире средство расчета и моделирования схем электрических устройств. Это один из самых распространенных пакетов, предназначенных для моделирования поведения сложных аналоговых устройств.

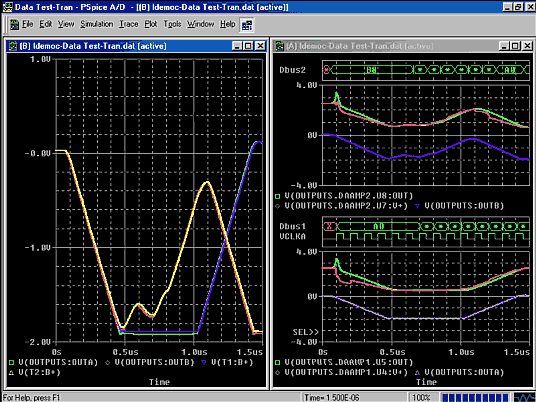


рис. ‑ Система PSpice

Длительный путь развития системы позволил накопить большое количество библиотек стандартных компонентов.

Язык PSpice достаточно сложен для изучения и во многом напоминает код ассемблера, но при этом и обладает большой мощностью.

Пример кода

### Verilog-A

Одним из наиболее популярных языков для описания устройств в системах проектирования электронных устройств является язык Verilog. Интегральные схемы слишком сложны для проектировщика, чтобы описывать их на уровне отдельных элементов – сопротивлений, конденсаторов, проводов и так далее. Поэтому, в основном, описание подобных устройств идет на более высоком уровне - целых цифровых устройств.

Однако достаточно часто встречаются задачи, где необходимо учесть аналоговое поведение системы. Для этих целей в рамках Verilog был создан язык смешанного сигнала для отражения поведения взаимодействия между цифровыми и аналоговыми частями схемы. Этот язык получил название Verilog-A.

Verilog-A – представляет собой язык высокого уровня для описания аналоговых компактных моделей. Конструкция языка позволяет проектировщику отражать потенциальное и потоковое описание электрических, механических и других систем.

Verilog-A относится к группе процедурных языков с простым синтаксисом, напоминающим язык C. Разработчик модели имеет возможность описать основные взаимоотношения между входами и выходами, а также параметры модели и их значения. Вопросы взаимодействия между созданной моделью и остальной программой моделирования решает Verilog-A компилятор.

Приведем простейший пример сопротивления на этом языке.

‘include "disciplines.vams"

module R(p,n);

electrical p,n;

parameter real R=50.0;

analog

V(p,n) <+ R \* I(p,n);

Endmodule

Строка module R(p,n); задает описание имени модуля R и наличие двух портов p и n. Далее указывается электрическая сущнсоть этихъ портов.

Модуль имеет один параметр вещественного типа R с значением по умолчанию равным 50. При использовании модуля в дальнейшем, проектировщик может менять значения описанных параметров на этапе инициализации.

Аналоговый блок (analog) модели содержит непосредственно описание ее поведения. В данном случае задается зависимость напряжения на выходах от протекающего тока.

После создания модели происходит ее компиляция, после чего ее можно использовать для моделирования.

### Анализ языков

Рассмотренные нами языки различны, как своими возможностями, так и нотацией записи. Но при этом можно заметить, что ни один из них в должной мере не реализует сформулированные нами признаки объектно-ориентированного проектирования ТО.

Некоторые из них умеют создавать новые модели, инкапсулируя ранее созданные, другие позволяют относить разработанную модель к некоторым группам (диоды, транзисторы и так далее). Но полностью ОО подход не реализует ни одна система.

Наиболее близка к этому – Па9, преобразующая модель на базовом языке в ее же описание на Java, однако, такая возможность разработчиками используется крайне редко, по причине сложности получающегося кода, что при внесении изменений может привести к большому количеству ошибок. Кроме того, сама возможность работы с Java кодом ТО лишь упоминается в документации, без каких-либо других пояснений.

Таким образом, мы приходим к выводу, что средств существующих языков недостаточно, для создания системы, полноценно поддерживающей ОО проектирование.

Каким же должен быть язык, соответствующий нашим требованиям?

Ответ очевиден, и дает его все та же ПА9: в качестве языка моделей можно, да и нужно, использовать непосредственно один из ОО языков программирования. При этом выбор следует делать среди наиболее современных языков, таких как Java или С#, обладающих простым, понятным с первого взгляда синтаксисом, и при этом мощными возможностями.

Отметим, что вводить какие-то дополнительные ключевые слова в язык, думается, совершенно излишне. Четко спроектированные классы для описания переменных, внешних контактов и других составляющих компонентов способны полностью заменить их.

Важной особенностью такого подхода является отсутствие потребности разработки каких-либо компиляторов или интерпретаторов, по той причине, что их и так существует огромное количество, в том числе и бесплатных. Дополнительно, использование обычных компиляторов дает возможность в дальнейшем применять разработанные модели не только в рамках системы, но и вне ее, если такая потребность существует, причем не только для задач моделирования. Например, компонент, моделирующий электродвигатель, может содержать в себе не только вычислительные методы, но и информацию, описывающую его трехмерную модель. Такой подход значительно расширяет рамки повторного использования разработанных моделей

В использовании языков программирования есть и такой важный плюс, как легкость изучения. Во-первых, с такими языками знакомо большое количество инженеров, и в таком случае потребность в изучении какого-то нового языка отпадает полностью. Даже если проектировщику язык не знаком, уже существует огромное количество специализированной литературы в этой области.

Конечно, мы не хотим заставлять пользователей отказываться от огромного числа ранее разработанных библиотек моделей. Всегда можно разработать конвертеры из одного языка в другой, либо создать специальный компонент, который будет уметь переадресовывать запросы к внешней модели.

## Вывод

Исследования современного состояния дел в области систем моделирования динамики технических объектов показывает, что подобные пакеты становятся все более востребованы в конструкторских бюро различных предприятий.

Несмотря на то, что системы эти бурно развиваются, существует целый ряд потенциальных нововведений, которые могут значительно повлиять на сам процесс проектирования. Одно из них - внедрение принципов объектно-ориентированного проектирования.

К сожалению, современные системы слабо подготовлены к таким изменениям, в силу исторической привязанности к их языкам описания моделей объектов, которые, в большинстве своем, совершенно не приспособлены к подобному переходу на новую методологию.

Весь спектр предложенных изменений можно реализовать лишь в принципиально новой системе, каковой вполне способна стать Па10.

# Конструкторская часть

## Библиотека DMAN

В качестве вычислительного ядра ПА10 используется библиотека DMAN, разработанная на кафедре РК6.

Программа DMAN предназначена для интегрирования систем ДАУ общего вида, не разрешенных относительно производных:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| , | | | | (‑) |
| Где |  | - | вектор дифференцируемых переменных системы размерностью *m* | |
|  |  | - | вектор производных дифференцируемых переменных по времени размерностью *m*, то есть | |
|  |  | - | алгебраических переменных системы размерностью *l* | |
|  |  | - | независимая переменная (например, время) | |
|  |  |  | вектор-функция системы размерностью *n*, где | |

В процессе интегрирования система (4‑1) на каждом шаге интегрирования автоматически дополняется системой:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| , | | | | (‑) |
| Где |  | - | шаг интегрирования | |
|  |  | - | вектор-функция размерностью *m*, которая соответствует выбранному методу интегрирования. Например, для неявного метода Эйлера *G* имеет вид:, где  *XN* - значение вектора дифференцируемых переменных в начале шага интегрирования  *X* - значение вектора дифференцируемых переменных в текущий момент времени | |

В программе DMAN реализованы 3 метода интегрирования систем ДАУ:

1. А-устойчивый неявный метод Эйлера первого порядка точности;
2. А-устойчивый неявный метод второго порядка точности;
3. А-устойчивый неявный метод четвертого порядка точности;

Во всех методах на каждом шаге интегрирования оценивается относительная, локальная погрешность интегрирования для каждой дифференцируемой переменной (по отношению к максимальному абсолютному значению каждой переменной на заданном отрезке интегрирования). Вычисленная погрешность сравнивается с заданной погрешностью EPS и, если она ее превышает, то шаг интегрирования *h* уменьшается. Максимальное абсолютное значение каждой дифференцируемой переменной определяется в ходе интегрирования автоматически, эти значения можно также задать перед началом интегрирования.

Во всех методах на каждом шаге интегрирования решается система нелинейных алгебраических уравнений относительно переменных *X*, *XP*, *Y* методом Ньютона, для которого необходимо вычислять матрицу Якоби для системы (4‑1) и системы (4‑2). Матрица Якоби для системы (4‑2) заполняется в программе DMAN автоматически. Для системы же (4‑1) она должна вычисляться в специальной подпрограмме пользователя FCT, наряду с расчетом вектор-функции F. Эта матрица состоит из двух подматриц:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| , | | | |  |
| где | *RJ* | - | матрица Якоби размером (*n*\**n1*); *n1=m+n* | |
|  | *RJ1* | - | матрица частных производных вектор-функции *F* по переменным *XP* размером (*n\*m*), т.е. матрица | |
|  | *RJ2* | - | матрица частных производных вектор-функции F по переменным *Z* размером (*n\*n*), т.е. матрица , где *Z* - вектор переменных системы ОДУ размерностью *n*, который включает дифференцируемые переменные и алгебраические переменные системы (4‑1) т.е. *Z = (X,Y)* | |

### Работа с библиотекой

Библиотека DMAN была разработана на языке Fortran, а в последствии была переведена на язык C.

Формат вызова DMAN:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DMAN(Z, XP, Z1, XP1, F, RJ1, RJ2, T, T0, TK, H, HMN, HMX, EPS, TKV, AR, N, M, NM, NCON, NBAD, IER, IP, FCT, OUT) | | | |  |
| где | Z | - | массив переменных системы ДАУ размерности N, в первых M элементах этого массива размещаются дифференцируемые переменные X системы ДАУ, в остальных элементах размещаются алгебраические переменные, поэтому всегда должно выполняться условие (N >= M). Перед началом интегрирования в первых М элементах этого массива размещаются начальные значения дифференцируемых переменных - X0. | |
|  | XP | - | массив производных дифференцируемых переменных по времени размерностью M. | |
|  | F | - | массив размерностью N для вычисления вектор-функции F системы ДАУ в подпрограмме FCT. | |
|  | RJ1 | - | матрица размером (N\*M) для вычисления частных производных вектор-функции F по переменным XP в подпрограмме FCT, т.е.  и т.д. | |
|  | RJ2 | - | матрица размером (N\*N) для вычисления частных производных вектор-функции F по переменным Z в подпрограмме FCT, т.е.  и т. д. | |
|  | T | - | текущее время интегрирования. | |
|  | T0 | - | заданное время начала интегрирования. | |
|  | TK | - | заданное время окончания интегрирования. | |
|  | H | - | текущий шаг интегрирования. | |
|  | HMN | - | заданный минимальный шаг интегрирования. | |
|  | HMX | - | заданный максимальный шаг интегрирования. | |
|  | EPS | - | заданная относительная погрешность интегрирования, одинаковая для всех переменных (число верных знаков в решении, обеспечиваемых для каждой дифференцируемой и алгебраической переменной). | |
|  | TKV | - | переменная, с помощью которой можно точно стробировать задаваемые пользователем моменты времени в ходе интегрирования. Эта переменная устанавливается в подпрограмме пользователя OUT и обычно используется для табуляции результатов. | |
|  | N | - | общее число уравнений в системе ДАУ. | |
|  | M | - | количество дифференцируемых переменных в системе ДАУ. | |
|  | NM |  | заданный номер метода интегрирования от 1 до 3. | |
|  | NCON | - | ключ расчета начальных значений переменных  NCON = 0 - выполняется расчет начальных значений переменных XP0 и Y0 при заданном Х0 для момента времени Т0 от нулевых значений XP0 и Y0. Значения Х0 соответствуют исходным значениям первых М элементов массива Z. До начала интегрирования с этим значением ключа NCON выполняется одно обращение к подпрограммам FCT и OUT для выполнения однократных вычислений параметров, которые в ходе интегрирования не изменяются. После выполнения однократных вычислений следует установить NCON=1 в подпрограмме FCT.  NCON = 1 - расчет начальных значений переменных XP0 и Y0 не производится. Этот признак следует использовать в случаях повторных обращений к программе DMAN, если надо продолжить интегрирование решаемой системы ДАУ. До этого должно быть хотя бы одно обращение к DMAN с NCON=0 или с NCON=2.  При нормальном выходе из программы DMAN устанавливается NCON=1.  NCON = 2 - выполняется расчет начальных значений переменных XP0 и Y0 при заданном Х0 для момента времени Т0 от вводимых начальных значений XP0 и Y0. Следует использовать для продолжения прерванного процесса интегрирования. | |
|  | NBAD | - | ключ уменьшения шага интегрирования, устанавливаемый в подпрограммах вычислений элементов вектор-функции ДАУ. Перед обращением к FCT устанавливается NBAD=0 на каждой итерации. Если отдельные переменные принимают значения, которые могут привести к останову вычислений (корень из отрицательного числа, превышение допустимого порядка в степенных функциях и т.п.), то необходимо установить NBAD=1, тогда итерации будут прерваны, и будет происходить уменьшение шага интегрирования до тех пор, пока можно будет продолжать вычисления, либо до значения минимального шага с сообщением о не сходимости итераций. Если в моделях установить NBAD=2, то произойдет уменьшение шага до учетверенного минимального с дальнейшим продолжением итераций с этим шагом (фактически расчет с новыми начальными условиями). NBAD=2 можно установить только в том случае, если NBAD не равно 1. NBAD=2 следует использовать для идентификации точек разрыва производных в кусочно-нелинейных функциях. | |
|  | IER | - | код ошибки. | |
|  | IP | - | Массив, содержащий информацию о ходе вычислений. Содержит такую информацию как номер итерации, суммарное число итераций и т.д. | |
|  | FCT | - | внешняя подпрограмма пользователя. В этой подпрограмме пользователь вычисляет вектор-функцию F решаемой системы ДАУ и матрицы RJ1 и RJ2 частных производных dF/dXP и dF/dZ.  К подпрограмме FCT происходит обращение на каждой итерации. При первом обращении к подпрограмме DMAN (ключ NCON=0 или NCON=2) организовано одно обращение к подпрограмме FCT при T=T0 и H=HMIN для выполнения однократных вычислений параметров, не изменяющихся в ходе интегрирования. После выполнения однократных вычислений следует установить ключ NCON=1. | |
|  | OUT |  | имя внешней подпрограммы пользователя для вывода и обработки результатов интегрирования. Обращение к этой подпрограмме происходит:   * после расчета начальных значений переменных XP0 и Y0 для момента времени Т0. При этом перед обращением к подпрограмме OUT устанавливается ключ NCON=0. Это можно использовать для однократных вычислений параметров подпрограммы OUT, которые не изменяются в ходе интегрирования. * после каждого успешно выполненного шага интегрирования; * в случае ошибки (IER не равно 0). | |

### Интеграция

Были рассмотрены два варианта интеграции библиотеки с пакетом:

1. Перевод библиотеки на язык C# с последующим непосредственным ее использованием при разработке математического ядра.
2. Компиляция библиотеки в ее С-варианте в native библиотеку с последующим «небезопасным» ее вызовом.

Оба варианта имеют свои плюсы и минусы. В первом случае основное достоинство – удобство последующего использования. Выбрав такой вариант, мы лишаемся надобности управлять используемой памятью (за нас это сделает сборщик мусора), а также значительно упрощаем себе процедуру вызова функции DMAN на расчет. К сожалению, язык С# (точнее будет сказать его компилятор) слабо приспособлен для выполнения сложных математических операций. Поэтому, применяя C# - вариант библиотеки мы значительно теряем в скорости вычислений.

Вариант с обычной native библиотекой значительно выигрывает в скорости. Однако недостатков тоже много.

Основные недостатки: трудности совместной отладки безопасного и native кода, и высокая сложности реализации маршаллинга[[1]](#footnote-1) данных. Кроме того, во втором случае нам приходится самим управлять памятью, используемой библиотекой. Проблема заключается даже не в ее своевременном выделении и очистке, за нас это сделает фреймворк. Главная сложность в том, что используемый для созданных в безопасном коде переменных сборщик мусора может в любой момент времени начать очистку памяти, и почти наверняка все массивы, указатели на которые были переданы в DMAN, переместятся. Как результат старые указатели окажутся неверны. Разработчики .NET такие случаи предусмотрели, позволив программисту блокировать переменные, указывая сборщику, что эти участки памяти передвигать нельзя ни в коем случае. Однако по каким то странным причинам, для многомерных массивов эта процедура была ими сильно осложнена, и такую задачу не удавалось решить длительное время.

Изначально был выбран первый вариант интеграции, в основном по причинам его простой реализации, однако, из-за ужасающе низкой производительности (смотрите ) впоследствии пришлось вернуться ко второй схеме.



рис. ‑Тестирование производительности различных версий библиотеки DMAN

## Математическое ядро

Программный комплекс ПА10 ориентирован, прежде всего, на задачи, связанные с решением систем дифференциально-алгебраических уравнений.

Описание таких систем может производиться многочисленными способами, например, с помощью эквивалентных электрических схем или просто в виде естественной символьной записи.

Возможность легкой адаптации математического ядра к различным методам описания моделируемого объекта – одна из важнейших задач, ставящихся при проектировании.

Следующая проблема – обеспечение объектно-ориентированности процесса разработки вычислительных компонентов. Это значит, что необходимо обеспечить легкое создание сущностей, описывающих математические модели физических процессов и объектов и связанных между собой отношениями “is a” и “part of”. Причем эта легкость должна быть обеспечена не только разработчикам программного комплекса, но и пользователям, желающим вести разработку не в визуальном режиме, а непосредственно на одном из языков программирования, поддержка которого реализована в среде .NET.

Кроме того, не стоит забывать, что и сама по себе возможность разработки визуального редактора компонентов и его возможности во многом зависит от архитектуры и возможностей математического ядра.

Также важным требованием является независимость моделей элементов от вычислительного модуля. Это значит, что его замена на более новую версию, либо на какую-то принципиально другую библиотеку, не должна приводить к переписыванию разработчиками и пользователями ранее разработанных вычислительных компонентов.

Как уже было сказано, основную вычислительную функциональность (интегрирование ДАУ) призвана была взять на себя библиотека DMAN. Таким образом, требовалось разработать обертку над этой библиотекой, отвечающую требованиям, предъявленным к математическому ядру.

Описанная ниже схема призвана решить перечисленные проблемы и требования.

### Классы-компоненты

Компонент – элементарная единица вычислительного процесса. Каждый компонент описывает часть вычислительной задачи и символизирует собой одно или несколько уравнений в общей системе ДАУ. Можно заметить, что в случае эквивалентных электрических схем компонентами являются не только какие-то элементы (сопротивления, источники и так далее), но и узлы.

Для того, чтобы какой-то класс стал компонентом, он должен реализовывать интерфейс IСomponent.

public interface IComponent

{

/// <summary>

/// Контроллер компонента

/// </summary>

IComponentController Controller

{

get;

}

}

Хочется сразу отметить, что, на самом деле, компонент далеко не всегда знает, в каких вычислениях ему приходится участвовать. В большом числе случаев он лишь несет описательную информацию, такую как свойства или, например, дифференциальное уравнение в текстовом виде. В нем, безусловно, могут производиться какие-то вычисления, но опять же, далеко не всегда результаты этих вычислений могут быть сразу же, без дополнительной обработки, применены при решении системы уравнений. Такая реализация обусловлена задачей избежать многократной реализации однотипных действий в разных компонентах, объединенных единой структурой построения. Примером могут служить элементы эквивалентных электрических схем. Кроме того, разделение описательной и вычислительной части представляет более гибкий архитектурный подход, и может облегчить, например, замену вычислительного ядра системы на принципиально другое. В таком случае описательная часть большинства компонентов останется нетронутой.

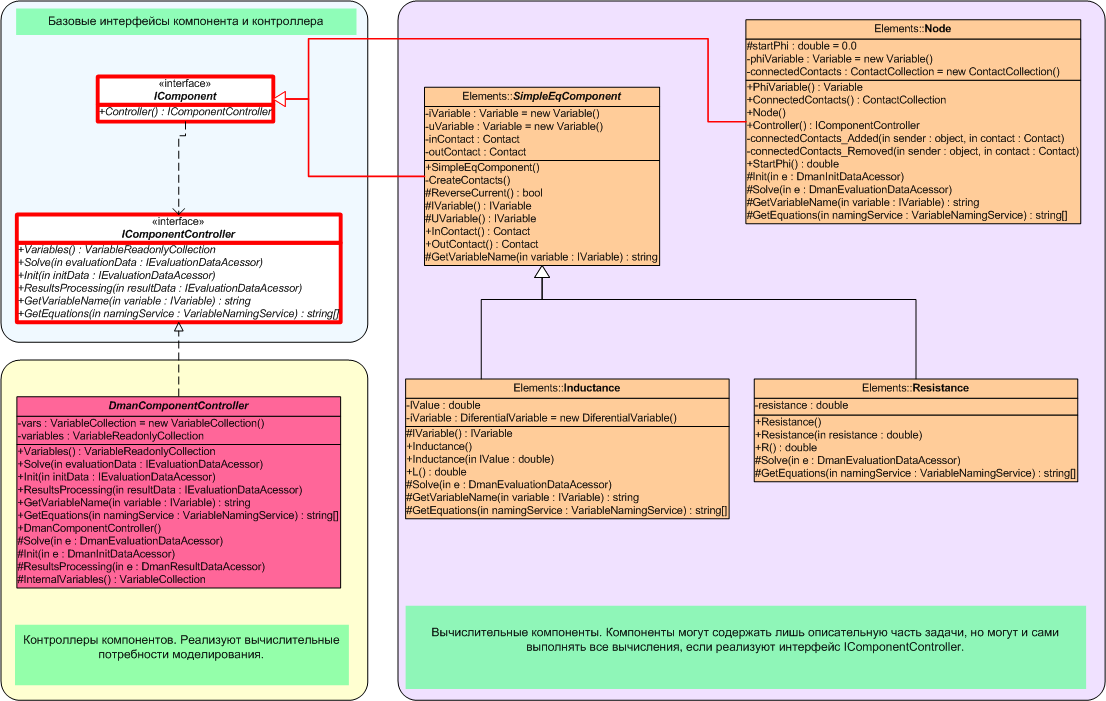


рис. ‑ Компоненты и контроллеры

Для перехода от данных и вычислений компонента к непосредственному заполнению матрицы Якоби и вектора значений функций (что собственно и требуется для решения системы ДАУ с помощью библиотеки DMAN) введен интерфейс IComponentController. Объекты, реализующие его, и производят необходимые действия для заполнения матриц и другие сопутствующие вычисления.

public interface IComponentController

{

/// <summary>

/// переменные компонента

/// </summary>

VariableReadonlyCollection Variables

{

get;

}

/// <summary>

/// Рассчет состояния компонента в текущий момент времени.

/// </summary>

/// <param name="evaluationData">доступ к различным данным для рассчета.</param>

void Solve(IEvaluationDataAcessor evalData);

/// <summary>

/// Инициализация компонента в начале рассчета.

/// </summary>

/// <param name="initData">Доступ к различным данным для инициализации.</param>

void Init(IEvaluationDataAcessor initData);

/// <summary>

/// Обработка результата.

/// </summary>

/// <param name="resultData">Доступ к различным данным для обработки результата.</param>

void ResultsProcessing(IEvaluationDataAcessor resultData);

...

}

Контроллер, исходя из его описания, предоставляет следующие свойства и методы:

VariableReadonlyCollection Variables - переменные, задействованные в уравнениях компонента. Переменные реализуют интерфейс IVariable и, в нашем случае, бывают двух типов – Variable (алгебраическая переменная) и DiferencialVariable (дифференциальная переменная).

Init(...) - функция инициализации вычислений в компоненте. Вызывается в самом начале вычислений, чтобы контроллер мог произвести вычисление параметров, значения которых не будет меняться на протяжении всего процесса. Соизмеряя с библиотекой DMAN – это аналог установки параметра NCON = 0.

Solve(...) – функция расчета состояния компонента в текущий момент времени. Именно в ней происходит вычисление значений в матрице Якоби системы и векторе значений функций.

ResultParams(...) – вызов этой функции происходит на каждом шаге интегрирования после вычислений, для вывода в какой-либо форме полученных результатов.

Кроме упомянутых свойств и функций контроллер также содержит методы для вывода информации об описываемых компонентом уравнениях в текстовом виде.

Следует отметить, что было бы абсурдом позволять контроллерам компонентов иметь полный и неограниченный доступ к матрицам и векторам, участвующим в вычислениях. Это не только повышает возможность непреднамеренных ошибок, но и допускает специальную “порчу” данных. Именно поэтому все обращения к матрицам и на запись и на чтение реализованы через специальный класс, реализующий интерфейс IEvaluationDataAcessor, приходящий в методы в качестве их параметра. Его задача – предоставить возможность доступа только к тем элементам матрицы, в которых компонент действительно нуждается и имеет какое-либо право на их изменение.

Например, в компоненте сопротивления, для заполнения данных для одного из описываемых им уравнений используется следующий код:

// I - U / R = 0

e.EqutationValues[this.IVariable] = e.Values[this.UVariable] – R \* e.Values[this.IVariable];

e.Jacobi[this.IVariable, this.UVariable] = 1.0;

e.Jacobi[this.IVariable, this.IVariable] = -R;

В данном случае, e – это объект класса, который реализует упомянутый выше интерфейс. Первая строка примера задает уравнение сопротивления. Две другие – производят заполнение матрицы Якоби, для этого уравнения.

Как видно из примера, изменение каких-либо значений возможно произвести только для тех переменных, информацией (по сути ссылкой на объект переменной) о которых обладает компонент.

### Классы, управляющие вычислением и выполняющие его

Теперь рассмотрим подробнее классы, которые управляют процессом моделирования.

Во-первых, следует сразу же назвать класс DmanSolver, собственно и являющийся основным классом вычислителя. Он позволяет задать все параметры вычисления: шаги по независимой переменной, погрешность, конечное время моделирования и так далее. Так же в нем пользователь задает моделируемую схему – набор компонентов, которые образует заданную систему уравнений. DmanSolver осуществляет инициализацию библиотеки DMAN перед моделированием, во время же вычислений реализует переадресацию всех данных библиотеке.

Особое внимание следует уделить сборщику схемы, реализованному в классе DmanLinker. Он выполняет две основные задачи:

1. Инициализация схемы. Как уже неоднократно замечалось, система ДУ описана набором компонентов-вычислителей. Непосредственно перед моделированием требуется произвести сборку системы – отделить вычислительные компоненты от управляющих, определить размерность задачи и проиндексировать переменные, создать ацессоры, которыми будут пользоваться компоненты для доступа к данным. Все эти функции возложены на сборщик в начальной стадии вычисления.
2. Вызовы компонентов по ходу моделирования. На каждом шаге интегрирования, библиотека DMAN производит вызов внешней функции FCT, в которой пользователь должен произвести все требующиеся вычисления. В нашем случае этот вызов переадресуется сборщику, чтобы он мог вызвать компоненты на расчет. То же самое происходит и при вызове DMAN-ом функции OUT для обработки результата.

### Процесс моделирования

Для внесения большей ясности опишем поэтапно процесс моделирования:

1. Сборщик получает список компонентов, для которых надо провести моделирование. Все компоненты опрашиваются, дабы определить число дифференцируемых и алгебраических переменных. Выделяется память под матрицы, необходимые для моделирования.
2. Создание классов доступа к данным для компонентов (ацессоров).
3. Инициализация библиотеки DMAN и начало непосредственно процесса моделирования.
4. На каждом шаге (по вызову функции FCT) библиотеки DMAN происходит сборка матрицы Якоби и заполнение вектор-функции системы. Как уже писалось выше, это заполнение производят контроллеры компонентов с помощью соответствующих ацессоров.
5. На каждом временном шаге (по вызову функции OUT) библиотеки DMAN производится обработка и выдача результатов вычисления.

## Компоненты

Задача разработки качественных, как в вычислительном, так и в архитектурном плане, компонентов представляется одной из наиболее важных.

Как уже неоднократно было отмечено, при разработке математического ядра ПА10 стояла задача максимально избежать ограничений, накладываемых на пользователя при создании математических моделей, и введением описанной выше архитектурой и применением в качестве языка моделей непосредственно языков программирования, поддерживаемых .NET это во многом удалось. Текущая реализация позволяет не привязываться к какому-либо одному методу задания моделируемой системы ДУ, будь то ЭЕС или просто текстовое описание системы на некоем языке. Принципиально возможен любой вариант. Следует лишь написать компоненты, которые бы осуществляли ваш метод описания задачи, и предоставляли бы требуемые данные ядру.

Такая гибкость во многом накладывает большую ответственность на разработчиков базовых компонентов, так как от их реализации во многом будет зависеть удобство разработки новых моделей.

Рассмотрим в этом разделе более подробно подходы к разработке набора компонентов для моделирования с помощью электрических эквивалентных схем, основного способа для описания задачи в системе ПА10.

### Базовые элементы ЭЕС

В качестве базовых будем рассматривать следующие элементы: источник потока, источник потенциала, сопротивление, конденсатор, индуктивность.

Все указанные элементы двухполюсники, которые описываются двумя типами переменных, составляющих исходный базис переменных ФС:

I – поток, протекающий через двухполюсник;

U – напряжение между полюсами двухполюсника.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Название** | **Условное изображение** | **Задаваемые переменные** | | **Уравнения** |
| **дифференциальные** | **алгебраические** |
| Источник потока |  | - | *I* |  |
| Источник потенциала |  | - | *U* |  |
| Сопротивление |  | - | *Ir , Ur* |  |
| Емкость |  | *Uc* | *Ic* |  |
| Индуктивность |  | *Il* | *Ul* |  |

Таблица ‑ Базовые элементы

Кроме того, базовыми элементами следует считать и узлы: обычный и базовый. Они так же, как и указанные выше элементы, задают свои переменные (потенциал в узле) и уравнения. Уравнения узлов описывают топологию схемы.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Название** | **Условное изображение** | **Задаваемые переменные** | | **Уравнения** |
| **дифференциальные** | **алгебраические** |
| Узел |  | - | *φ* |  |
| Базовый узел |  | - | *φ* |  |

Таблица ‑ Типы узлов

### Базовые классы компонентов

Для разработки ЕЭС был введен базовый абстрактный класс многополюсника EqComponent.

public abstract class EqComponent : DmanComponentController, IComponent

{

public EqComponent()

{

...

}

...

/// <summary>

/// Ножки компонента.

/// </summary>

public ContactsReadonlyCollection Contacts

{

get

{

return this.contacts;

}

}

#region IComponent Members

/// <summary>

/// Возвращает контроллер компонента.

/// </summary>

IComponentController IComponent.Controller

{

get

{

return this;

}

}

#endregion

}

Создание общего базового класса обусловлено задачей вынесения общих свойств, присущих всем компонентам такого типа. Кроме того, такой базовый класс упрощает другим модулям системы отбор компонентов такого типа, если это требуется.

EqComponent реализует не только интерфейс IComponent, но и IComponentController через наследование от DmanComponentController. Это объясняется тем, что в данном случае попытка вынесения единых вычислительных процедур для всех компонентов затруднительна, и значительно проще реализовать их для каждого элемента отдельно, непосредственно в теле их классов.

Абстрактный класс многополюсника содержит описание коллекции контактов – ножек компонента. Для подключения элемента к какому-либо узлу достаточно проставить этот узел в качестве значения свойства Node у нужного контакта.

Кроме многополюсника введен класс-наследник для двуполюсников, упрощающий их описание.

public abstract class SimpleEqComponent : EqComponent

{

public SimpleEqComponent(): base()

{

...

}

...

/// <summary>

/// Возвращает переменную выходного тока.

/// </summary>

protected virtual IVariable IVariable

{

get

{...}

}

/// <summary>

/// Возвращает переменную напряжения в компоненте.

/// </summary>

protected virtual IVariable UVariable

{

get

{...}

}

/// <summary>

/// Входящая нога источника тока.

/// </summary>

public Contact InContact

{

get

{

return this.inContact;

}

}

private Contact outContact;

/// <summary>

/// Выходящая нога источника тока

/// </summary>

public Contact OutContact

{

get

{

return this.outContact;

}

}

}

Как видно из кода класса, в базовое описание добавлены переменные для потока – IVariable и напряжения – UVariable. Как уже говорилось, пользователь может создавать переменные двух типов – алгебраические (Variable) и дифференциальные (DiferentialVariable). DiferentialVariable отличается от обычной тем, что автоматически создает переменную-производную по времени. Она также может быть применена в уравнениях.

Кроме того, в SimpleEqComponent уже описаны внешние контакты элемента InContact - вход, и OutContact - выход, хотя для большинства элементов разделение контактов на вход и выход является условным.

### Создание элементов

Для создания двухполюсника, на основе класса SimpleEqComponent следует сделать следующие шаги:

1. Пронаследоваться от базового класса двухполюсника SimpleEqComponent.
2. Добавить свойства, задающие внешние параметры элемента*.*
3. Описать переменные компонента IVariable и UVariable, перегрузив соответствующие свойства. В случае надобности можно добавить и дополнительные переменные, добавив объекты переменных в коллекцию Variables.
4. Написать основную функцию вычислений Solve, в которой для приходящего в качестве параметра времени вычислить значения уравнений компонента, а также элементы матрицы Якоби.

Ниже приведен пример создания конкретного элемента ЕЭС на примере емкости.

public class Condenser : SimpleEqComponent

{

public Condenser(): base()

{

}

private double cValue;

/// <summary>

/// емкость конденсатора

/// </summary>

public double C

{

get

{ return this.cValue; }

set

{ if(value > 0) this.cValue = value; }

}

private Variable iVariable = new Variable();

protected override IVariable IVariable

{

get { return this.iVariable; }

}

private DiferentialVariable uVariable = new DiferentialVariable();

protected override IVariable UVariable

{

get { return this.uVariable; }

}

protected override void Solve(DmanEvаluationDataAcessor e)

{

// C \* dU/dt - I = 0

e.EqutationValues[this.IVariable] = e.Values[this.IVariable] - this.C \* e.Values[this.uVariable.Diferential];

e.Jacobi[this.IVariable,this.uVariable.Diferential] = -this.C;

e.Jacobi[this.IVariable,this.IVariable] = 1.0D;

// U - (phi2- phi1)

e.EqutationValues[this.uVariable] = e.Values[this.uVariable]

+ e.Values[this.OutContact.Node.PhiVariable]

- e.Values[this.InContact.Node.PhiVariable];

e.Jacobi[this.uVariable,this.uVariable] = 1.0D;

e.Jacobi[this.uVariable, this.OutContact.Node.PhiVariable]= 1.0;

e.Jacobi[this.uVariable, this.InContact.Node.PhiVariable] = -1.0D;

}

}

Следует обратить внимание (см. коде выше) на то, как организован доступ к значениям переменных, функций и элементам матриц Якоби с помощью ацессоров.

### Создание многополюсников

Разработка многополюсников принципиально не отличается от разработки двухполюсников. Все пункты, действующие для двухполюсников, актуальны и тут.

Основное отличие: пользователю требуется самому создать нужное количество внешних контактов, в то время как в двухполюснике эта работа уже сделана за него. При создании контакта требуется указывать переменную потока, который протекает через данный контакт, а также направление потока в контакте.

...

Contact inContact = new Contact(this, this.IVariable, CurrentDirection.In);

this.InternalContacts.Add(inContact);

...

Направление потока используется компонентами узлов при составлении своих уравнений для простановки верного знака перед переменными.

### Исключительные ситуации в компонентах

Библиотека DMAN способна обрабатывать два типа исключительных ситуаций:

1. Недопустимое значение переменной (корень из отрицательного числа, превышение допустимого порядка в степенных функциях и т.п.)
2. Точка разрыва производной.

Указать библиотеке на возникновение таких ситуаций можно с помощью параметра NBAD (значения 1 и 2 соответственно).

Эта возможность реализована и в математическом ядре ПА10.

Рассмотрим, функцию f(x) с ограниченной областью определения х, например, функция  определена только для . Поэтому при каждом вычислении такой функции необходимо проверить значение аргумента х и, если аргумент выходит из области определения функции, сгенерировать исключение ProhibitiveAmountException, не производя дальнейших вычислений.

...

if(x < 0)

{

throw new ProhibitiveAmountException(“Значение переменной х меньше нуля”);

}

...

При получении такого исключения итерации будут прерваны, и произойдет уменьшение шага интегрирования до тех пор, пока можно будет продолжать вычисления, либо до значения минимального шага с сообщением об ошибке при решении нелинейных алгебраических уравнений

Некоторые базисные функции, теоретически определенные для любых значений аргументов, практически могут принимать значения, выходящие за пределы разрядной сетки ЭВМ, или накладываемыми реально возможными значениями аргумента и функции. Например, функция  обычно выходит за допустимые значения в реальных физических системах. В этом случае следует поступать также.

Важнейшей практической базисной функцией f(x) является кусочно-нелинейная функция, определяемая на множестве заданных точек. В точках излома эта функция имеет разрыв всех производных.

Для таких случаев в точках излома следует генерировать DiscontinuityException. Математическое ядро сообщит библиотеке DMAN (с помощью параметра NCON=2) о точке разрыва производной. После этого интегрирование продолжится за точкой с минимальным шагом интегрирования и с новыми начальными условиями.

Все остальные типы исключений (не указанные выше) трактуются математическим ядром как первый случай, и моделирование будет продолжаться с уменьшением шага интегрирования, пока не будет достигнуто минимальное значение.

## Вопросы производительности

Как уже сказано в главе 4.1.2, в целях ускорения процесса моделирования был сделан выбор в пользу native реализации базовой библиотеки DMAN.

Однако кроме DMAN-а вопросы производительности ставит и само применение объектно-ориентированной обертки.

Очевидно, что, повышая удобство библиотеки и ее потенциальные возможности, мы получаем потерю в скорости. Кроме непосредственно вычислений время затрачивается на сборку схемы перед вычислением, распределение вызовов к компонентам, а также обработку результатов, в его процессе. Кроме того, не следует забывать, что сами компоненты написаны на .NET языке, что также приводит к временным потерям по сравнению с аналогичной моделью на С или FORTRAN.

Проведенное тестирование быстродействия показало, что эти потери составляют 25 - 35% процентов, в зависимости от задачи, числа компонентов в модели, и используемого варианта DMAN.

Ниже приведен пример для одной из тестируемых задач (код тестовой задачи для разработанного математического ядра приведен в приложении 1).

|  |
| --- |
| **Задача:** |
|  |
| **Параметры вычислений:** |
| Метод интегрирования: A3 |
| **Результаты тестирования:** |
|  |

Аналогичные результаты прослеживаются и в других тестовых задачах.

Как видно из примера, при использовании разработанной оболочки для DMAN действительно наблюдается снижение производительности. Однако такие потери можно считать незначительными, и обоснованными возросшими возможностями и удобством использования при разработке современного программного продукта.

## Выводы

В описанном подходе к архитектуре математического ядра системы ПА10 были решены все основные проблемы, стоявшие перед нами, при разработке ПА10, а именно:

* Реализация подхода объектно - ориентированного проектирования, на основе компонентов;
* Максимальное упрощение процесса разработки новых математических моделей, с возможностью автоматической генерации компонентов.
* Независимость моделей от низкоуровневого ядра. Абстрагирование разработчика от схемы представления данных в нем;
* Обеспечение должного быстродействия.

Особо стоит отметить, что спроектированная и реализованная архитектура, несмотря на заявленную гибкость и универсальность, совершенно не обрезает возможностей базового ядра – библиотеки DMAN, что возможно далеко не всегда.

# Технологическая часть

В качестве платформы для разработки ПА10 была использована Microsoft.NET и ее основной язык разработки – C#.

Большое число разработчиков с осторожностью относится к перспективам использования современных платформ, таких как .NET и Java, в качестве инструмента для разработки САПР систем, как работающих с графической информацией, так и систем моделирования.

Основные доводы, приводящиеся в таком случае: низкая производительность и ориентированность таких платформ, в первую очередь, на быструю разработку решений для бизнеса. С последним замечанием трудно поспорить. Однако это не помешало Microsoft создать действительно удобную и производительную платформу для реализации совершенно различных, в том числе и научных, проектов.

Обещанная скептиками значительная потеря скорости оказывается на практике не такой уж существенной. Выигрыш же, от применения представленных инструментов и библиотек в совершенно различных областях, начиная от разработки интерфейса, заканчивая созданием распределенных приложений, огромен. Как результат, все больше компаний, специализирующихся на выпуске программных продуктов для инженерных разработок, используют платформу от Microsoft.

В качестве примера можно привести компанию UGS, последние разработки которой в области графических САПР, разработаны под платформу .NET.

## Применение C# в проектах для научных вычислений

Появление компьютеров позволило ученым легче доказывать теории, решать комплексные уравнения, моделировать трехмерные среды, прогнозировать погоду и выполнять многие другие задачи, требующие интенсивных вычислений. С годами были разработаны буквально сотни высокоуровневых языков, помогающих использовать компьютеры в данных областях. Но лишь немногие из них стали выдающимися языками научного программирования, в том числе C, C++ и FORTRAN, каждый из которых сыграл важную роль в мире научных вычислений.

Чтобы язык был пригоден как платформа разработки кода для научных расчетов, он должен помимо прочего предоставлять богатый набор инструментов для измерения производительности, а также простые в использовании и эффективные языковые средства. По сути, язык научного программирования должен создавать эффективный высокопроизводительный код, поддающийся тонкой настройке.

### Производительность и языки

Одним из главных отличий среди языков, применяемых в научном программировании, была производительность. Компиляторы и генераторы кода часто считаются факторами, ограничивающими производительность, но это утверждение не вполне корректно. Например, наиболее распространенные компиляторы C++ хорошо справляются с генерацией и оптимизацией кода.

Нельзя сказать, что единственный фактор, влияющий на производительность, — особенности языка. При сравнении быстродействия языков на самом деле сравнивается мастерство создателей компиляторов, а не сами языки. Если вы можете получить приемлемое быстродействие от языка, исполняющей среды или платформы, то выбор становится делом вкуса.

### MSIL и портируемость

Как и остальные .NET-ориентированные языки, C# компилируется в MSIL (Microsoft intermediate language), который выполняется в общеязыковой исполняющей среде (CLR). CLR можно упрощенно представить как комбинацию оптимизирующего JIT компилятора и сборщика мусора. C# предоставляет и использует большую часть функциональности CLR, поэтому важно детальнее рассмотреть, что происходит в исполняющей среде.

Одно из ключевых требований ученых — портируемость кода. Научно-исследовательские институты и лаборатории работают с большим количеством платформ и машин, включая рабочие станции Unix и персональные компьютеры.

CLR позволяет писать приложения и библиотеки на нескольких языках, компилируемых в MSIL. Затем MSIL может быть запущен в любой поддерживающей его архитектуре. Теперь ученые имеют возможность писать свои математические библиотеки на FORTRAN, задействовать их в C++ и использовать C# и ASP.NET для публикации результатов в Интернете.

CLR — в отличие от Java Virtual Machine (JVM) — это универсальная среда и целевая платформа для многих языков программирования. Более того, CLR обеспечивает взаимодействие на уровне данных, а не только на уровне приложений, и позволяет разделять ресурсы между языками.

Есть много языков, для которых существуют компиляторы, генерирующие MSIL-код. К ним, в частности, относятся Ada, C, C++, Caml, COBOL, Eiffel, FORTRAN, Java, LISP, Logo, Mixal, Pascal, Perl, PHP, Python, Scheme и Smalltalk. Кроме того, пространство имен System.Reflection.Emit предоставляет функциональность, значительно упрощающую создание новых компиляторов под CLR.

Перенос CLR на другие архитектуры уже ведется, но эта работа еще не закончена. Однако Mono/Ximian разработали реализацию с открытым исходным кодом для архитектур s390, SPARC и PowerPC, а также для систем StrongARM. Microsoft выпустила версию с открытым исходным кодом, работающую на системах FreeBSD, включая Mac OS X.

Эти разработки появились всего за несколько последних лет. Вероятно, через некоторое время полнофункциональная CLR станет доступна для всех распространенных архитектур.

### Автоматическое управление памятью

С точки зрения реализации, автоматическое управление памятью, наверное, — лучший подарок CLR разработчикам. Память выделяется относительно быстро (указатель кучи просто перемещается на следующий свободный слот) — по сравнению с более медленным и расточительным просмотром списка свободных страниц в вызовах malloc или new в C/C++. Более того, в период выполнения память управляется автоматически, освобождая и уплотняя незадействованное пространство. Программистам больше не надо гоняться за указателями, преждевременно освобождая блоки памяти или не освобождая их вовсе (хотя такие языки, как C# и Visual C++ по-прежнему дают такую возможность).

Реакция многих разработчиков на идею использования сборщика мусора вполне предсказуема. Хотя сборщики мусора действительно приводят к некоторым издержкам в интенсивно работающих с памятью приложениях, они все же берут на себя все запутанные детали отслеживания утечек памяти и очистки «брошенных» указателей. Сборщики мусора постоянно управляют кучей, сжимают ее и обеспечивают повторное использование.

Недавние исследования и эксперименты показали, что в приложениях с интенсивными вычислениями, где объекты выделяются и освобождаются чаще, сбор мусора способен реально увеличить производительность за счет сжатия кучи. Кроме того, часто используемые объекты, которые иначе были бы случайным образом распределены в памяти, собираются вместе, что повышает локальность и эффективность работы кэша. Это намного увеличивает общую производительность приложения. В то же время один из недостатков сборщиков мусора — их непредсказуемость, из-за которой трудно проводить сбор мусора именно тогда, когда это нужно. В этой области ведутся исследования, и сборщики мусора постоянно совершенствуются. Со временем появятся улучшенные алгоритмы с более предсказуемым поведением.

Вычислительный код обычно не инициирует сбор мусора. Это верно для сравнительно простых приложений, выполняющих в основном числовые операции без выделения большого количества блоков памяти. Многое зависит от предметной области и способа реализации решения. Если в нем задействовано множество объектов с коротким и средним сроком жизни, сбор мусора будет инициироваться довольно часто. Если у нескольких объектов большой срок жизни и они не освобождаются до завершения работы приложения, эти объекты получат статус постоянных и сбор мусора будет проводиться гораздо реже (или вообще не будет выполняться).

### Ориентация на объекты

C# — объектно-ориентированный язык. Поскольку реальный мир состоит из тесно взаимосвязанных объектов с динамическими свойствами, объектно-ориентированное программирование (ООП) зачастую лучше подходит для решения задач, связанных с научными расчетами. Более того, структурированный объектно-ориентированный код легче модифицировать, заменяя внутренние части в соответствии с изменениями в научных моделях.

Однако не все научные проблемы можно выразить через объекты и их отношения — в таких случаях ориентация на объекты приводит к лишнему усложнению. Кроме того, между объектами возможно сложное взаимодействие, которое в программной реализации приведет к еще большему усложнению или нежелательным издержкам. Моделирование взаимодействия между двумя атомами с использованием ООП может быть оправданным. Но представьте воображаемый куб, содержащий N3 атомов, где N — очень большое число. В итоге уравнения для вычисления всех парных сил и энергий могут оказаться настолько сложными, что для расчетов придется отказаться от ориентации на объекты и отдать предпочтение традиционному процедурному подходу. Но и процедурный код может оказаться менее производительным. Все зависит от способа хранения данных и применяемых алгоритмов.

Для многих задач научного программирования ООП может быть ценным инструментом, поскольку оно обеспечивает модульность и целостность данных, облегчающих распространение и повторное использование кода.

### Высокоточные операции с плавающей точкой

В научном коде нельзя игнорировать точность и корректность преобразований. Даже самые мощные современные компьютеры могут обеспечить точность лишь конечным числом разрядов. Значимость точности трудно переоценить — она подтверждается катастрофами, которые происходили из-за арифметических ошибок; вспомните, например, известный взрыв непилотируемой ракеты «Ариан 5» в 1996 году (64-битное число с плавающей точкой в системе инерциальной ориентации было неправильно преобразовано в 16-битное целое со знаком).

C# позволяет использовать в арифметических операциях с плавающей точкой более точные типы данных на аппаратных платформах с соответствующей поддержкой, например 80 битный формат Intel двойной точности. Этот тип имеет большую точность, чем double, и неявно задействуется нижележащим оборудованием при выполнении всех вычислений с плавающей точкой, давая абсолютно корректные или очень близкие к тому результаты. Следующая цитата взята прямо из раздела 4.1 спецификации C#: «…в выражениях вида x\*y/z, где умножение дает результат, выходящий за пределы диапазона значений double, но последующее деление возвращает промежуточный результат обратно в этот диапазон, тот факт, что выражение вычисляется в формате более широкого диапазона может дать конечный результат вместо бесконечности».

C# также поддерживает тип decimal — 128-битный тип данных, пригодный для финансовых и валютных расчетов.

### Многомерные массивы

C# поддерживает три типа массивов: одномерные (single-dimensional arrays), неровные (jagged arrays) и прямоугольные (rectangular arrays).

Как и в C, одномерные массивы индексируются от 0 и содержат элементы, хранящиеся в памяти последовательно. Такие массивы неявно обрабатываются специальным набором MSIL инструкций (newarr, ldelem, stelem и др.). Это делает одномерные массивы особенно привлекательными, поскольку компилятор может оптимизировать их самыми разнообразными способами.

Хотя одномерные массивы являются частью практически любого вычислительного приложения, численные расчеты немыслимы без эффективных многомерных массивов. Многомерные массивы в C# бывают двух видов: неровные и прямоугольные.

Неровный массив — это одномерный массив одномерных массивов. Неровный массив удобнее всего представить как вертикальный столбец, каждый слот которого указывает на другое место в памяти, где хранится какой-либо одномерный массив («строка»). Учитывая, что одномерные массивы в C# оптимизируются на уровне MSIL, массив одномерных массивов должен быть очень эффективен. Однако это почти полностью зависит от способа доступа к массиву. Если код доступа часто нарушает локальность, затраты на переходы по указателям в памяти могут стать весьма велики. Другое преимущество неровных массивов (или недостаток в зависимости от ваших критериев) заключается в том, что их «строки» могут быть разной длины, откуда и появился термин «неровный массив». Из-за этого при каждом обращении к другой «строке» проводится множество проверок границ. Однако в текущей версии исполняющей среды индексация неровных массивов оптимизируется лучше, чем при доступе к прямоугольным массивам.

Понимая недостатки неровных массивов, разработчики C# решили включить в спецификацию языка C-подобные многомерные массивы, предназначенные для критичных к быстродействию приложений и для пользователей, которые просто предпочитают прямоугольные массивы неровным. В отличие от неровных массивов прямоугольный массив хранится в смежных областях памяти, а не рассредоточен по куче. К сожалению, для прямоугольных массивов не предусмотрен отдельный набор MSIL инструкций, вместо них для доступа к элементам массивов используется пара вспомогательных методов set/get. Эти методы не создают издержек в период выполнения (по крайней мере для двух- и трехмерных массивов), поскольку JIT компилятор считает их внутренними. В сущности, код генерируется со множеством оптимизаций, в том числе без проверок выхода индексов за границы.

Если приходится выбирать элементы по диагонали или произвольным образом, неровные массивы резко ухудшают производительность из-за почти полного отсутствия локальности. С другой стороны, прямоугольные массивы в таких сценариях работают в четыре-пять раз лучше, чем неровные.

### Небезопасный подход

Удивительно, но C# поддерживает указатели в стиле C. Удивительно потому, что, как уже было сказано, управление памятью доверено в .NET сборщику мусора.

К счастью, исполняющая среда позволяет выйти из-под контроля сборщика мусора, фиксируя объекты в памяти, после чего они помечаются как неподлежащие перемещению при сборе мусора. Для тех, кто много работает с массивами это хорошая новость. Во-первых, к многомерным прямоугольным массивам можно легко обращаться так, будто они одномерные. Во-вторых, при доступе к элементу массива исполняющая среда выполняет проверку границ массива, чтобы убедиться в том, что вы не обращаетесь к памяти за пределами массива (однако есть несколько ситуаций, в которых JIT может проанализировать границы доступа и установить, что при каждом обращении проверка границ не нужна). Если вы уверены в своем коде, такая проверка лишь вносит нежелательные издержки.

Но есть один недостаток. Код, использующий указатели, должен быть помечен как небезопасный (unsafe) и может запускаться только в полностью доверяемой среде (fully trusted environment), а это, к примеру, означает, что вы не сможете запустить такой код прямо из Интернета. Однако это не должно волновать научных программистов, которые запускают код в полностью доверяемой среде, если только не создают сетевое или общедоступное приложение.

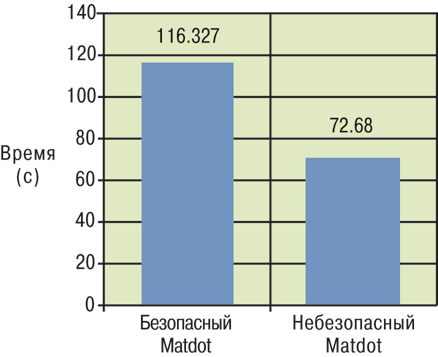


рис. ‑Сравнение безопасного и небезопасного кода

Чтобы сравнить небезопасный код с безопасным, на рис. 5‑1 показаны результаты эталонного тестирования двух методов перемножения матриц; один использует указатели для доступа к отдельным элементам массива, а второй — нет. Результаты показывают, что небезопасная версия примерно в два раза быстрее безопасной. Разница была бы еще больше, если бы небезопасная версия для перехода к следующему элементу массива использовала арифметику указателей, заменив умножение и сложение на приращение.

### Эталонные тесты

Эталонный набор математических тестов, входящих в пакет SciMark включает в себя следующие ядра: быстрые преобразования Фурье (Fast Fourier Transformations, FFT), итерации последовательных сверхрелаксаций (Successive Over-Relaxation iterations, SOR), квадратура Монте-Карло (Monte-Carlo quadrature), умножение разреженных матриц (sparse matrix multiplications) и разложение плотных матриц на множители (dense matrix factorization) для решения комплексных линейных систем.

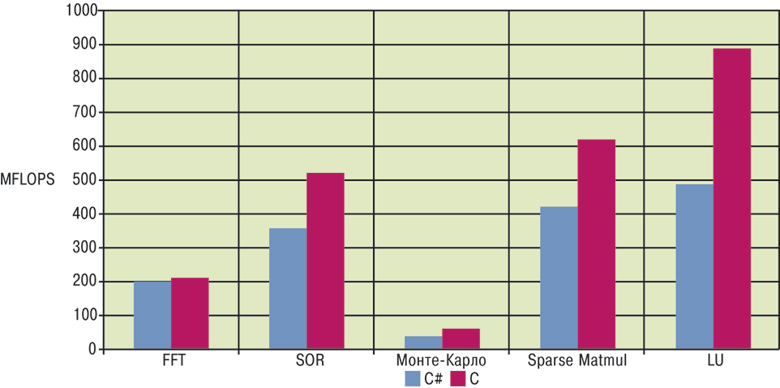


рис. ‑ Сравнение C# и C

Показанные выше результаты тестов были получены при выполнении комплекса эталонных тестов SciMark 2.0 на процессоре Pentium 4 с тактовой частотой 2.4 ГГц и 256 МБ оперативной памяти. Для сравнения были взяты результаты от C версии SciMark 2.0.

Следует отметить, что текущая C# версия SciMark использует двухмерные неровные массивы в большинстве своих ядер, таких как SOR, перемножение разреженных матриц (Sparse matmul) и LU. Очевидно, именно в этом причина того, что C# уступает C в этих трех ядрах и работает почти так же, как C в FFT. Возможно, переход на прямоугольные массивы дал бы значительно лучшие результаты.

Не стоит воспринимать эти результаты как окончательный вердикт по производительности двух языков. Поскольку тесты SciMark напрямую перенесены в C# из Java, эта реализация использует не самые оптимальные средства C#; поэтому результаты могут быть улучшены за счет таких альтернатив, как структуры, прямоугольные массивы и даже блоки небезопасного кода.

## Выводы

Вышеизложенное дает возможность заявить, что платформа .NET в целом и язык С# в частности, является вполне конкурентноспособной платформой для разработки научных и инженерных приложений.

Несмотря на более низкую производительность, широкие возможности платформы, вкупе с многоплатформенностью, уже сейчас позволяют нам это утверждать. Также не следует забывать, что .NET сейчас переживает этап бурного развития. На ближайший год уже заявлены к выпуску новые версии как самой платформы, так и C#, где будут улучшены уже существующие средства (в том числе и производительность), и добавлены новые, например шаблоны.

Все это позволяет с уверенностью заявлять, что выбор .NET для разработки ПА10 не был ошибочен, и во многом именно этот выбор позволит обеспечить в будущем развитие этой системы.

# Приложение 1 Пример кода сборки моделируемой схемы из базовых компонентов

//Создание элементов

BaseNode b1 = new BaseNode();

Node n1 = new Node();

Node n2 = new Node();

Node n3 = new Node();

Node n4 = new Node();

Node n5 = new Node();

Node n6 = new Node();

USource uSource = new USource(1);

Resistance r1 = new Resistance(1);

Resistance r2 = new Resistance(1);

Inductance i1 = new Inductance(1001);

Inductance i2 = new Inductance(999);

Condenser c1= new Condenser(0.001);

Condenser c2= new Condenser(1);

Condenser c3= new Condenser(0.001);

//Связывание схемы

b1.ConnectedContacts.Add(uSource.InContact);

b1.ConnectedContacts.Add(c2.InContact);

b1.ConnectedContacts.Add(r2.OutContact);

n1.ConnectedContacts.Add(uSource.OutContact);

n1.ConnectedContacts.Add(r1.InContact);

n2.ConnectedContacts.Add(r1.OutContact);

n2.ConnectedContacts.Add(i1.InContact);

n3.ConnectedContacts.Add(i1.OutContact);

n3.ConnectedContacts.Add(c1.InContact);

n4.ConnectedContacts.Add(c1.OutContact);

n4.ConnectedContacts.Add(c2.OutContact);

n4.ConnectedContacts.Add(c3.InContact);

n5.ConnectedContacts.Add(c3.OutContact);

n5.ConnectedContacts.Add(i2.InContact);

n6.ConnectedContacts.Add(i2.OutContact);

n6.ConnectedContacts.Add(r2.InContact);

in6 = new PotentialIndicator();

in6.Node = n6;

///Создание вычислителя

Pa10.Mathematics.DmanSolver solver = new Pa10.Mathematics.DmanSolver();

//Задание вычислителю компонентов схемы

solver.Components.Add(b1);

solver.Components.Add(n1);

solver.Components.Add(n2);

solver.Components.Add(n3);

solver.Components.Add(n4);

solver.Components.Add(n5);

solver.Components.Add(n6);

solver.Components.Add(uSource);

solver.Components.Add(r1);

solver.Components.Add(r2);

solver.Components.Add(c1);

solver.Components.Add(c2);

solver.Components.Add(c3);

solver.Components.Add(i1);

solver.Components.Add(i2);

solver.Components.Add(in6);

//Инициализация параметров вычисления

solver.DeltaTimeMin = 0.001;

solver.Epsilon =0.01;

solver.EndTime = 128;

solver.DeltaTimeMax = 128.0D;

solver.DeltaTimeMin = 128.0D/1e12;

solver.IntegrationMethod = IntegrationMethod.AStableImplicitQuadroPrecise;

//Запуск процесса вычисления

solver.Evaluate();

1. маршаллинг (англ. marshalling – расположение в определенном порядке) – механизм передачи данных через границы различных сред. В данном случае – процесс преобразования данных, из типов C# в переменные для применения в С-библиотеке DMAN и наоборот. [↑](#footnote-ref-1)