ТЕОРЕТИЧЕСКИЙ И ПРИКЛАДНОЙ НАУЧНО-ТЕХНИЧЕСКИЙ ЖУРНАЛ

Издается с сентября 2010 г.

СОДЕРЖАНИЕ

Журнал зарегистрирован в Федеральной службе по надзону в сфере связи, информационных технологий и массовых коммуникаци. Свидетельство о регистрации ПИ № ФС77-38590 от 24 декабря 2009 г.

Журнал распространяется по подписке, которую можно оформить в любом почтовом отделении (индексы: по каталогу агентства "Роспечать" — **22765**, по Объединенному каталогу "Пресса России" — **39795**) или непосредственно в редакции.
Тел.: (499) 269-53-97. Факс: (499) 269-55-10.
Http://novtex.ru E-mail: prin@novtex.ru

Журнал включен в систему Российского индекса научного цитирования.
Журнал входит в Перечень научных журналов, в которых по рекомендации ВАК РФ должны быть опубликованы научные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

УДК 004.415.2

Б. Г. Кухаренко, канд. физ.-мат. наук, ст. науч. сотр., вед. науч. сотр.,
Институт машиноведения РАН, г. Москва, e-mail: kukharenko@imash.ru

Принцип открытости-закрытости в программной инженерии и паттерны проектирования. Часть 3*

Принцип открытости-закрытости проявляется в объектно-ориентированном программировании на уровне микроархитектуры программных систем. Паттерны проектирования представляют иерархии классов, которые формируют общее решение задачи проектирования программных систем. Рассматриваются паттерны проектирования, которые при программировании на языке C++ с использованием шаблонов классов полезны при проектировании шаблонов арифметических выражений (expression templates).

Ключевые слова: *паттерны проектирования, шаблоны классов, шаблоны арифметических выражений, отображение типа, трейты, мета — программирование на основе шаблонов, библиотеки шаблонов*

Шаблоны функций и шаблоны классов языка C++ и библиотека STL

Шаблоны функций (*functions templats*) в C++ специализируются по крайней мере одним параметром типа [1]. Мотивация при создании шаблонов классов (*class templates*) или параметризованных типов (*parametrized types*) в C++ тесно связана с использованием контейнеров (*containers*). Классы контейнеров имеют интересное свойство: типы объектов, которые они содержат, не представляют интереса для проектировщика класса контейнера, но очень важны для пользователя конкретного контейнера. Таким образом, требуется, чтобы тип объекта, хранящегося в контейнере, был параметром класса контейнера. Это означает, что контейнер (например, вектор) должен хранить объекты любого типа. Это достигается использованием шаблонов классов. Стандартная библиотека шаблонов контейнеров — *The Standard Template Library* (STL) — это библиотека программирования на C++, разработанная А. Степановым и М. Ли во время работы в *Hewlett Packard laboratories* (*Palo Alto, California*) [2]. Она обеспечила активное использование шаблонов классов при объектно-ориентированном программировании на C++. Полный обзор парадигмы программирования с использованием библиотеки STL (*STL programming paradigm*) содержится в работе [3]. Библиотека STL является фундаментальной частью стандарта C++ [4]. Естественно, что использование шаблонов классов порождает специфические для такого программирования паттерны проектирования [5]. Примером является паттерн забавно рекуррентных шаблонов — *Curiously Reccuring Templets Pattern* (CRTP) [6]. Этот паттерн с необычным названием ссылается на общий класс методов, которые состоят в передаче производного класса, как параметра, одному из его базовых шаблонов классов. Простое применение паттерна CRTP состоит в отслеживании числа созданных объектов определенного класса. Это достигается увеличением значения статического поля — члена класса (целого типа) при каждом вызове конструктора и уменьшением его при каждом вызове деструктора. Можно сказать, что паттерн CRTP полезен при необходимости избежать реализации интерфейсов только из функций-членов (например, конструкторов, деструктора и оператора индекса). В настоящей работе рассматривается, как при проектировании программ на C++ с шаблонами классов используются канонические (стандартные) паттерны проектирования, представленные в работах [7—8]. Оказывается, что эти паттерны полезны при проектировании шаблонов арифметических выражений (*expression templents*), реализуя принцип открытости-закрытости (*Open Closed Principje (OCP)*) и обеспечивая повторное использование программных компонентов [9—12].

* Части 1, 2 см. в журналах "Программная инженерия" № 4, 5, 2011.

Шаблоны арифметических выражений

Шаблоны арифметических выражений — это метод метапрограммирования на основе шаблонов классов C++ (*C++ template metaprogremming*), в котором эти шаблоны используются для представления частей арифметических выражений. Как правило, сам шаблон класса представляет конкретный тип операции, в то время как параметры — любую операцию, к которой эта конкретная операция применяется. Шаблон арифметических выражений может впоследствии вычисляться или передаваться функции как параметр. Метод шаблонов арифметических выражений предложен и применен в работах [13—16]. Оказывается, что он существенно ускоряет вычисление арифметических выражений [17]. Ниже этот метод иллюстрируется на примере из работы [13] рекурсивного (времени компиляции) вычисления факториала (рис. 1 и 2).

```
template <int n>
struct Factorial {
  enum { ret = Factorial<n-1>::ret * n };
};

template <>
struct Factorial<0> {
  enum { ret = 1 };
};
```

Рис. 1. Шаблон класса `Factorial`

```
int main()
{
    cout << Factorial<4>::ret << endl;
    system(«Pause»);
    return 0;
}
/*
24
*/
```

Рис. 2. Функция `main()`, результат которой равен 24

Шаблон класса `Factorial` (см. рис. 1) не имеет ни полей членов (данных), ни функций-членов, он определяет только безымянный перечислимый тип (*unnamed enum type*), который имеет только одно значение. (Оказывается, что это перечислимое значение `factorial::ret` служит в качестве возвращаемого значения рассматриваемого вычисления (времени компиляции).) Для того, чтобы вычислить значение этого перечислимого типа для целого числа *n*, компилятор должен обратиться к другой версии шаблона факториала для $n - 1$, и так начинается рекурсивное подтверждение шаблона (*recursive template instantiotion*). Любая рекурсия нуждается в завершении. При использовании метода шаблонов классов конец рекурсии обеспечивается посредством специализации шаблонов (*template specialization*). На рис. 1 используется специализация шаблона для $n = 0$. На рис. 2 показана функция `main ()`, вычисляющая значение 4!.

Использование паттерна Composite при проектировании шаблонов арифметических выражений

UML-диаграмма паттерна Composite показана на рис. 3. Паттерн Composite дает способ представления отношения часть-целое (*part-whole relationship*), в котором клиент может игнорировать различие между индивидуальными объектами и композициями (коллекциями) объектов. Паттерн Composite использует идею, что структура дерева данных (*tree data structure*) легко конструируется и меняется, если тип узла дерева определяется полиморфно (*polymorphically*), так что узел либо лист (*leaf*), либо содержит список указателей на узлы-потомки (*child nodes*). Таким образом, дерево является в точности такой же сущностью, как его корень (*root*), что дает очень простые рекурсивные

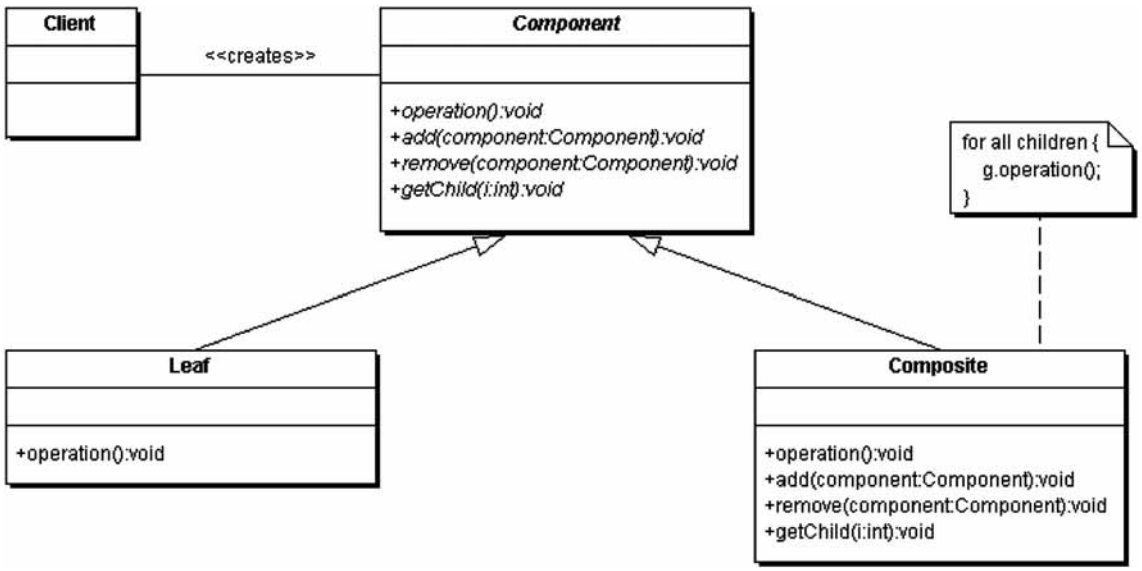

Рис. 3. UML-диаграмма паттерна Composite [8]

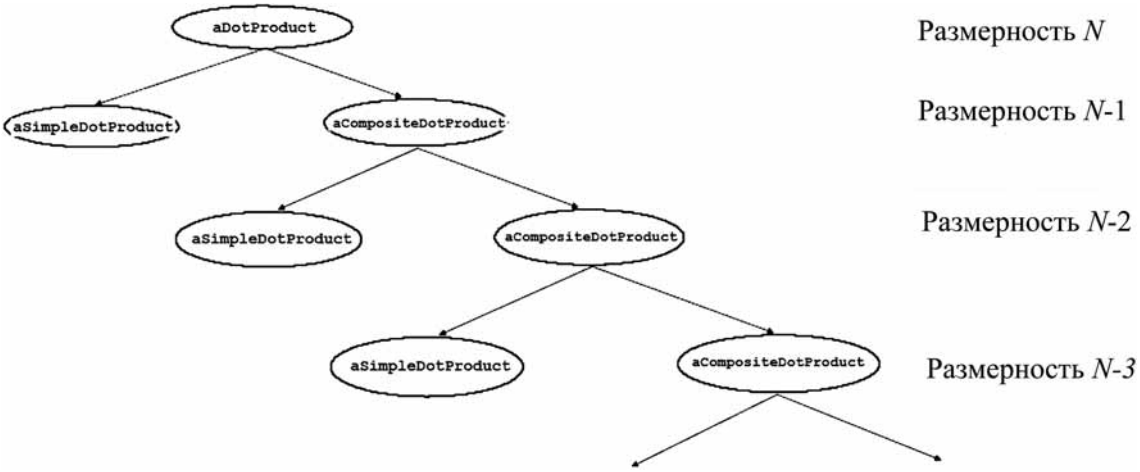

Рис. 4. Скалярное произведение как специальный случай паттерна Composite

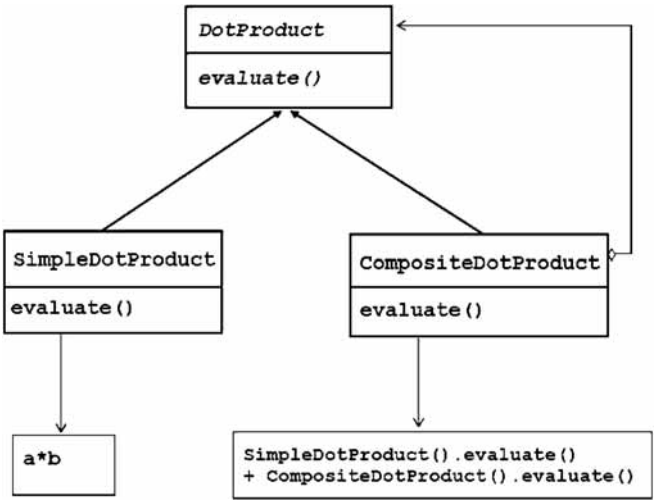

Рис. 5. Диаграмма классов Dot Product

алгоритмы работы с деревьями (*recursive tree-handling algorithmis*). Ключевыми элементами паттерна Composite являются лист и композит:

- лист (*leaf*) определяет поведение примитивных объектов в этой композиции;
- композит (*composite*) определяет поведение коллекций листьев.

Вычисление скалярного произведения двух векторов (*vector dot product*) может представляться в виде дерева, производимого паттерном Composite (рис. 4) [15—16]. На рис. 4 скалярное произведение (*dot product*) расщепляется на лист (а именно, скалярное произведение двух векторов размерности единица) и композит (а именно, скалярное произведение двух векторов размерности $N - 1$) и т. д.

Диаграмма паттерна Composite (см. рис. 3) дает диаграмму классов для шаблона скалярного произведения векторов (*class diagram of Dot Product*) (рис. 5).

На рис. 6 представлены классы иерархии *Dot Product* (см. рис. 5), а на рис. 7 функция `dot(.)`, реализующая логику вычислений, представленную на рис. 4, и функция `main()`.

Паттерн Interpreter

Паттерн Interpreter предполагает, что предметная область моделируется рекурсивной грамматикой (*recursive grammar*). Каждое правило (*rule*) грамматики является либо композитом (правилом, которое ссылается на другие правила), либо терминалом (*terminal*) (листом в структуре дерева, представляющего рекурсивную грамматику). Паттерн Interpreter использует рекурсивный обход паттерна Composite для интерпретации предложений (*sentences*) и определяет грамматическое представление для языка и интерпретатор его грамматики. UML-диаграмма паттерна Interpreter показана на рис. 8.

При представлении рекурсивной грамматики посредством паттерна Interpreter, иерархия классов C++ подобна иерархии классов C++ для представления этой грамматики посредством паттерна Template Method [8]. Это демонстрирует (реализующая систему продукций (см. Часть 1 статьи, "Программная инженерия" № 4, 2011, рис. 3)) иерархия классов, обеспечивающих переход от римской к позиционной системе счисления (см. Часть 1 статьи, "Программная инженерия" № 4, 2011, рис. 4 и 5). При представлении грамматики посредством паттерна Template Method базовый класс `RNInterpreter` объявляется абстрактным (см. Часть 1 статьи, "Программная инженерия" № 4, 2011, рис. 4). При этом используется инновация: полиморфный метод `RNInterpreter::interpret()` существует в двух версиях — для клиента и для классов, производных от `RNInterpreter`. При представлении грамматики посредством паттерна Interpreter базовый класс `RNInterpreter` конкретный, поэтому все (`protected`) виртуальные функции реализованы (рис. 9). При этом "слоеная" структура интерфейса

```
template <class T>
class DotProduct  {
   public:
      virtual T evaluate() = 0;
        virtual ~DotProduct () {
        }
};

template <class T>
class SimpleDotProduct  : public DotProduct<T> {
      private:
      T* v1; T* v2;
   public:
      SimpleDotProduct (T* a, T* b) : v1(a), v2(b) {
              cout<<"SimpleDotProduct::SimpleDotProduct"<<endl;
      }
      virtual T evaluate () {
              cout<<"SimpleDotProduct::evaluate()"<<endl;
              return (*v1)*(*v2);
      }
      virtual ~SimpleDotProduct () {
              cout<<"SimpleDotProduct::~SimpleDotProduct"<<endl;
      }
};

template <class T>
class CompositeDotProduct  : public DotProduct<T> {
      protected:
            SimpleDotProduct<T>* s;
            CompositeDotProduct<T> * c;
      public:
            CompositeDotProduct (T* a, T* b, size_t dim):
            s(new SimpleDotProduct<T>(a,b)),
            c((dim==1)? 0:new CompositeDotProduct<T>(a+1,b+1,dim-1)){
                  cout<<"CompositeDotProduct::CompositeDotProduct"<<endl;
            }
            virtual T evaluate()
            {
                  cout<<"CompositeDotProduct:: evaluate()"<<endl;
                  return (s-> evaluate() + ((c)? c-> evaluate():0));
            }

  virtual ~CompositeDotProduct () {
        cout<<"CompositeDotProduct::~CompositeDotProduct"<<endl;
        delete c;
        delete s; }
};
```

Рис. 6. Классы иерархии Dot Product

```
template <class T> T dot(T* a, T* b, size_t dim)
{
      cout<<"Dot:"<<endl;
      cout <<endl;
      return (dim==1)? SimpleDotProduct<T>(a,b).evaluate()
         : CompositeDotProduct<T>(a,b,dim).evaluate();
}

int main()
{
      int a[4] = {1,2,3,4};
      int b[4] = {4,3,2,1};
      cout << dot(a,b,4)<<endl;
      system("Pause");
      return 0;
}
/*
20
*/
```

Рис. 7. Функция `dot(.)` и функция `main()` с результатом 20

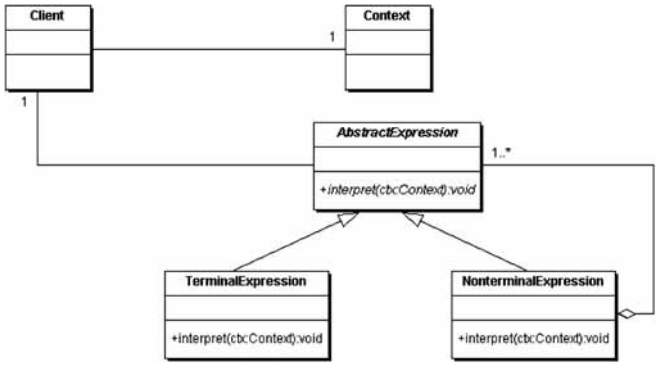

Рис. 8. UML-диаграмма паттерна Intepreter [8]

класса `RNInterpreter` усиливается. Поскольку конструктор конкретного базового класса `RNInterpreter()` (см. рис. 9) должен создавать объекты производных классов (в динамической памяти), то чтобы избежать бесконечного цикла, вводится дополнительный (параметризованный) конструктор `RNInterpreter(int)` для вызова в конструкторах производных классов. Классы, производные от `RNInterpreter`, показаны на рис. 10.

После определения производных классов определяется тело метода (для 1 клиента) `int RNInterpreter::interpret(char*)` (рис. 11), который посредством указателей производных классов вызывает полиморфный метод `void RNInterpreter::interpret(char*, int&)`.

Клиентская (пользовательская) функция `main ()` и результат перевода римских цифр показаны на рис. 12.

Использование паттерна Interpreter при проектировании шаблонов арифметическихвыражений

UML-диаграмма паттерна Interpreter (см. рис. 8) определяет структуру классов при проектировании шаблона арифметических выражений (рис. 13) [18]. Диаграмма на рис. 13 содержит три абстрактных класса: корневой класс иерархии `AbstractExpression` и производные от него `TerminalExpression` и `NonTerminalExpression`. Классы `Literal`, `Variable`, `UnaryExpression` и `BinaryExpression` являются конкретными.

```
class TenThousand; class Thousand; class Hundred; class Ten; class One;

class RNInterpreter {
public:
      RNInterpreter() {
// ctor для клиента вызывает 1-arg ctor чтобы избежать бесконечного цикла
      thousands = new Thousand(1);  hundreds = new Hundred(1);
      tens      = new Ten(1);       ones     = new One(1);
      }
      RNInterpreter( int ) { }  // ctor для производных классов
      int interpret( char* );   // метод interpret() для клиента
      virtual void interpret( char* input, int& total )
           {// метод interpret() для производных классов
            int  index;  index = 0;
            if ( ! strncmp(input, nine(), 2)){total += 9 * multiplier();
                  index += 2; }
            else if ( ! strncmp(input, four(), 2)) {total += 4 * multiplier();
                  index += 2; }
            else  {
                  if (input[0] == five()) { total += 5 * multiplier();
                        index = 1; }
                  else index = 0;
                  for (int end = index + 3 ; index < end; index++)
                        if (input[index] == one()) total += 1 * multiplier();
                        else break;
            }
            strcpy( input, &(input[index]));
      }     // remove leading chars processed
protected:  // cannot be pure virtual because client asks for instance
      virtual char  one()  { return ' '; }
      virtual char* four() { return ""; }
      virtual char  five() { return ' '; }
      virtual char* nine() { return ""; }
      virtual int   multiplier() {return 1; }
private:
      RNInterpreter*  thousands;              RNInterpreter*  hundreds;
      RNInterpreter*  tens;                   RNInterpreter*  ones;
};
```

Рис. 9. Базовый класс `RNInterpreter`

```
class Thousand : public RNInterpreter {
public:  // provide 1-arg ctor to avoid infinite loop in base class ctor
      Thousand( int ) : RNInterpreter(1) { }
protected:
      char  one()  { return 'M'; }
      char* four() { return ""; }
      char  five() { return '\0'; }
      char* nine() { return ""; }
      int   multiplier() { return 1000; }
};
class Hundred : public RNInterpreter {
public:
      Hundred( int ) : RNInterpreter(1) { }
protected:
      char  one()  { return 'C'; }
      char* four() { return "CD"; }
      char  five() { return 'D'; }
      char* nine() { return "CM";}
      int   multiplier() { return 100; }
};
class Ten : public RNInterpreter {
public:
      Ten( int ) : RNInterpreter(1) { }
protected:
      char  one()  { return 'X'; }
      char* four() { return "XL";}
      char  five() { return 'L'; }
      char* nine() { return "XC"; }
      int   multiplier() { return 10; }
};
class One : public RNInterpreter {
public:
      One( int ) : RNInterpreter(1) { }
protected:
      char  one() { return 'I'; }
      char* four(){ return "IV";}
      char  five(){ return 'V'; }
      char* nine(){ return "IX";}
      int   multiplier() { return 1; }
};
```

Рис. 10. Классы, производные от `RNInterpreter`

```
Int RNInterpreter::interpret( char* input )
{
      int  total;  total = 0;
      thousands->interpret( input, total );
      hundreds->interpret( input, total );
      tens->interpret( input, total );
      ones->interpret( input, total );
      if (strcmp(input, “”))  // if input was invalid, return 0
            return 0;
      return total; }
```

Рис. 11. Тело метода (для клиента) `int RNInterpreter::interpret (char*)`

На рис. 14 представлены абстрактные классы шаблона арифметических выражений из диаграммы на рис. 13. Классы `Literal` и `Variable` представлены на рис. 15. Классы `UnaryExpression` и `BinaryExpression` представлены на рис. 16. Конкретное значение передается в выражение через параметр `v` метода `result_type evaluate (result_type v) const`. Метод `Literal: :evaluate()` игнорирует значение этого параметра, а `Variable: :evaluate()` отражает значение этого параметра (см. рис. 15). Методы `UnaryExpression: :evaluate()` и `BinaryExpression::evaluate()` передают значение этого параметра в рекурсивный вызов (см. рис. 16).

Среди классов на рис. 16 присутствуют отображения типа или трейты *(traits)* `template < class > struct Representation < >` [19—20]. Шаблон класса `Representation` абстрагируется от типов классов выражений (*expression class types*) и обеспечивает под-

```
int main(){
      RNInterpreter interpreter;
      char          input[20];
      cout << "Enter Roman Numeral: ";
      while (cin >> input){
      cout << "   interpretation is " << interpreter.interpret(input) << endl;
      cout << "Enter Roman Numeral: "; }
      system("Pause");
      return 0;
}

// Enter Roman Numeral: MCMXCVI
//    interpretation is 1996
// Enter Roman Numeral: MMMCMXCIX
//    interpretation is 3999
// Enter Roman Numeral: MMMM
//    interpretation is 0
// Enter Roman Numeral: MDCLXVIIII
//    interpretation is 0
// Enter Roman Numeral: CXCX
//    interpretation is 0
```

Рис. 12. Клиентский код и результат его выполнения

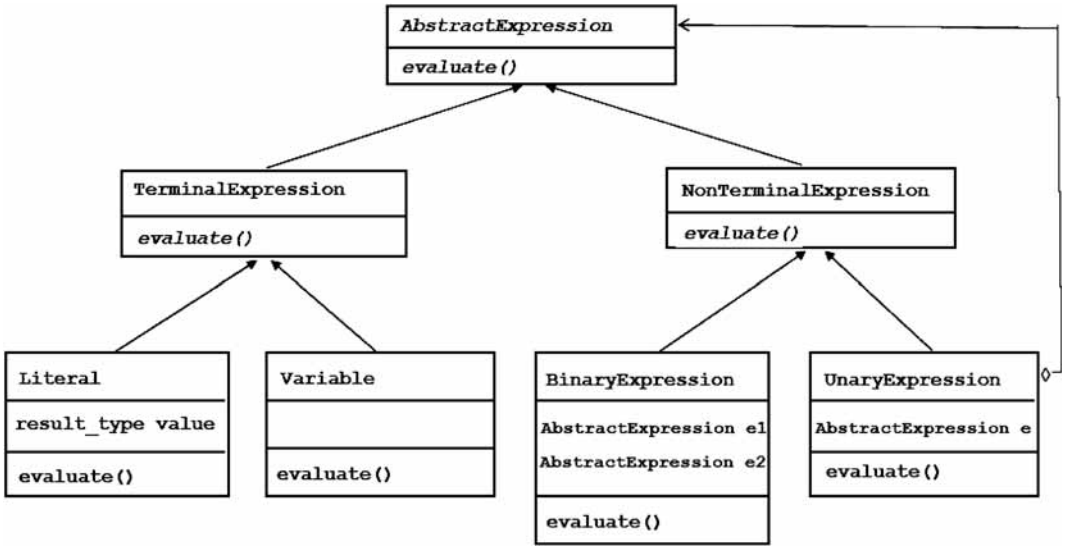

Рис. 13. Диаграмма паттерна Interpreter для проектирования шаблона арифметических выражений

```
// The type of the value returned by evaluate().
typedef double result_type;

class AbstractExpression {
public:
      virtual result_type evaluate(result_type) const = 0;
      ~AbstractExpression () {}
};

class TerminalExpression : public AbstractExpression{
public:
      ~TerminalExpression (){}
};

class NonTerminalExpression : public AbstractExpression{
public:
      ~NonTerminalExpression (){}
};
```

Рис. 14. Абстрактные классы шаблона арифметических выражений

```
class Literal : public TerminalExpression{
public:
 // Create an expression that always evaluates to the given value.
      Literal(result_type v) : value(v) { }
 // Evaluate this expression.
    result_type evaluate(result_type) const { return value; }
private:
      result_type value;
};

class Variable : public TerminalExpression{
public:
  // Evaluate this expression.
     result_type evaluate(result_type v) const { return v; }
};
```

Рис. 15. Классы `Literal` и `Variable`

```
// The default type-representation mapping.
template <class ValueType>
struct Representation { typedef ValueType type; };

// Type-representation mapping specializations for pod types.
template <>
struct Representation<int> { typedef Literal type; };

template <>
struct Representation<float> { typedef Literal type; };

template <>
struct Representation<double> { typedef Literal type; };

template < class Operand, class UnaryOp >
class UnaryExpression : public NonTerminalExpression{
public:
  // Create an expression that evaluates to the result of applying the given
  // unary operation to the result of evaluating the given expression.
     UnaryExpression(Operand e, UnaryOp op = UnaryOp()): expr(e), oprtr(op) { }
    // Return the result of evaluating this expression.
     result_type evaluate(result_type v) const
      { return oprtr(expr.evaluate(v)); }
private:
      typename Representation<Operand>::type expr;
      UnaryOp oprtr;
};

template < class LeftOperand, class RightOperand, class BinaryOp >
class BinaryExpression : public NonTerminalExpression{
public:
    // Create an expression that evaluates to the result of applying the given
    // binary operation to the result of evaluating the given expressions.
       BinaryExpression(LeftOperand l, RightOperand r, BinaryOp op =
BinaryOp()): left(l), right(r), oprtr(op) { }
    // Return the result of evaluating this expression.
       result_type evaluate(result_type v) const
       { return oprtr(left.evaluate(v), right.evaluate(v)); }
private:
      typename Representation<LeftOperand>::type left;
      typename Representation<RightOperand>::type right;
      BinaryOp oprtr;
};

#define def_binexp(_op1, _op2) \
  template <class LExpr, class RExpr> \
  static BinaryExpression <LExpr, RExpr, std::_op2<result_type> > \
  operator _op1 (LExpr l, RExpr r) { \
      return BinaryExpression<LExpr, RExpr, std::_op2<result_type> >(l, r); }
```

Рис. 16. Классы `UnaryExpression` и `BinaryExpression`

```
template < class Expression >
static result_type
find_max(Expression expr, double l, double r, double s){
      result_type mx = expr.evaluate(l);
      while (l < r)
      {     mx = std::max(expr.evaluate(l), mx);
            l += s; }
      return mx;
}

int main(){
  Variable x;
  cout << find_max(x/(x*x + 5), 1, 10, 0.01) << endl;
  system("Pause");
  return 0;
}
/*
0.223606
*/
```

Рис. 17. Шаблон функции `find_max()` для определения максимума выражения и функция `main()` с результатом 0.223606

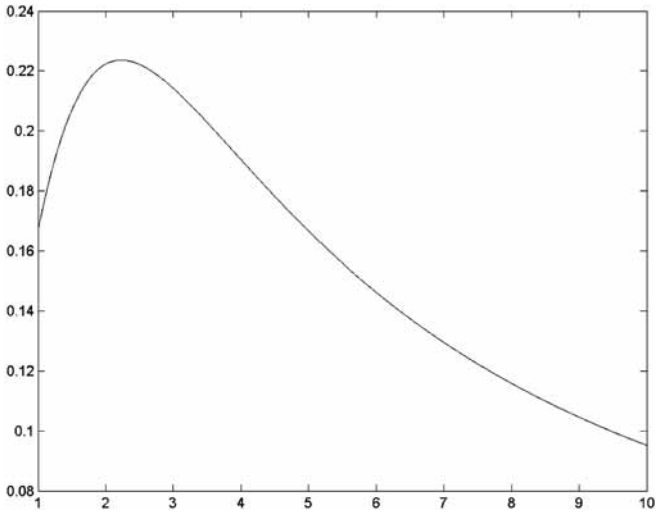

Рис. 18. График функции $\frac{x}{x^2+5}$, $x \in [0, 10]$

гонку (приспособление) типа (*type-based customization*). Тип `Representation` при специализации классом выражением (*expression class*) `T` — это `Representation <T>: :type` и он тот же самый, что и `T`. Трейты широко используются в библиотеке STL [3].

На рис. 17 показан пример использования шаблона арифметического выражения, представленного на рис. 14—16 для определения максимума функции $\frac{x}{x^2+5}$. Анализ показывает, что эта функция имеет максимум при $x = \sqrt{5}$, и максимум равен $1/2\sqrt{5} \approx$ $\approx 0{,}223606797749979$. График функции $\frac{x}{x^2+5}$, $x \in [0, 10]$ показан на рис. 18.

Паттерн Flyweight

Паттерн Flyweight (Приспособленец) описывает, как правильно разделять объекты на части для сокращения затрат памяти. Для этого каждый объект — кандидат на Flyweight делится на две части: зависящую от состояния (*state-dependent*, внешнюю, несущественную) часть и независящую от состояния (*state-independent*, внутреннюю, существенную) часть. Существенная часть (*intrinsic state*) остается в объекте Flyweight. Несущественная часть (*extrinsic state*) хранится или вычисляется клиентом (*client objects*), и передается объекту Flyweight при вызове его методов. Покажем это на примере. На рис. 19 приведен (базовый) класс двумерных точек `Point2D`, а на рис. 20 две альтернативы определения класса окружностей: `Circle1` —

```
class Point2D {
private:
      double _x, _y;
public:
      Point2D (){ _x = 0; _y = 0;
      }
      Point2D (double x, double y) { _x = x; _y = y; }
      void put_xy (double x, double y) { _x = x; _y = y; }
      void display () { cout << _x << " " << _y <<endl; }
};
```

Рис. 19. Класс двумерных точек `Point2D`

```
class Circle1 // инкапсуляция
{
private:
      Point2D _point;
      double _r;
public:
      Circle1(double x, double y, double r) { _point.put_xy(x,y); _r = r; }
      void display () { _point.Point2D::display(); cout << _r << endl; }
};

class Circle2 : public Point2D // наследование
{
private:
      double _r;
public:
      Circle2(double x, double y, double r):Point2D(x, y) { r = _r; }
      void display () { Point2D::display(); cout << _r << endl; }
};
```

Рис. 20. Два объявления класса окружностей

```
class Circle // Flyweight
{
private:
      double x, y;
public:
      Circle (double _x, double _y) { x = _x; y = _y; }
      void display (double r) { cout << x << " " << y << " " << r << endl; }
};
```

Рис. 21. Класс окружностей Flyweight

через инкапсуляцию объекта `Point2D` и `Circle2` — через наследование класса `Point2D`.

Третья альтернатива (инкапсуляции и наследованию) — это класс окружностей Flyweight (рис. 21).

Однако при определении производящего метода фабрики (`Factory`) для объектов `Circle` (см. рис. 21) оказывается, что потенциальная мощность множества объектов `Circle` гораздо меньше множества объектов `Circle1` или `Circle2` (см. рис. 20). Таким образом, класс `Factory` для объектов `Circle` (т. е. `Flyweight`) может быть определен, только если ограничиться производством нескольких семейств концентрических окружностей. Например, это могут быть четыре семейства концентрических окружностей с центрами в точках с координатами (x, y): (1, 1), (1, −1), (−1, −1) и (−1, 1) (рис. 22).

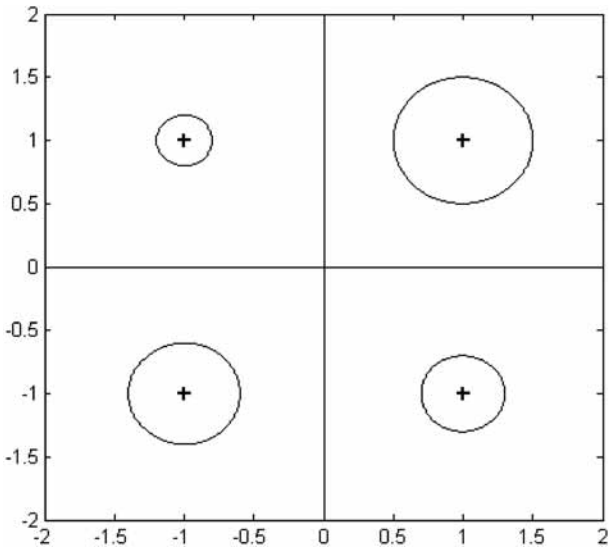

Рис. 22. Четыре семейства концентрических окружностей

Для центров в этих точках существует проективное преобразование $(x - 2y + 3)/2 \Rightarrow i$ (это преобразование не единственное), отображающее эти точки в различные значения индекса $i \in \{0, 1, 2, ...\}$ (желательно, без пропусков), как показано в ниже.

(x, y) . . . . . $.i$
(1, 1) 1
(1, −1) 3
(−1, −1) 2
(−1, 1) 0

Существование такого проективного преобразования (отображения) является необходимым условием для определения статического производящего метода `Circle* Factory::getCircle(double, double)` (рис. 23). Пример функции `main()`, которая создает и демонстрирует "концентрические окружности", приведен на рис. 24.

```
class Factory
{
public:
      static Circle* getCircle (double x, double y)
      {
            int i = static_cast<int>((x-2*y+3)/2); // отображение(x,y)→i
            if (_circle[i]==NULL) _circle[i] = new Circle (x,y);
            return _circle[i];
      }
      static void report()
      {
            for (int i=0; i<4; i++) {
                  if (_circle[i]!=NULL) _circle[i]->display(0.0);
            }}
private:
      static Circle* _circle[4];
};

Circle* FlyFactory :: _circle[4] = {NULL, NULL, NULL, NULL};
```

Рис. 23. Класс `Factory` для объектов `Circle` (рис. 21)

```
int main()
{
      for (float i =0; i<10; i++)
      {
            (FlyFactory::getCircle(-1,1))->display(i/10);
            (FlyFactory::getCircle (1,-1))->display(i/10);
      }
      FlyFactory::report();
      for (float i =0; i<100; i++)
      {
            (FlyFactory::getCircle(-1,1))->display(i/100);
      }
      FlyFactory::report();
      system("Pause");
      return 0;
}
```

Рис. 24. Пример функции `main()`

Паттерн Flyweight в библиотеке Boost

Как отмечается в Части 1 статьи (см. "Программная инженерия" № 4, 2011) [12], паттерны проектирования из канонического (стандартного) набора [7—8] легко гибридизуются, т. е. реализуются одной и той же иерархией классов. Большие деревья Composite, которые имеют высокую степень дублирования узлов (*nodes*) и поддеревьев (*subtrees*) (примером являются деревья, которые генерируются при разборе компьютерной программы), естественно подходят для идиомы Flyweight: простое превращение типа узла в Flyweight автоматически решает проблему дублирования на уровне узлов и поддеревьев. Пример гибридизации паттернов Flyweight и Composite (см. рис. 3) содержится в демонстрационной программе библиотеки Boost [21]. В настоящее время библиотека Boost C++ (Boost C++ Libraries) является наиболее профессионально спроектированной библиотекой на C++ [22]. Демонстрационная программа проводит разбор списков в стиле языка Lisp, имеющих форму $(a_1, a_2, ..., a_n)$, где каждый a_i, $i = \overline{1, n}$ — это терминальная строка (*terminal string*) или список (*list*). Разобранная структура данных — это тип Composite, определенный с использованием Boost.Flyweight в сочетании с рекурсивными возможностями Boost.Variant [21]. Т. е. узел Lisp-списка моделируется как `boost::variant`:

```
list      ::= flyweight<list_impl>

list_impl ::= std::string | std::vector<list>
```

Рис. 25. Структура данных в стиле Бэкуса—Наура

```
/* Boost.Flyweight example of a composite design.
 *
 * Copyright 2006-2008 Joaquin M Lopez Munoz.
 * Distributed under the Boost Software License, Version 1.0.
 * (See accompanying file LICENSE_1_0.txt or copy at
 * http://www.boost.org/LICENSE_1_0.txt)
 *
 * See http://www.boost.org/libs/flyweight for library home page.
 */
#include <boost/flyweight.hpp>
#include <boost/functional/hash.hpp>
#include <boost/tokenizer.hpp>
#include <boost/variant.hpp>
#include <boost/variant/apply_visitor.hpp>
#include <boost/variant/recursive_wrapper.hpp>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
//using namespace std;
using namespace boost::flyweights;
struct list_elems;
typedef boost::variant<
  std::string,
  boost::recursive_wrapper<list_elems>
> list_impl;
struct list_elems:std::vector<flyweight<list_impl> >{};

typedef flyweight<list_impl> list;
struct list_hasher:boost::static_visitor<std::size_t>
{
  std::size_t operator()(const std::string& str)const
  {
    boost::hash<std::string> h;
    return h(str);
  }

  std::size_t operator()(
    const boost::recursive_wrapper<list_elems>& elmsw)const
  {
    const list_elems& elms=elmsw.get();
    std::size_t res=0;
    for(list_elems::const_iterator it=elms.begin(),it_end=elms.end();
        it!=it_end;++it){
      const list_impl* p=&it->get();
      boost::hash_combine(res,p);
    }
    return res;
  }
};
#if defined(BOOST_NO_ARGUMENT_DEPENDENT_LOOKUP)
namespace boost{
#endif
std::size_t hash_value(const list_impl& limpl)
{
  return boost::apply_visitor(list_hasher(),limpl);
}
#if defined(BOOST_NO_ARGUMENT_DEPENDENT_LOOKUP)
} /* namespace boost */
#endif
```

Рис. 26. Boost.Flyweight в сочетании с рекурсивными возможностями Boost.Variant

строка (*string*, примитивный узел), или вектор узлов (*vector of nodes*, вложенный список).

Таким образом, как это обычно бывает с рекурсивными структурами, узел может рассматриваться также как список. Для экономии памяти, вектор узлов хранится как вектор (облегченных) объектов типа Flyweight. Чтобы завершить цикл паттерна Flyweight, окончательный тип списка — это оболочка (*wrapper*) Flyweight, так что окончательная структура данных описывается в стиле Бэкуса—Наура (*BNF-like style*) (рис. 25).

Например, для списка (=(tan(+*x y*))(/(+(tan *x)* × × (tan *y*))(−1(*(tan *x*)(tan *y*))))) результирующая структура данных неявно обнаруживает дублирующие вхождения +, *x*, *y*, tan, (tan *x*) и (tan *y*). Пример программы библиотеки Boost C++ для разбора Lisp-списка приведен на рис. 26—28 [21].

Пример функции `main()` приведен на рис. 29, а арифметическое выражение (= (tan(+ *x y*))(/(+(tan *x*) × × (tan *y*))(-1(*(tan *x*)(tan *y*))))) в префиксной форме и дерево его разбора на рис. 30.

```
/* basic pretty printer with indentation according to the nesting level */

struct list_pretty_printer:boost::static_visitor<>
{
  list_pretty_printer():nest(0){}

  void operator()(const std::string& str)
  {
    indent();
    std::cout<<str<<std::endl;
  }

  void operator()(const boost::recursive_wrapper<list_elems>& elmsw)
  {
    indent();
    std::cout<<"("<<std::endl;
    ++nest;
    const list_elems& elms=elmsw.get();
    for(list_elems::const_iterator it=elms.begin(),it_end=elms.end();
        it!=it_end;++it){
      boost::apply_visitor(*this,it->get());
    }
    --nest;
    indent();
    std::cout<<")"<<std::endl;
  }

private:
  void indent()const
  {
    for(int i=nest;i--;)std::cout<<"  ";
  }

  int nest;
};

void pretty_print(const list& l)
{
      std::cout<<"data structure : "<<std::endl;
      list_pretty_printer pp;
      boost::apply_visitor(pp,l.get());
}
```

Рис. 27. Класс для печати дерева с отступами, показывающими уровни вложенности

```
/* list parser */

template<typename InputIterator>
list parse_list(InputIterator& first,InputIterator last,int nest)
{
  list_elems elms;
  while(first!=last){
    std::string str=*first++;
    if(str=="("){
      elms.push_back(parse_list(first,last,nest+1));
    }
    else if(str==")"){
      if(nest==0)throw std::runtime_error("unmatched )");
      return list(elms);
    }
    else{
      elms.push_back(list(str));
    }
  }
  if(nest!=0)throw std::runtime_error("unmatched (");
  return list(elms);
}

list parse_list(const std::string str)
{
  typedef boost::tokenizer<boost::char_separator<char> > tokenizer;
  tokenizer tok(str,boost::char_separator<char>(" ","()"));
  tokenizer::iterator begin=tok.begin();
  return parse_list(begin,tok.end(),0);
}
```

Рис. 28. Шаблон функции для разбора Lisp-списка

```
int main()
{
  std::cout<<"enter list: ";
  std::string str;
  std::getline(std::cin,str);
  std::cout<<std::endl;
  try{
    pretty_print(parse_list(str));
  }
  catch(const std::exception& e){
    std::cout<<"error: "<<e.what()<<std::endl;
  }
  system("Pause");
  return 0;
}

/*Revised $Date: 2009-06-07 12:46:25 -0400 (Sun, 07 Jun 2009) $
Copyright Beman Dawes, David Abrahams, 1998-2005.
Copyright Rene Rivera 2004-2008.
Distributed under the Boost Software License, Version 1.0.
XHTML 1.0
CSS
OSI Certified
*/
```

Рис. 29. Пример функции `main()`

```
enter list: (= (tan (+ x y))(/ (+ (tan x)(tan y))(- 1 (* (tan x)(tan y)))))

data structure :
(
  (
    =
    (
      tan
      (
        +
        x
        y
      )
    )
    (
      /
      (
        +
        (
          tan
          x
        )
        (
          tan
          y
        )
      )
      (
        -1
        (
          *
          (
            tan
            x
          )
          (
            tan
            y
          )
        )
      )
    )
  )
)
```

Рис. 30. Результат функции `main()` с рис. 29 в виде дерева разбора арифметического выражения

Список литературы

1. **Vandevoorde P., Josuttis N.** C++ Templates: The Complete Guide. Addison Wesley Publishing Company, Inc., 2002.
2. **Stepanov A., Lee M.** The Standard Template Library. Palo Alto, CA: Hewlett Packard Laboratories, 1995.
3. **Matthew H. A.** Generic Programming and the STL — Using and Extending the C++ Standard Template Library. Professional Computing-Series. Addison-Wesley, 1999.
4. **Stroustrup B.** C++ in 2005. Extended foreword to "Design and Evolution of C++". Addision Wesley, 2005. 32 p.
5. **Duret-Lutz A., Géraud T., Demaille A.** Design patterns for generic programming in C++ // Proc. of the 6th USENIX Conference on Object-Oriented Technologies and Systems. 2001. January 29—February 2. San Antonio, Texas, USA. P. 189—202.
6. **Coplien J.** Curiously recurring template pattern / Lippman S. B. ed. C++ Gems. Cambridge University Press & Sigs Books. 1996. P. 135—144.
7. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.
8. **Huston V.** Design Patterns in a Nutshell. Sebastopol, CA: O'Reilly & Associates, 2007.
9. **Garlan P., Allen R., Ockerbloom J.** Architectural mismatch or, Why it's hard to build systems out of existing parts // IEEE Software. 1995. V. 12. Issue 6. P. 17—26.
10. **Garlan D., Allen R., Ockerbloom J.** Architectural mismatch: Why reuse is still so hard // IEEE Software. 2009. July. V. 26. Issue 4. P. 66—69.
11. **Martin R. C.** The open-closed principle / More C++ gems. New York, NY: Cambridge University Press. 2000. P. 97—112.
12. **Кухаренко Б. Г.** Принцип открытости-закрытости в программной инженерии и паттерны проектирования. Часть 1 // Программная инженерия. 2011. № 4. С. 20—25.
13. **Veldhuizen T. L.** Expression templates // C++ Report. 1995. V.7. No. 5. P. 26—31 (reprinted in Lippman S. B. ed. C++ Gems. Cambridge University Press & Sigs Books. 1996. P. 475—487).
14. **Veldhuizen T. L.** Using C++ template metaprograms // C++ Report. 1995. May. V. 7. No 4. P. 36—43.
15. **Veldhuizen T. L.** Blitz++: The library that thinks it is a compiler // Proc. of SciTools 98 Conference. Oslo, Norway. 1998. September 10. P. 1—36.
16. **Veldhuizen T. L.** Blitz++ User's Guide. A C++ Class Library for Scientific Computing. Version 0.9. Free Software Foundation, Inc., 2006.
17. **Haney S., Crotinger J.** How templates enable high-performance scientific computing in C++ // IEEE Computing in Science and Engineering. 1999. V. 10. No. 4. P. 66—72.
18. **Langer A., Kreft K.** C++ expression templates // C/C++ Users Journal. 2003. March. P. 1—12.
19. **Myers N. C.** Traits: a new and useful template technique // C++ Report. 1995. June. V. 70. No. 5. P. 32—35.
20. **Maddock J., Cleary S.** C++ type traits // Dr. Dobb's Journal of Software Tools. 2000. October. V. 25. No 10. P. 38—44.
21. **Gurtovoy A., Abrahams D.** The Boost C++ Metaprogramming Library. Addison-Wesley Publishing Company, Inc., 2002.
22. **Sutter H., Alexandrescu A.** C++ Coding Standards. Boston, Ma: Addison-Wesley Publishing Company, Inc., 2005.

УДК 004.75; 004.415, 004.946

В. Ю. Харитонов, канд. техн. наук, мл. науч. сотр., Московский энергетический институт (технический университет),
e-mail: kharitonovvy@gmail.com

Инструментальные программные средства для построения распределенных систем виртуальной реальности. Часть I*

Статья посвящена важной и актуальной в настоящее время проблеме создания распределенных систем виртуальной реальности. Предлагается комплексный подход, основанный на разработке специализированных инструментальных программных средств, призванных упростить и одновременно ускорить процесс создания конечной системы.

В первой части статьи приводится типизация и краткий обзор существующих распределенных систем виртуальной реальности, рассматриваются основные свойства таких систем, предлагаются программная архитектура и высокоуровневый сетевой протокол, представляющие собой основу для их построения.

Ключевые слова: *распределенные системы виртуальной реальности, обеспечение согласованности данных, распределенное моделирование, вычислительные сети, трехмерная интерактивная компьютерная графика*

Распределенные системы виртуальной реальности (РСВР) представляют одно из наиболее интенсивно развивающихся направлений распределенных систем, позволяя организовать взаимодействие множества географически удаленных пользователей в общей для них виртуальной среде, что может быть востребовано во многих прикладных областях, таких как промышленность, образование, медицина и др. В основе любой РСВР лежат специализированные программные механизмы, которые должны обеспечивать согласованное и своевременное взаимодействие множества неоднородных *компонентов*, входящих в ее состав. Реализация данных механизмов является нетривиальной задачей, требующей применения низкоуровневого сетевого программирования, что влечет значительные временные и материальные затраты, а также предъявляет повышенные требования к квалификации разработчика. В этом свете актуальной является разработка инструментальных средств, призванных упростить и одновременно ускорить процесс создания конечной РСВР.

В данной работе предлагается подход к построению инструментальных средств для создания РСВР, основанный на разработке специализированного программного обеспечения (ПО) промежуточного уровня. Предлагаемый программный комплекс TerraNet (далее — библиотека TerraNet) позволяет значительно упростить процесс разработки и развертывания РСВР, предоставляя высокоуровневый программный интерфейс для создания, управления и распределения объектов в рамках единой виртуальной среды.

Статья публикуется в двух частях. В первой части основное внимание уделяется фундаментальным и архитектурным аспектам построения РСВР. Рассматриваются основные свойства таких систем, предлагаются программная архитектура и высокоуровневый сетевой протокол, представляющие собой основу для построения РСВР.

* Работа выполнена при поддержке РФФИ (грант № 11-07-00751-а), Совета по грантам Президента РФ (НШ-7239.2010.9), аналитической ведомственной целевой программы "Развитие научного потенциала высшей школы" (проекты 2.1.2/6718, 2.1.2/13283) и Федеральной целевой программы "Научные и научно-педагогические кадры инновационной России" (Государственный контракт П2227, 16.740.11.0038).

1. Обзор существующих решений

О разработках РСВР в нашей стране практически ничего не известно. Большинство известных систем разработано за рубежом [1]. Условно их можно разделить на три больших класса: *тренажерные РСВР, сетевые виртуальные среды и многопользовательские сетевые компьютерные игры.*

Первый тип систем, в основном, используется в военных разработках для подготовки личного состава. Такие системы представляют собой коллективные тренажеры различного назначения — от авиасимуляторов до симуляторов проведения наземных операций. Наиболее известными представителями данного класса РСВР являются системы SIMNET (*SIMulation NETwork*) и ее последователи, ставшие промышленными стандартами — DIS (*Distributed Interactive Simulation,* IEEE Standard 1278) и HLA (*High Level Architecture*, IEEE Standard 1516) [2].

Следующий тип РСВР, сетевые виртуальные среды (*Networked Virtual Environments*) — это в основном научные и учебные разработки. Подобные системы нашли применение в различных исследовательских задачах (например, в дистанционном обучении) и в других областях, требующих удаленного взаимодействия многих участников (например, в медицине [3]). Примеры подобных систем: DIVE, RING, NPSNET [4], MASSIVE [5], MR Toolkit, Avocado [6].

Многопользовательские сетевые компьютерные игры представляют собой наиболее динамично развивающийся и массовый класс среди всех РСВР. Они вобрали в себя самые современные технологии данной отрасли. Среди наиболее успешных можно выделить игры серий *Unreal Tournament*, *Counter Strike*: *Source* [7] и *Quake*. Сейчас растет интерес к новому жанру сетевых игр — многопользовательским онлайн-играм (*Massively Multiplayer Online Game*, MMOG) [8], отличительной чертой которого является поддержка особенно большого числа пользователей (в настоящий момент — до сотен тысяч). Подобные исследования в области повышения масштабируемости РСВР выходят на класс GRID-систем, позволяющих выполнять расчет виртуальных миров распределенно [9].

Из проведенного анализа рассмотренных РСВР [10] видно, что:

- почти все системы разработаны за рубежом, информация по многим ключевым вопросам, связанным с обеспечением согласованности данных, отсутствует;
- ни одна система не предоставляет реализации высокоуровневого сетевого протокола для работы с виртуальной средой;
- большинство систем не имеют встроенных механизмов управления репликацией данных;
- в большинстве систем отсутствует механизмы регулирования совместного доступа к состоянию виртуальной среды;
- в открытой печати практически нет сведений по вопросу использования временной синхронизации в существующих РСВР;
- почти все системы ориентированы под конкретные области применения и задачи: *отсутствует общий подход и инструментарий для построения РСВР.*

Среди вышеназванных систем наиболее близким аналогом библиотеки TerraNet является HLA (дословно *Высокоуровневая архитектура*). Стандарт HLA был разработан с двумя основными целями: обеспечить взаимодействие между различными видами тренажеров (симуляций) и сделать возможным повторное использование уже разработанных средств моделирования. Благодаря использованию HLA различные компьютерные симуляции могут взаимодействовать друг с другом. Это взаимодействие осуществляется с помощью так называемой *инфраструктуры времени выполнения* (*Run-Time Infrastructure*, RTI).

При всех своих достоинствах стандарт HLA обладает рядом существенных недостатков:

- стандарт определяет лишь интерфейс и набор правил построения RTI, но не его реализацию, в результате, существующие реализации (например, MÄK RTI [11], Portico, pRTI) имеют различное быстродействие и функциональность, что может приводить к их неправильному или неполноценному взаимодействию;
- HLA поддерживает только одноранговую схему взаимодействия процессов РСВР, которая обладает плохой масштабируемостью, и может эффективно использоваться только в масштабах локальных сетей;
- в HLA RTI на уровне стандарта отсутствуют какие-либо встроенные механизмы по преодолению сетевых ограничений и оптимизации объемов передаваемых данных [12];
- стандарт HLA является слишком сложным и требует значительных временных затрат для изучения и дальнейшего использования.

Из анализа существующих решений можно сделать вывод, что в настоящее время многие вопросы построения РСВР все еще остаются открытыми и требуют дальнейших исследований. И главным из них является вопрос обеспечения согласованности данных в РСВР при одновременном поддержании заданной чувствительности системы.

2. Задача обеспечения компромисса между согласованностью и чувствительностью

Основной целью, к которой необходимо стремиться при построении любой РСВР, является обеспечение баланса двух основных показателей качества таких систем — *согласованности данных* между процессами системы и *чувствительности* системы к действиям пользователей.

Под согласованностью данных в РСВР [13, 14] понимается поддержание идентичных копий глобального состояния виртуальной среды на множестве процессов, входящих в ее состав, на протяжении времени работы системы. Именно согласованность позволяет добиться эффекта одновременного присутствия многих пользователей в общей виртуальной среде.

Как известно из теории классических распределенных систем, основным способом поддержания согласованности данных является *репликация* [15] — механизм синхронизации содержимого нескольких копий объекта (например, содержимого базы данных), под которым понимается копирование данных из одного источника на множество других и наоборот. Скорость, с которой осуществляется репликация, напрямую влияет на согласованность данных в системе.

Чувствительность системы характеризуется временем, необходимым для того, чтобы действие, осуществленное пользователем, произвело результат в виртуальной среде. Говоря кратко, чувствительность есть время отклика системы на действие пользователя.

В общем случае эти два требования противоречат друг другу: любое действие, проведенное каким-либо процессом РСВР, выводит систему из согласованного состояния до тех пор, пока информация об этом действии не будет передана другим процессам, что может занимать значительное время, превышающее максимально допустимое время отклика. Поэтому в РСВР необходимо искать компромисс между согласованностью и чувствительностью. Рассмотрим формальную постановку задачи обеспечения компромисса между согласованностью и чувствительностью.

Определение 1. *Виртуальная среда VE* есть совокупность виртуальных объектов:

$$VE = \{O_1,\ O_2,\ ...,\ O_m\}.$$

В рамках виртуальной среды небосвод, ландшафт, деревья, летательные аппараты, наземные сооружения — все это объекты, либо их совокупности.

Определение 2. *Глобальное состояние виртуальной среды* S_{VE} есть совокупность состояний виртуальных объектов с заданными наборами атрибутов:

$$S_{VE} = \left\{ S_{O_1}, S_{O_2}, ..., S_{O_m} \right\},$$

$$S_{O_j} = \left\{ a_1^{O_j}, a_2^{O_j},\ ..., a_{n_j}^{O_j} \right\}, j = \{1,\ ...,\ m\},$$

где $S_{O_1}, S_{O_2}, ..., S_{O_m}$ — состояния объектов $O_1,\ O_2,\ ...,\ O_m$ соответственно, $a_1^{O_j}, a_2^{O_j},\ ..., a_{n_j}^{O_j}$ — атрибуты, n_j — число атрибутов объекта O_j. В зависимости от сложности виртуальной среды процессы РСВР могут хранить как полное состояние виртуальной среды S_{VE}, так и его k-ую часть $S_{VE}^k \subset S_{VE}$.

Рассмотрим теперь, как осуществляется хранение состояния виртуальной среды в процессах РСВР. Для этого введем понятие *реплики состояния объекта.*

Определение 3. *Реплика* $R_{O_k}^i$ *состояния объекта* O_k есть копия его состояния $S_{O_k}^i$, хранимая в процессе p_i.

Будем говорить, что процесс *реплицирует* объект, если реплика данного объекта существует в локальной памяти процесса. При этом реплика называется *активной*, если процесс, на котором она хранится, транслирует изменения в состоянии соответствующего объекта другим процессам. В противном случае, реплика называется *пассивной.*

Итак, необходимо обеспечить требуемую согласованность данных при ограничениях на сетевые ресурсы и заданной чувствительности. Для этого в каждом цикле моделирования глобального состояния виртуальной среды должно выполняться условие:

$$T_R = \frac{1}{B}\left[\sum_{\forall S_{O_k} \in S_{VE}} \left(\sum_{\forall a_l \in \Delta S_{O_k}} size(a_l) \right) \right] \to \min, \quad (1)$$

$$\text{при ограничениях} \begin{cases} T_R \leqslant T_S, \\ B \leqslant B_{\max}, \end{cases}$$

где $size(a_l)$ — размер атрибута a_l, байт; ΔS_{O_k} — измененная в процессе моделирования часть состояния объекта O_k; B — пропускная способность вычислительной сети, байт/с; $B_{\max}$ — максимальная пропускная способность вычислительной сети, байт/с; T_R — время установления согласованного состояния в системе, мс; T_S — чувствительность системы, мс.

Условие (1) гласит, что для обеспечения требуемой чувствительности системы при сохранении согласованности данных, суммарный трафик, передаваемый в РСВР за цикл моделирования, не должен превышать доступную пропускную способность сети. В процессе разработки ПО TerraNet решалась задача обеспечения согласованности данных при сохранении требуемого уровня чувствительности, который принимается равным 1/60 секунды, что является современным стандартом, предъявляемым к интерактивным приложениям трехмерной графики.

3. Архитектура программного комплекса TerraNet

На программном уровне будем рассматривать РСВР как множество самостоятельных процессов $P = \{p_1,\ p_2,\ ...,\ p_n\}$, взаимодействующих между собой по сети [16]. Без потери общности положим, что каждый процесс выполняется на отдельном вычислительном узле. Процессы не имеют общей памяти и взаимодействуют друг с другом исключительно на основе передачи сообщений по определенному *высокоуровневому протоколу.* В системе отсутствуют общие глобальные часы, к которым можно получить мгно-

венный доступ. Выполнение процессов и передача сообщений являются асинхронными: процессы могут осуществлять действия в произвольные моменты времени, а посылка сообщений не блокирует процесс вычислений.

Взаимодействие процессов, как правило, осуществляется на основе определенного типа *схемы взаимодействия процессов*, определяющей способ взаимного соединения процессов и роли процессов в процессе обмена. Основными способами взаимодействия процессов являются одноранговая (*peer-to-peer*) и клиент-серверная (*client/server*) схемы (рис. 1, см. вторую сторону обложки). Соответственно, в зависимости от выбранной схемы можно выделить клиентские (*client*), серверные (*server*) и одноранговые (*peer*) процессы.

В одноранговой схеме взаимодействия (рис. 1, *а*) все процессы (на рис. обозначены как *C*) являются равноправными и могут передавать свои данные любым другим процессам сети. Ее достоинством является простота реализации и высокая скорость доставки данных по сети. Однако с ростом числа процессов, число связей между ними быстро увеличивается, начинает лавинообразно нарастать трафик и сеть быстро становится узким местом, препятствующим дальнейшему росту числа пользователей системы. Поэтому такую схему в чистом виде используют, когда число процессов в системе невелико, чаще всего в локальных сетях. Общее число процессов в одноранговой схеме обычно не превышает 30. Такая схема широко применялась в ранних РСВР, однако, и поныне она пользуется популярностью, особенно в контексте использования технологии множественной рассылки (*multicasting*).

В клиент-серверной схеме взаимодействия (рис. 1, *б*) выделяется один главный процесс (*S*), называемый сервером, через который осуществляются коммуникации между остальными процессами системы, называемыми клиентами. Достоинство данной схемы состоит в том, что в ней каждый клиент обращается к другим клиентам через сервер и соединен только с ним, т. е. общее число связей в такой сети меньше по сравнению с одноранговой схемой. Также становится возможным централизованный контроль и мониторинг состояния клиентов. Причем клиенты могут располагаться как в пределах локальной сети, так и быть разбросанными в пределах сети Интернет. Основной недостаток — с ростом числа клиентов единственный сервер быстро становится узким местом системы и начинает плохо справляться со своими основными задачами — моделированием состояния виртуальной среды и обработкой запросов пользователей. По этой причине клиент-серверная схема, в основном, применяется при построении РСВР среднего масштаба (до 100 пользователей).

Для дальнейшего увеличения масштаба системы применяют *мультисерверную схему* взаимодействия (рис. 1, *в*). Мультисерверная схема позволяет распределить обработку большого числа клиентов между несколькими серверами. Данное распределение может осуществляться различными способами, хороший обзор можно найти, например, в работе [8].

Текущая версия TerraNet ориентирована преимущественно на РСВР среднего масштаба, поэтому за основу была взята именно клиент-серверная схема взаимодействия. Однако отличительной особенностью используемой схемы является возможность применения многоадресной рассылки данных помимо стандартной централизованной посылки данных через сервер (рис. 1, *г*). Таким образом, наряду с взаимодействием через сервер возможно прямое взаимодействие клиентов. Многоадресная рассылка является более выигрышной по сравнению с широковещательной рассылкой, поскольку позволяет за одно обращение к сетевой подсистеме рассылать данные только тем клиентам, кто заинтересован в их получении. При этом у клиентов появляется возможность *подписываться* на необходимые им данные. В результате можно снизить нагрузку на сервер и сократить время доставки критически важных данных от одних клиентов к другим.

Программная архитектура библиотеки TerraNet представлена на рис. 2 (см. вторую сторону обложки), где изображены основные, на наш взгляд, блоки, из которых должна состоять реальная РСВР. Согласно предлагаемой структурной схеме, РСВР рассматривается как множество процессов, взаимодействующих друг с другом на основе определенной схемы взаимодействия. Каждый процесс обеспечивает функционирование отдельного компонента РСВР и построен на основе ядра библиотеки *TerraNet core*.

Каждый компонент РСВР, в зависимости от конкретной задачи, может быть *пассивным* либо *активным*. Если пассивные компоненты могут быть только наблюдателями текущего состояния виртуальной среды, то активные компоненты способны модифицировать состояние виртуальной среды путем создания новых или изменения состояния уже существующих объектов. Так, пример пассивного компонента в современных авиационных тренажерах — система визуализации, основная задача которой отображение внекабинного пространства летательного аппарата. Примером активного компонента является модуль расчета физической модели самолета.

В соответствии с клиент-серверной схемой взаимодействия в архитектуре TerraNet можно выделить ведущий и ведомый компоненты. Ведущий компонент располагается на сервере и применяется для координации работы и обеспечения взаимодействия ведомых компонентов, расположенных на клиентах, а также для контроля и мониторинга состояния всей системы.

Компонент РСВР включает три основных блока. *Блок обработки данных с устройств ввода* является опциональным и может присутствовать только на активных компонентах. Он считывает данные с устройств ввода пользователя и преобразует их в управляющие воздействия, поступающие в *блок моделирования локального состояния виртуальной среды*, который проводит обработку и моделирование текущего состоя-

ния виртуальной среды в соответствии с логикой работы компонента. Блок визуализации также является опциональным и может входить в состав как активных, так и пассивных компонентов. Он предназначен для визуализации текущего состояния виртуальной среды.

Локальное хранилище данных хранит копию той части состояния виртуальной среды, которая необходима для построения связанного с компонентом вида виртуальной среды. Для поддержания непротиворечивости данных присутствует *блок контроля доступа к состоянию виртуальной среды*, который позволяет разграничивать доступ к состоянию виртуальной среды между процессами. Предполагается, что в каждый момент времени только один процесс имеет право модифицировать состояние виртуальной среды.

Глобальное хранилище состояния виртуальной среды, расположенное на сервере, хранит полную копию состояния виртуальной среды, считающуюся эталонной во всей системе. Все изменения, которые процессы пользователей проводят в локальных копиях состояния виртуальной среды, передаются и фиксируются в глобальном хранилище. В случае необходимости данные, отсутствующие в локальных хранилищах процессов, загружаются из глобального хранилища. Так, например, происходит в моменты подключения новых клиентов и при создании новых объектов. Таким образом, глобальное хранилище является аккумулятором всех изменений состояния виртуальной среды, проводимых пользователями. Между локальным и глобальным хранилищем происходит постоянный обмен данными в обе стороны. Вместе глобальное хранилище сервера и локальные хранилища клиентов образуют распределенное хранилище данных.

Для наших целей удобно организовать данные в хранилище в виде иерархической структуры, разделяемой между процессами — *распределенного графа сцены*, который представляет собой направленный ациклический граф $G = (V, E)$, где V — множество вершин-узлов графа сцены, соответствующее множеству объектов виртуальной среды, а E — множество дуг, устанавливающих логические и пространственные взаимоотношения между ними.

Наиболее важным в рассматриваемой архитектуре является *блок обеспечения согласованности данных*. Он является составным и решает следующие задачи:

- обеспечение межпроцессного взаимодействия (с использованием *высокоуровневого протокола и интерфейса межпроцесснорного взаимодействия*);
- подключение и обслуживание новых клиентов (*менеджер клиентов*);
- компенсация аппаратных ограничений сети (*блок предсказаний и интерполяции состояний объектов*);
- управление репликацией данных (*блок управления репликацией данных и блок управления подпиской на данные*);
- временная синхронизация процессов PCBP;
- управление совместным доступом к состоянию виртуальной среды (*блок контроля доступа к состоянию виртуальной среды*).

4. Высокоуровневый протокол межпроцессного взаимодействия

Взаимодействие между процессами внутри программного комплекса TerraNet основано на специально разработанном высокоуровневом проколе DVRP (*Distibuted Virtual Reality Protocol*). Он опирается на два протокола транспортного уровня стека TCP/IP: TCP и UDP, сочетая в себе надежность первого и скорость второго. По протоколу TCP передаются одиночные сообщения, требующие надежной доставки (например, данные о подключении/отключении удаленных пользователей), осуществляется синхронизация пользователей, а также проводится контроль соединения клиентов с сервером. UDP используется для часто передаваемых сообщений (составляющих основной поток данных), для которых важна скорость доставки, а потеря либо некритична, либо легко восполнима. К таким сообщениям относятся обновления состояний объектов виртуальной среды. Такое разделение сообщений по транспортным протоколам позволяет более рационально использовать пропускную способность сети, уменьшает латентность при пересылке данных и при этом позволяет выполнять требования по надежности доставки критически важных данных.

4.1. Состав протокольных сообщений

Для удобства обработки и дальнейшего расширения протокола сообщения разделены по группам:

- *мировые сообщения* (*world messages*) — глобальная модификация состояния виртуальной среды;
- *объектовые сообщения* (*object messages*) — манипуляции с отдельными объектами: создание/удаление объектов, модификация распределенного графа сцены, управление доступом к состоянию объектов;
- *сообщения камеры* (*camera messages*) — манипуляция с виртуальной камерой пользователя — информирование удаленных пользователей о положении и настройках вида текущего пользователя;
- *сообщения для работы с атрибутами объектов* (*attribute messages*) — создание/удаление/модификация и другие операции с отдельными атрибутами состояний объектов;
- *служебные сообщения* (*service messages*) — реализация служебных функций: адресной посылки, групповой посылки, механизма подписки, сбора сетевой статистики, механизма удаленного вызова процедур (*Remote Procedure Call, RPC*), передачи файлов по запросу и т. д.

Одной из отличительных особенностей предлагаемого протокола, закладываемой при его проектировании, является возможность использования сообщений переменной длины. Это достигается за счет отказа от заранее предопределенных структур протокольных сообщений (рис. 3, *а*). Единственными постоянными полями сообщений являются заголовок сообщения, состоящий из части, идентифицирующей принадлежность сообщения к той или иной группе (*SuperID*), и части, задающей позицию данного сообщения внутри группы (*ID*).

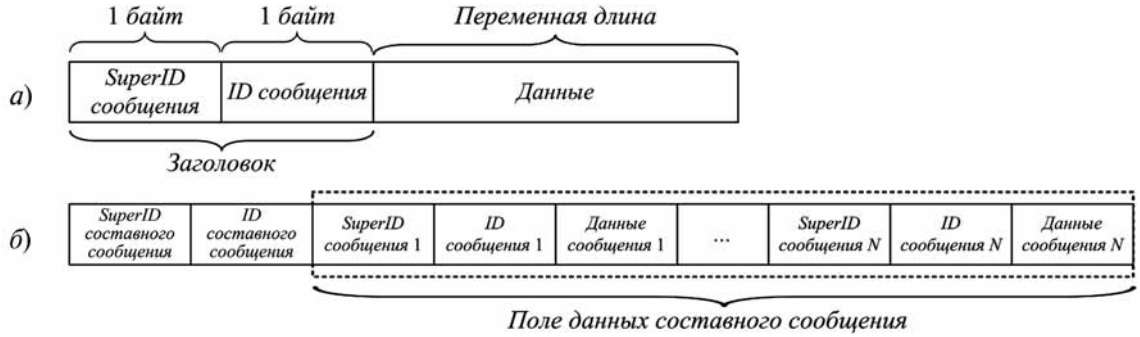

Рис. 3. Форматы сообщений протокола DVRP

Каждое сообщение представляет собой *битовый поток* (*BitStream*), который компонуется налету. Вначале в него записывается заголовок сообщения, а затем, последовательно, поля, специфические для конкретного типа сообщения. Причем имеется возможность побитового формирования сообщения, что может быть использовано для дополнительной экономии пропускной способности сети. Например, при записи логических переменных в сообщение традиционным способом каждая такая переменная представляется типом *bool*, который в языке C++ занимает в памяти 4 байта. При использовании битового потока булева переменная может быть представлена всего 1 битом.

Другим достоинством использования битовых потоков является возможность группирования нескольких сообщений в одно (рис. 3, *б*). В протоколе для этого вводится специальный *пакетный* тип сообщения. После записи его заголовка в поток последовательно записываются составные сообщения.

Обработка протокольных сообщений на приемной стороне проводится в том же порядке, что и компоновка при отправке. Вначале из битового потока зачитывается заголовок, затем по нему находится соответствующий обработчик, и ему на вход подается неразобранный остаток сообщения. Если требуемый обработчик отсутствует, сообщение отбрасывается. Единственным условием, предъявляемым к обработчикам, является строгое соблюдение порядка следования и размера считываемых полей (они должны быть такими же, как при отправке).

Использование битовых потоков позволяет писать универсальные обработчики, способные изменять логику обработки в зависимости от значений зачитываемых полей сообщений. Например, одно и то же сообщение может использоваться как для положительных, так и отрицательных ответов на какой-либо запрос. Причем в случае положительного ответа сообщение-ответ может включать дополнительную информацию об объекте запроса, а в случае отрицательного ответа — информацию о причинах отклонения запроса.

Благодаря использованию битовых потоков протокол DVRP при необходимости может быть легко модифицирован. Добавление новых сообщений не требует введения в программу новых структур данных: каждое сообщение компонуется по месту своего возникновения. Необходимо лишь зарегистрировать в системе обработчик для нового типа сообщения.

Битовый поток также представляет собой удобное средство для *сериализации* объектов виртуальной среды — преобразования объектов из внутреннего представления в последовательную форму, пригодную для передачи по сети. В каждом цикле моделирования модифицированные атрибуты состояний объектов сериализуются один за другим в единый битовый поток и высылаются в сеть за один вызов *Send()* (см. далее).

4.2. Типы взаимодействий процессов

В протоколе DVRP существуют два основных типа взаимодействия процессов: *взаимодействие клиент-сервер* и *взаимодействие клиент-клиент*.

Взаимодействие клиент-сервер является основным и соответствует выбранной клиент-серверной схеме взаимодействия процессов. Основные его фазы показаны на рис. 4.

Фаза соединения. Фаза описывает подключение нового клиента к системе. При запуске клиент устанавливает три соединения с сервером: основное TCP-соединение, RPC TCP-соединение и UDP-соединение (типы взаимодействий *1*, *2*, *4*). По основному TCP-соединению проводится обмен данными, требующими надежной доставки. RPC TCP-соединение используется для блокирующих RPC-запросов. Соединение поверх UDP-протокола, как было сказано, используется для доставки данных, не чувствительных к потерям. Поскольку протокол UDP не поддерживает установления соединений, в разрабатываемом протоколе организуется дополнительный логический уровень-надстройка над UDP, предоставляющий данную возможность.

RPC-запрос в разрабатываемом протоколе имеет схожий с классическим RPC [15] алгоритм работы, позволяет вызывать функции на удаленных процессах PCBP и получать результат их выполнения. При этом происходит блокировка вызывающего процесса до момента получения ответа от удаленного процесса. Механизм битовых потоков позволяет помещать любое число аргументов функции в RPC-запрос, а также

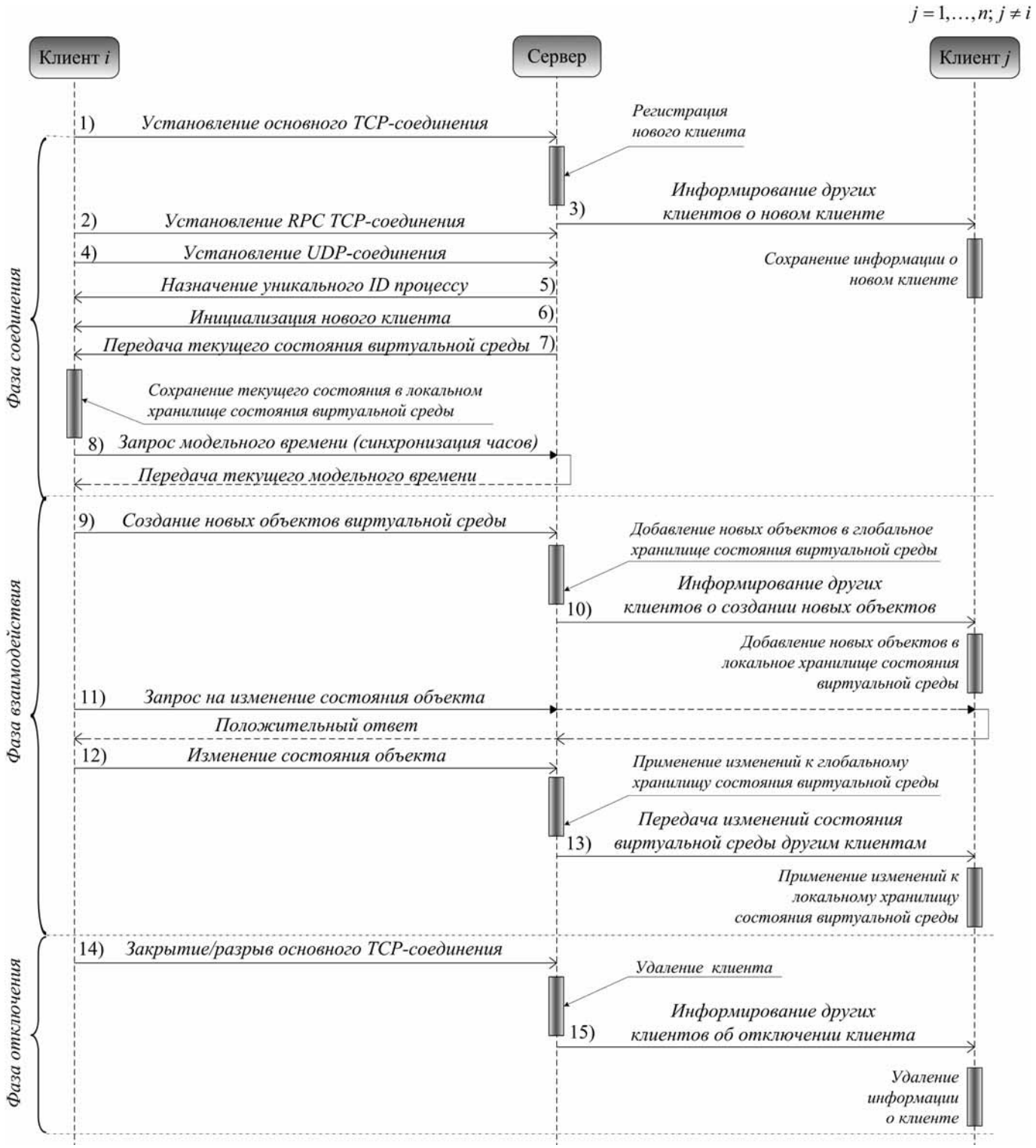

Рис. 4. Фазы взаимодействия между клиентом и сервером по протоколу DVRP

возвращать любую необходимую информацию во время RPC-ответа. Для того чтобы RPC-запрос был корректно обработан, соответствующая функция-обработчик должна быть зарегистрирована на удаленной системе.

После того, как все три соединения успешно установлены, сервер назначает клиенту уникальный идентификатор, под которым в дальнейшем данный клиент будет доступен в системе (тип взаимодействия *5*). Затем сервер проводит инициализацию клиента, пересылая ему служебную информацию, включающую информацию об уже подключенных клиентах (тип взаимодействия *6*) и данные о текущем состоянии виртуальной среды (тип взаимодействия *7*). По завершении инициализации клиент проводит синхронизацию своих часов с часами сервера (тип взаимодействия *8*). С этого момента он считается подключенным к распределенной системе.

Фаза взаимодействия. Фаза описывает возможные варианты взаимодействия клиентского процесса с сервером после успешного установления соединения. Во время фазы взаимодействия клиент может осуществлять множество действий, основные из них: создание новых объектов виртуальной среды (тип взаимодействия *9*) и изменение состояния существующих объектов (типы взаимодействия *11*, *12*). Частота отправки изменений (репликации) может задаваться как фиксированной, так и переменной, в зависимости от выбранной стратегии репликации (см. часть II статьи). Сервер информирует остальных клиентов о модификации состояния виртуальной среды (типы взаимодействия *10*, *13*). Изменения в состояниях объектов могут распространяться как по протоколу UDP, так и по TCP, в зависимости от выбранного для атрибута транспортного протокола.

Фаза отключения. Клиент считается отключенным, если он закрывает основное TCP-соединение, либо происходит неожиданный разрыв соединения, например, вызванный сбоем в сети (тип взаимодействия *14*). В любом случае сервер надежно контролирует подключение по протоколу TCP и проводит корректную процедуру отключения, оповестив других клиентов (тип взаимодействия *15*).

Взаимодействие клиент-клиент может осуществляться как через сервер, так и напрямую — при использовании множественной рассылки. Первый тип посылки взаимодействия между клиентами используется при посылке данных, требующих надежной доставки (по протоколу TCP). Для этого в протоколе предусмотрено специальное сообщение: `NET_MSG_SERVICE_ADDRESS_SEND`. Получив его, сервер перенаправляет сообщение указанному в сообщении клиенту. В результате, для клиента, отправляющего сообщение, все выглядит прозрачно: он посылает сообщение напрямую нужному клиенту.

Второй тип взаимодействия клиент-клиент подразумевает посылку клиентом данных на определенные multicast-группы, на которые подписаны другие клиенты. В данном случае за доставку данных отвечает транспортный протокол. При этом для пользователя библиотеки непосредственная работа с multicast-группами скрыта. Эти группы автоматически создаются при создании объектов виртуальной среды, шаблонов состояний и взаимодействий.

Во второй части статьи мы продолжим знакомство с программным комплексом TerraNet. Будут рассмотрены механизмы обеспечения согласованности данных в предлагаемой программной архитектуре, а также основные возможности и детали реализации программного комплекса TerraNet.

Список литературы

1. **Miranda B.** et al. A taxonomy for the design and evaluation of Networked Virtual Environments: its application to collaborative design // International Journal on Interactive Design and Manufacturing, 2008. Vol. 2. No. 1. P. 17—32.

2. **IEEE** Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) — Framework and Rules. IEEE Standard 1516-2000 (2000).

3. **Qin J.** et al. A framework using cluster-based hybrid network architecture for collaborative virtual surgery // Computer Methods and Programs in Biomedicine, 2009. Vol. 96. Is. 3. P. 205—216.

4. **Capps M.** et al. NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments // IEEE Computer Graphics & Applications. 2000. Vol. 20. P. 12—15.

5. **Greenhalgh C., Benford S**. MASSIVE: a collaborative virtual environment for teleconferencing // ACM Transactions on Computer-Human Interaction. 1995. Vol. 2. Is. 3. P. 239—261.

6. **Tramberend H.** Avocado: A distributed virtual reality framework // In Proceedings of the IEEE Virtual Reality Conference, 1999. P. 14—21.

7. **Source Multiplayer Networking.** Valve Software [Electronic resource]. Electronic data. URL: http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

8. **Glinka F.** et al. High-level development of multiserver online games // Int. J. Comput. Games Thechnol. 2008. Jan. P. 1—16.

9. **Müller J., Gorlatch S.** Enhancing Online Computer Games for Grids // 9th International Conference on Parallel Computing Technologies (PaCT 2007). Lecture Notes in Computer Science. Russian Federation, Pereslavl-Zalessky: Springer Verlag, September 2007. Vol. 4671. P. 80—95.

10. **Харитонов В. Ю.** Сетевые механизмы обеспечения согласованности данных в распределенных системах виртуальной реальности. Автореферат диссертации на соискание ученой степени кандидата технических наук. М.: Полиграфический центр МЭИ (ТУ), 2010. 20 с.

11. **MÄK** High Performance RTI [Electronic resource]. Electronic data. URL: http://www.mak.com/products/rti.php.

12. **Strassburger S., Schulze T., Fujimoto R.** Future trends in distributed simulation and distributed virtual environments: results of a peer study // In Proceedings of the 40th Conference on Winter Simulation. Miami, Florida, USA. 2008. P. 777—785.

13. **Харитонов В. Ю.** Модели согласованности для распределенных систем виртуальной реальности // Труды третьей всероссийской научно-технической конференции "Методы и средства обработки информации", Москва, 6—8 октября 2009 г., МГУ им. М. В. Ломоносова / Под ред. Л. Н. Королева. М.: МАКС Пресс. 2009. С. 64—70.

14. **Kharitonov V. Y.** A Consistency Model for Distributed Virtual Reality Systems// Proc. of 4th International Conference on Dependability of Computer Systems DepCoS-RELCOMEX 2009. IEEE Computer Society. Los Alamitos. CA. USA, 2009. P. 271—278.

15. **Таненбаум Э., Стен М.** Распределенные системы: принципы и парадигмы. СПб.: Питер, 2003. 877 с.

16. **Харитонов В. Ю. Дзегеленок И. И., Орлов Д. А.** Вычислительные аспекты построения распределенных систем виртуальной реальности // Вестник Московского Энергетического института. 2008. № 5. С. 27—32.

УДК 519.685

А. Б. Фролов, д-р техн. наук, проф., **А. М. Винников**, студент,
Московский энергетический институт (технический университет),
e-mail: abfrolov@mail.ru

О машинном синтезе некоторых линейных программ*

Рассматриваются два способа автоматического синтеза программ умножения многоразрядных чисел по методу Карацубы, основанные на использовании рекурсивных описаний этого метода. Описан аналогичный подход к синтезу программ умножения многочленов высокой степени над конечным полем. Приводится метод сравнения получаемых программ для доказательства корректности программ их синтеза и оценивается их сложность.

Ключевые слова: *автоматизация программирования, машинный синтез программ, линейная программа, умножение многоразрядных чисел, умножение многочленов над конечным полем, декомпозиционная схема, метод Карацубы, рекурсия в глубину, рекурсия в ширину, корректность программ, оценки сложности*

Введение

Изучение методов автоматического синтеза программ является важным разделом программной инженерии. Основной особенностью автоматически синтезированной компьютерной программы является ее надежность и отсутствие необходимости ее тестирования при конкретных значениях входных данных. Под автоматизацией синтеза программ будем понимать создание программного средства для формирования кода программы, реализующей алгоритмы заданного класса, по значениям глобальных параметров этих алгоритмов. Применением такого программного средства и осуществляется автоматический синтез программы. Алгоритмы, допускающие автоматический синтез программ, можно найти, в частности, среди алгоритмов компьютерной алгебры [1, 2], описывающих операции в конечных алгебраических структурах и некоторые алгебраические преобразования. Так, умножение многоразрядных чисел, умножение многочленов высокой степени над конечным полем, быстрое преобразование Фурье и преобразование Уолша—Адамара используются во многих алгоритмах цифровой обработки сигналов, кодирования и криптографии. При этом получаются последовательные или линейные программы, не содержащие циклов и условных переходов.

Впервые задача оптимизации умножения рассматривалась в работе [3]. Этой задаче посвящены работы [1, 4]. В частности, в работе [4] обоснована так называемая декомпозиционная схема умножения многочленов над конечным полем, допускающая схемную и программную реализацию. Аналогичная схема может быть составлена для умножения многоразрядных целых чисел. В работах [5, 6], выполненных под руководством одного их авторов, сообщается о возможности автоматизации синтеза подобных схем и соответствующих компьютерных программ. Аргументы о корректности получаемых алгоритмов и программ, а также вопросы оптимизации распределения памяти для сокращения времени пересылок применительно к алгоритмам умножения были рассмотрены в работе [7]. В работе [2] дано более полное их освещение. В настоящей работе эти идеи излагаются применительно к синтезу программ умножения многоразрядных целых чисел с возможной модификацией применительно к умножению многочленов над конечным полем.

1. Умножение многоразрядных чисел по методу Карацубы

Как известно, целое число A удобно представлять в позиционной системе счисления по некоторому основанию p в виде цепочки $(a_{n-1}a_{n-2}...a_1a_0)_p$, совокупность элементов которой образует вектор

$$(a_0, a_1, ..., a_{n-1}), \qquad (1)$$

где элементы таких цепочек и векторов это числа a_i, $0 \leqslant a_i \leqslant p - 1$, или цифры позиционной системы счисления. Число n называется разрядностью представления числа в данной системе счисления.

* Работа подготовлена при финансовой поддержке РФФИ, проект № 11-01-00792-а.

Так числу 126 в десятичной системе счисления соответствуют цепочка $(126)_{10}$ и вектор (6,2,1),

в двоичной системе счисления — цепочка $(111110)_2$ и вектор (0, 1, 1, 1, 1, 1),

в восьмеричной системе счисления — цепочка $(176)_8$ и вектор (6, 7, 1).

По вектору (1) число A определяется как сумма

$$A = \sum_{i=0}^{n-1} a_i p_i. \qquad (2)$$

Например,

$$126 = 6 \times 10^0 + 2 \times 10^1 + 1 \times 10^2 = 6 + 20 + 100$$

или

$$126 = 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + \\ + 1 \times 2^5 + 1 \times 2^6 = 0 + 2 + 4 + 8 + 16 + 32 + 64.$$

Если $n = 2m$ (m — натуральное число, т. е. n четно), то сумма (2) разбивается на две части:

$$A = \sum_{i=0}^{n-1} a_i p^i = \sum_{i=0}^{m-1} a_i p^i + \sum_{m}^{n-1} a_i p^i.$$

Вынесем за скобки p^m во второй сумме и получим

$$A = \sum_{i=0}^{n-1} a_i p^i = \sum_{i=0}^{m-1} a_i p^i + p^m \sum_{i=0}^{m-1} a_{i+m} p^i = A_0 + p^m A_1,$$

где числа A_0 и A_1 имеют разрядность не более m, а разрядность числа A не более $2m$. Это ни что иное, как двухразрядное представление числа A в системе счисления по основанию p^m.

Пусть надо перемножить числа A и B, разрядность которых не превышает $2m$. Представим каждое из них в этой системе счисления по основанию p^m

$$A = A_0 + p^m A_1,$$

$$B = B_0 + p^m B_1.$$

Тогда по "школьному алгоритму" (или алгоритму сдвигов и сложений) мы можем получить произведение по формуле

$$A \times B = A_0 \times B_0 + \\ + (A_0 \times B_1 + A_1 \times B_0) p^m + A_1 \times B_1 p^{2m}. \qquad (3)$$

При этом надо выполнить четыре операции умножения чисел вдвое меньшей разрядности (одноразрядных чисел по основанию p^m).

По формуле Карацубы [3] произведение тех же чисел получается следующим образом

$$A \times B = A_0 \times B_0 (1 + p^m) + (A_1 - A_0)(B_0 - B_1)(p^m) + \\ + A_1 \times B_1 (p^m + p^{2m}). \qquad (4)$$

При этом надо выполнить три операции умножения чисел вдвое меньшей разрядности. Нетрудно проверить, что формулы (3) и (4) эквивалентны.

Для примера рассмотрим умножение по формуле Карацубы чисел $A = 1234$ и $B = 2314$ ($n = 4$, $m = 2$) в десятичной системе счисления, полагая, что $A_0 = 34$, $A_1 = 12$, $B_0 = 14$, $B_1 = 23$:

$$34 \times 14(1 + 10^2) + (12 - 34)(14 - 23)10^2 + 12 \times \\ \times 23 \times (10^2 + 10^4) = 476 \times 101 + 198 \times \\ \times 100 + 276 \times 10\,100 = \\ = 48\,076 + 19\,800 + 2\,787\,600 = 2\,855\,476.$$

В данном случае десятичная система счисления использовалась для большей наглядности. Далее будем полагать, что числа представляются в двоичной системе и при размещении в памяти занимают $n = 2^k$ ($k \geqslant 1$) машинных единиц (бит, байт или машинных слов)[1]. Используем здесь число n в виде степени двойки, чтобы была возможность применения формулы (4) при перемножении как самих чисел, так и их составных частей, образуемых при возможно многократном дроблении на две части. Тогда сомножители всегда рассматриваются как двухразрядные числа по основанию 2^{k-1}. При этом будем считать, что перемножение элементарных чисел осуществляется по известной подпрограмме (программной функции). Элементарные числа это одноразрядные числа по основанию 2^s. Например, если используется программная функция перемножения байтов, то $s = 8$, если это функция перемножения машинных слов, то $s = 32$. Нашей задачей является получение программы умножения многоразрядных чисел по основанию 2^s по методу Карацубы как последовательности действий с такими числами, содержащие операции умножения только элементарных чисел.

2. Две линейные программы умножения многоразрядных чисел

Будем обозначать $[a_A]^n$ n-разрядное число A, размещенное, начиная с адреса a_A, $n = 2^k$, $k \geqslant 1$. Рассматриваемые ниже программы вычисления произведения $[a_{AB}]^{2n}$ представляют собой линейные последовательности строк вида $op(P, Q, N)$ или $op(P, N)$, где op — обозначение операции, P относительный адрес первого операнда и результата, Q относительный адрес второго операнда, N разрядность по основанию 2^s операндов [2, 7]:

`equ(P,Q,N)` — операция присваивания: $[P]^N := [Q]^N$;

`sub(P,Q,N)` — операция вычитания: $[P]^N := [P]^N - [Q]^N$;

`add(P,Q,N)` — операция сложения: $[P]^{N+1} := [P]^N + [Q]^N$;

`mul(P,Q,N)` — операция умножения: $[P] = [P] \cdot [Q]$.

Для краткости, последовательно выполняемые операции `equ(-1,Q,N)`, `equ(Q,P,N)`, `equ(P,-1,N)`, где `-1` есть относительный адрес для временного сохранения операнда, обозначаются `xchg(P, Q, N)`,

[1] Мы выбрали двоичную систему счисления для упрощения изложения, на самом деле можно использовать позиционные системы счисления по любому основанию.

как операция перемещения (элементы с адресами P и Q меняются местами); последовательно выполняемые операции `equ(T,P,N)`, `sub(T,Q,N)` обозначаются `eqsub(T,P,Q,N)`; последовательно выполняемые операции `equ(T,P,N)`, `mul(T,Q,N)` обозначаются `eqmul(T,P,Q,N)`.

В обозначениях операций и синтезируемых программ относительные адреса задаются числами элементарных слов. Указатель на базисный адрес, как и индекс s разрядности 2^s элементарных чисел в силу их единственности считаются известными по умолчанию и явно не указываются. В обозначении `mul(T,Q,1)` операции умножения элементарных чисел имя используемой программной функции также считается известным по умолчанию. Полные обозначения синтезируемых программ и операций даны в разд. 3.

Программа для вычисления в системе счисления по основанию 2^s произведения $A \times B$ двухразрядных чисел $A = A_1A_0$ и $B = B_1B_0$ по формуле (4) может быть представлена в следующем виде.

Программа 1 (multiplicationdepth(1))[2]: множители (по два слова) и произведение (четыре слова) размещены начиная с относительных адресов 0, 2 и 6, две разности — по адресам 4 и 5, произведение разностей размещается начиная с относительного адреса 10, по этому адресу накапливается сумма трех слагаемых с возможным переполнением, размещаемым по адресу 12, в связи с этим требуется предварительная очистка:

```
eqsub(8,0,1,1)
eqsub(9,3,2,1)
egmul(4,0,2,1)
eqmul(6,1,3,1)
eqmul(10,8,9,1)
clean(12,1)
add(10,6,2)
add(10,4,2)
add(5,10,3).
```

Над программами рассматриваемого вида можно выполнять операции *удвоения*, удваивая все встречающееся в программе адреса и длины операндов (длина a операндов последней команды заменяется на $(2(a - 1) + 1 = 2a - 1)$). Например, удвоением программы 1 получим следующую программу.

Программа 2 (удвоение программы multiplicationdepth(1)):

```
eqsub(16,0,2,2)
eqsub(18,6,4,2)
eqmul(8,0,4,2)
eqmul(12,2,6,2)
eqmul(20,16,18,2)
clean(24,2)
add(20,12,4)
add(20,8,4)
add(10,20,5)
```

[2] Программа multiplicationdepth(k) есть программа умножения 2^k — разрядных чисел, размещенных по относительным адресам 0 и 2^k, сохраняющая вычисляемое 2^{k+1} — разрядное произведение по относительному адресу 0.

Линейная программа умножения методом Карацубы multiplicationdepth(k) может быть получена на основе программы 1 путем многократного выполнения двух преобразований — удвоения, описанного выше, и *рекурсивной детализации в глубину*.

Пусть построена программа multiplicationdepth($k - 1$).

Таблица 1

Рекурсивная детализация в глубину программы 2

Программа 2	multiplicationdepth(2)
`eqsub (16, 0, 2, 2)`	`eqsub (16, 0, 2, 2)`
`eqsub (18, 6, 4, 2)`	`eqsub (18, 6, 4, 2)`

`eqmul (8, 0, 4, 2)`	`eqsub (24, 0, 1, 1)`
	`eqsub (25, 5, 4, 1)`
	`eqmul (8, 0, 4, 1)`
	`eqmul (10, 1, 5, 1)`
	`eqmul (26, 24, 25, 1)`
	`clean (28,1)`
	`add (26, 8, 2)`
	`add (26, 10, 2)`
	`add (9, 26, 3)`

`eqmul (12, 2, 6, 2)`	`eqsub (24, 2, 3, 1)`
	`eqsub (25, 7, 6, 1)`
	`eqmul (12, 2, 6, 1)`
	`eqmul (14, 3, 7, 1)`
	`eqmul (26, 24, 25, 1)`
	`clean (28, 1)`
	`add (26, 12, 2)`
	`add (26, 14, 2)`
	`add (13, 26, 3)`

`eqmul (20, 16, 18, 2)`	`eqsub (24, 16, 17, 1)`
	`eqsub (25, 19, 18, 1)`
	`eqmul (20, 16, 18, 1)`
	`eqmul (22, 17, 19, 1)`
	`eqmul (26, 24, 25, 1)`
	`clean (28, 1)`
	`add (26, 20, 2)`
	`add (26, 22, 2)`
	`add (21, 26, 3)`

`clean (24, 2)`	`clean (24, 2)`
`add (20, 12, 4)`	`add (20, 12, 4)`
`add (20, 8, 4)`	`add (20, 8, 4)`
`add (10, 20, 5)`	`add (10, 20, 5)`

Программу multiplicationdepth(k) можно получить удвоением построенной программы и детализацией в глубину строк eqmul(a,b,c,2). При этом каждая такая строка заменяется следующей группой строк

```
eqsub(4*2^k-8,b,b+1,1)
eqsub(4*2^k-7,c+1,c,1)
eqmul(a,b,c,1)
eqmul(a+2,b+1,c+1,1)
eqmul(4*2^k-6,4*2^k-8,4*2^k-7,1)
clean(4*2^k-4,1)
add(4*2^k-6,a,2)
add(4*2^k-6,a+2,2)
add(a+1,4*2^k-6,3).
```

В табл. 1 представлены программа 2 и программа multiplicationdepth(2), полученная ее детализацией в глубину.

Возможен и другой вариант программы умножения двухразрядных чисел.

Программа 1′ (multiplicationwidth(1))[3]: Множители (их два) и произведение размещены начиная с относительных адресов 0, 2 и 0, две разности — по адресам 4 и 5, произведение разностей размещается начиная с относительного адреса 4, по этому адресу накапливается сумма трех слагаемых с возможным переполнением, размещаемым по адресу 6, в связи с этим требуется предварительная очистка.

```
eqsub(4,0,1,1)
eqsub(5,3,2,1)
xchg(1,2,1)
mul(0,1,1)
mul(2,3,1)
mul(4,5,1)
clean(6,1)
add(4,0,2)
add(4,2,2)
add(1,4,3).
```

Нетрудно видеть, что программы 1 и 1′ функционально эквивалентны: они реализуют вычисления по формуле (1), программа 1′ размещает произведение на месте сомножителей, а программа 1 размещает его в другой области памяти, сохраняя исходные данные.

Удвоением программы 1′ получим программу 2′:

```
sub(8,0,2,2)
sub(10,6,4,2)
xchg(2,4,2)
mul(0,2,2)
mul(4,6,2)
mul(8,10,2)
clean(12,2)
add(8,0,4)
add(8,4,4)
add(2,8,5).
```

[3] Программа multiplicationwidth(k) есть программа умножения 2^k — разрядных чисел, размещенных по относительным адресам 0 и 2^k, сохраняющая вычисляемое 2^{k+1} — разрядное произведение по относительному адресу 2^{k+1}.

Удвоенные программы соответствуют применению формулы (1) к сомножителям вдвое большей разрядности.

Рекурсивная программа умножения методом Карацубы multiplicationwidth(k) может быть получена на основе программы 1′ (multiplicationwidth(1)) путем многократного выполнения преобразований удвоения, описанного выше, и *рекурсивной детализации в ширину*.

Пусть построена программа multiplicationwidth($k-1$). Такая программа до и после удвоения имеет три раздела: раздел прямого хода рекурсии, предшествующий разделу умножений и завершающий раздел обратного хода рекурсии. В программе 1′, например, эти разделы образуются группами из первых трех строк, последующих трех строк и завершающих четырех строк соответственно.

Программу multiplicationwidth(k) можно получить удвоением построенной программы multiplicationwidth($k-1$) и детализацией имеющихся в удвоенной программе строк mul(c,c+2,2).

Рекурсивная детализация в ширину заключается в том, что:

а) непосредственно перед разделом умножений для каждой строки

```
mul(c,c+2,2)
```

удвоенной программы умножения с порядковым номером i добавляется очередной уровень прямого хода рекурсии, образуемый несколько измененными строками прямого хода рекурсии программы 1′.

```
eqsub(d+2i,c+1,1)
eqsub(d+2,i + 1,c+2,1)
xchg(c+1,c+2,1);
```

б) i-я строка умножения

```
mul(c,c+2,2)
```

заменяется тройкой строк

```
mul(c,c+1,1)
mul(c+2,c+3,1)
mul(d+2i,d+2i+1,1);
```

в) непосредственно после раздела умножений для каждой удвоенной строки умножения с порядковым номером i

```
mul(c,c+2,2)
```

добавляется очередной уровень обратного хода рекурсии, образуемый несколько измененными строками обратного хода рекурсии программы 1′.

```
clean(d+2(3-i),1)
add(d+2(3-i),c,2)
add(d+2(3-i),c+2,2)
add(c+1,d+2(3-i),3).
```

Так, рекурсивной детализацией в ширину программы 2′ получим программу multiplicationwidth(2), представленную в табл. 2, в левом столбце которой даны строки программы 2′.

Выше по умолчанию предполагается следующий способ представления знаков чисел: знак числа записывается многократно, сопровождая каждое элемен-

Таблица 2

Рекурсивная детализация в ширину программы 2′

Программа 2′	multiplicationwidth(2)
`eqsub (8, 0, 2, 2)`	`eqsub (8, 0, 2, 2)`
`eqsub (10, 6, 4, 2)`	`eqsub (10, 6, 4, 2)`
`xchg (2, 4, 2)`	`xchg (2, 4, 2)`

Прямой шаг рекурсии (уровень 2)	`eqsub (12, 0, 1, 1)`
	`eqsub (13, 3, 2, 1)`
	`xchg (1, 2, 1)`
	`eqsub (14, 4, 5, 1)`
	`eqsub (15, 7, 6, 1)`
	`xchg (5, 6, 1)`
	`eqsub (16, 8, 9, 1)`
	`eqsub (17, 11, 10, 1)`
	`xchg (9, 10, 1)`

Умножения
`mul (0, 2, 2)`
`mul (4, 6, 2)`
`mul (8, 10, 2)`
	`mul (0, 1, 1)`
	`mul (2, 3, 1)`
	`mul (12, 13, 1)`
	`mul (4, 5, 1)`
	`mul (6, 7, 1)`
	`mul (14, 15, 1)`
	`mul (8, 9, 1)`
	`mul (10, 11, 1)`
	`mul (16, 17, 1)`

Обратный шаг рекурсии (уровень 2)	`clean (18, 1)`
	`add (16, 8, 2)`
	`add (16, 10, 2)`
	`add (9, 16, 3)`
	`clean (16, 1)`
	`add (14, 4, 2)`
	`add (14, 6, 2)`
	`add (5, 14, 3)`
	`clean (14, 1)`
	`add (12, 0, 2)`
	`add (12, 2, 2)`
	`add (1, 12, 3)`

`clean (10, 4)`	`clean (12, 2)`
`add (8, 0, 4)`	`add (8, 0, 4)`
`add (8, 4, 4)`	`add (8, 4, 4)`
`add (2, 8, 5)`	`add (2, 8, 5)`

тарное число с относительным адресом этого числа, увеличенным на относительный адрес r области слов знаков. Число r это размер области памяти, занимаемой абсолютными значениями исходных, промежуточных и выходных данных.

3. Реализация основных операций программ и оценка объема используемой памяти

При описании и использовании основных операций программ в целях упрощения понимания функционирования были опущены детали, связанные с представлением знаков исходных и возникающих в процессе вычислений чисел. Опишем один из возможных способов уточнения описаний этих операций. Пусть в процессе работы синтезированной программы умножения чисел для хранения абсолютных значений используется область памяти с относительными адресами от 0 до $r - 1$.

Учитывая принцип наследования знака числа любой его частью, будем записывать по относительному адресу $a + r$ знак числа с адресом a (0 — если число неотрицательно или 1 — если оно отрицательно). В итоге получается, что знак числа многократно (по количеству разрядов числа) повторяется. При этом знаки исходных 2^t-разрядных множителей 2^t-кратно записываются, начиная с относительных адресов r и $a + r$. Это позволяет уточнить описания базовых функций следующим образом.

- *equ*(P,Q,N,d,s) — операция присваивания $[P]:=[Q]$, N базовых слов разрядности 2^s, записанных начиная с адреса $d + Q$, записываются, начиная с адреса $d + P$, и N базовых слов, записанных начиная с адреса $d + r + N$, записываются, начиная с адреса $d + r + P$.
- *sub*(P,Q,N,d,s) — операция вычитания: $[P]^N:=[P]^N - [Q]^N$. Начиная с адреса $d + P$, записывается абсолютное значение разности чисел из N базовых слов разрядности 2^s, абсолютные значения которых записаны по адресам $d + P$ и $d + Q$, а знаки N-кратно повторяются, начиная с адресов $d + r + P$ и $d + r + Q$ соответственно; знак разности записывается также N-кратным повторением, начиная с адреса $d + r + P$.
- *add*(P,Q,N,d,s) — операция сложения: $[P]^{N+1}:=[\mathrm{P}]^N + [Q]^N$. Начиная с адреса $d + P$, записывается абсолютное значение суммы чисел из N базовых слов разрядности 2^s, абсолютные значения которых записаны по адресам $d + P$ и $d + Q$, а знаки N-кратно повторяются, начиная с адресов $d + r + P$ и $d + r + Q$ соответственно, знак суммы записывается $(N + 1)$-кратным повторением, начиная с адреса $d + r + P$.
- *mul*$(P,Q,N,d,s,multiplication)$ — операция умножения: $[P] = [P] \cdot [Q]$. Начиная с адреса $d + P$, записывается абсолютное значение произведения чисел, модули которых как числа из N базовых слов разрядности 2^s записаны по адресам $d + P$ и $d + Q$, а знаки N-кратно повторяются, начиная с адресов $d + r + P$ и $d + r + Q$ соответственно; произведение в виде $2N$ базовых слов записывается, начиная с адреса $d + P$; знак

произведения записывается также $2N$-кратным повторением, начиная с адреса $d + r + P$. При исполнении операции *mul*(P,Q,1,d,s,*multiplication*) используется программная функция умножения multiplication($d + P$, $d + Q$,s) элементарных чисел разрядности 2^s.

- *xchg*(P,Q,N,d,s) — операция перемещения (N базовых слов разрядности 2^s, записанных, начиная с адресов $d + Q$ и $d + P$, взаимно перемещаются, как и N базовых слов, записанных начиная с адресов $d + r + Q$ и $d + r + P$ (сокращенное обозначение трех последовательно выполняемых операций присваивания).
- *clean*(P,N,d,s) — операция "обнуления": $[P]^N = 0^N$. Начиная с адреса $d + P$, записываются N нулевых базовых слов разрядности 2^s.

Число r уточняется для двух вариантов рекурсивной программы.

Для программы multiplicationdepth(k) число $r = 2(2^{k+2} - 1)$, а для программы multiplicationwidth(k) $r = 2 \times 3^k$. Таким образом, программа, полученная рекурсией в глубину, использует область памяти размером $2(2(2^{k+2} - 1) + 1)$, а для программы, полученной рекурсией в ширину, для хранения данных (с учетом элементарного слова с относительным адресом -1, используемого при исполнении операций *xchg* перемещения данных) требуется область памяти размером $4 \times 3^{k+1}$.

Программа multiplicationdepth(k) имеет преимущество по объему памяти начиная с $k = 4$.

Ниже используются как полные, так и сокращенные обозначения этих программ.

Программы multiplicationdepth(k,d,s,*multiplication*) и multiplicationwidth(k,d,s,*multiplication*) используются в программах умножения multdepth(a,b,c,k,s,*multiplication*) и multwith(a,b,c,k,s,*multiplication*) умножения 2^k — разрядных чисел по основанию 2^s.

По программе multdepth(a,b,c,k,s,*multiplication*):

1. резервируется память $[d, d + 2(2(2^{k+1} - 1)+1) - 1]$ объемом $2(2(2^{k+2} - 1) + 1)$ элементарных слов, при этом определяется базисный адрес d;
2. 2^k — разрядные множители из адресов a и b копируются по адресам d и $d + 2^k$;
3. исполняется программа multiplicationdepth(k, *multiplication*) (при известных по умолчанию d и s);
4. 2^{k+1} — разрядное произведение из адреса d копируется по адресу c.

По программе multwidth(a,b,c,k,s,*multiplication*):

1. резервируется память $[d - 1, d + 4 \times 3^{k-1}]$ объемом $4 \times 3^{k+1}$ элементарных слов, при этом определяется базисный адрес d;
2. 2^k — разрядные множители из адресов a и b копируются по адресам d и $d + 2^k$;
3. исполняется программа multiplicationwidth (k,d,s,*multiplication*);
4. 2^{k+1} — разрядное произведение из адреса $d + 2^{k+1}$ копируется по адресу c.

Оценим сложность и время исполнения синтезируемых программ. По построению, длины программ multiplicationdepth(2^k) и multiplicationwidth(2^k) при $k \geqslant 1$ определяются соответственно формулами

$$4(3^{k-1} - 1) + 14 \times 3^{k-1} = 6 \times 3^k - 4 = 2 \times 3^{k+1} - 4$$

и

$$11\frac{(3^{k-1} - 1)}{2} + 14 \times 3^{k-1},$$

которые нетрудно доказать для каждой программы по индукции.

Каждая такая программа содержит 3^k операций *mul* умножения элементарных чисел, $3^k - 1$ операций *sub* вычитания, $\frac{3(3^k - 1)}{2}$, операций *add* сложения и $\frac{3^k - 1}{2}$ операций *clean* "обнуления". Дополнительно программы multiplicationdepth(2^k) и multiplicationwidth(2^k) содержат соответственно $2 \times 3^k - 1$ и $\frac{5(3^k - 1)}{2}$ операций *equ* присваивания. Отсюда время их работы оценивается соответственно по формулам

$$t_{depth} = 3^k t_{mul} + (3^k - 1)t_{sub} + \frac{3(3^k - 1)}{2} t_{add} + \\ + \frac{3^k - 1}{2} t_{clean} + (2 \times 3^k - 1)t_{equ}$$

и

$$t_{width} = 3^k t_{mul} + (3^k - 1)t_{sub} + \frac{3(3^k - 1)}{2} t_{add} + \\ + \frac{3^k - 1}{2} t_{clean} + \frac{5(3^k - 1)}{2} t_{equ},$$

где t_{mul}, t_{sub}, t_{add}, t_{clean} и t_{equ} времена исполнения операций умножения, вычитания, сложения, "обнуления" и присваивания. Как видно, первая программа имеет преимущество по числу выполняемых операций присваивания.

4. О синтезе линейных программ умножения многочленов над конечным полем

Рассмотренные в данной работе методы синтеза программ умножения многоразрядных чисел достаточно просто модифицируются применительно к синтезу программ умножения многочленов над конечным полем $GF(p)$ характеристики p. В этом случае век-

торы (1) интерпретируются как последовательности коэффициентов многочленов

$$A(X) = a_0 + a_1X + ... + a_{n-1}X^{n-1} = \sum_{i=0}^{n-1} a_iX^i, \quad (5)$$

степень $\deg A(X)$ которых не превышает $n - 1$.

Если $n = 2m$ (т. е. четно), то сумма (5) разбивается на две части:

$$A = \sum_{i=0}^{n-1} a_iX^i = \sum_{i=0}^{m-1} a_iX^i + \sum_{m}^{n-1} a_iX^i.$$

Вынесем за скобки X^m во второй сумме и получим

$$A(X) = \sum_{i=0}^{n-1} a_iX_i = \sum_{i=0}^{m-1} a_iX^i + X^m \sum_{i=0}^{m-1} a_{i+m}X^i = A_0(X) + X^mA_1(X),$$

где многочлены $A_0(X)$ и $A_1(X)$ имеют степень не более $m - 1$, а степень многочлена $A(X)$ не более $2m - 1$. Пусть надо перемножить многочлены $A(X)$ и $B(X)$, степени которых не превышают $2m - 1$. По аналогии с разложением, полученным в разд. 1, представим каждый из них в виде суммы

$$A(X) = A_0(X) + X^mA_1(X),$$

$$B(X) = B_0(X) + X^mB_1(X).$$

Тогда по алгоритму сдвигов и сложений мы можем получить произведение многочленов по формуле

$$A(X) \times B(X) = A_0(X) \times B_0(X) + (A_0(X) \times B_1(X) + A_1(X) \times B_0(X))X^m + A_1(X) \times B_1(X)X^{2m}.$$

При этом надо выполнить четыре операции умножения многочленов степени не более, чем m.

По формуле Карацубы произведение тех же многочленов получается следующим образом

$$A(X) \times B(X) = A_0(X) \times B_0(X)(1 + X^m) + (A_1(X) - A_0(X))(B_0(X) - B_1(X))(X^m) + A_1(X) \times B_1(X)(X^m + X^{2m}).$$

При этом надо выполнить три операции умножения многочленов меньшей степени.

Линейные программы умножения многочленов, подобные программам, алгоритмы синтеза которых описаны в разд. 2, 3 и 4, можно получить по аналогичным алгоритмам[4], упрощая их с учетом следующих особенностей:

- перемножение элементарных многочленов осуществляется по известной подпрограмме. Элементарные многочлены это многочлены степени не более 2^s. Т. е. $s = 8$ в случае использования программной функции перемножения байтов, $s = 32$, если это функция перемножения машинных слов. В результате синтеза получаются программы умножения многочленов в кольце над полем $GF(p)$ по методу Карацубы как последовательности действий, содержащие операции умножения только элементарных многочленов над этим полем;
- нет необходимости отражать знаки операндов (ввиду простоты операции взятия противоположного элемента кольца многочленов);
- операции сложения и вычитания выполняются поэлементно как операции в данном поле, без переносов;
- ввиду отсутствия переносов память для них не предусматривается (длина операндов последних строк в программах 1 и 1′ уменьшается на 1) и нет необходимости "обнуления" памяти. Операцией удваивания умножаются на два длины операндов во всех строках программы без исключения.

Как следствие, более чем вдвое уменьшается объем используемой памяти.

Методы синтеза линейной программы умножения многочленов и метод удостоверения ее корректности сравнением цепочек действий над фрагментами "выходного" слова, выполняемых по программам, синтезированным разными методами, рассмотрены в работах [2, 7]. Этот способ сравнения цепочек арифметических действий, выполняемых по описанным в разд. 2 программам multiplicationdepth и multiplicationwidth, применим также для обоснования корректности этих программ умножения многоразрядных чисел.

В работе [2] показано, что аналогичный подход применим к синтезу линейных программ ряда алгебраических преобразований.

[4] Оригинальный способ синтеза "полиноминального" аналога программы multiplicationwidth предложен С. Ю. Жебетом [2, 7].

Список литературы

1. **Болотов А. А., Гашков С. Б., Фролов А. Б., Часовских А. А.** Элементарное введение в эллиптическую криптографию. Алгебраические и алгоритмические основы. М.: КомКнига, 2006.

2. **Фролов А. Б., Жебет С. Ю., Винников А. М.** О синтезе компьютерных программ алгебраических операций и преобразований в конечных полях // Вестник МЭИ. 2010. № 6. С. 136—147.

3. **Карацуба А. А., Офман Ю. П.** Умножение многозначных чисел на автоматах //ДАН СССР. 1962. Т. 145, № 2. С. 293—294.

4. **Болотов А. А., Гашков С. Б., Фролов А. Б., Часовских А. А.** Программные и схемные методы умножения многочленов для эллиптической криптографии // Известия РАН. Теория и системы управления. 2000. № 5. С. 66—75.

5. **Фомичев М. И.** Автоматизация процесса синтеза схем умножения многочленов над полем Галуа GF(2) на базе метода Карацубы // Пятая Московская международная телекоммуникационная конференция молодых ученых и студентов "Молодежь и наука". МИФИ (ТУ). 2001. URL: http://molod.mephi.ru/2001/reports.asp?rid = 157

6. **Дроздов А. Б.** Автоматическое построение декомпозиционных схем умножения многочленов над полем GF(2), основанных на методе Карацубы // Четвертая Всероссийская научная Интернет-конференция. Компьютерное и математическое моделирование в естественных и технических науках. 2002. Вып. 17. С. 6—7.

7. **Жебет А. Ю., Фролов А. Б.** Автоматический синтез программ умножения многочленов над конечным полем // Вестник МЭИ. 2008. № 6. С. 59—70.

УДК 004.415.2

В. А. Стенников, д-р техн. наук, проф., зам директора по науке,
Е. А. Барахтенко, канд. техн. наук, млад. науч. сотр., **Д. В. Соколов**, асп.,
Учреждение РАН Институт систем энергетики им. Л. А. Мелентьева СО РАН,
e-mail: barakhtenko@isem.sei.irk.ru

Применение метапрограммирования в программном комплексе для решения задач схемно-параметрической оптимизации теплоснабжающих систем

Современная практика решения задач проектирования систем теплоснабжения обуславливает необходимость учета множества свойств их подсистем, условий развития и используемого оборудования. Предлагается подход к построению программного обеспечения, который позволяет учитывать указанные особенности решения задач. Данный подход основан на применении метапрограммирования и онтологий. Излагаемый подход использован при реализации программного комплекса СОСНА-М.

Ключевые слова: *метапрограммирование, онтология, компонентный подход разработки программного обеспечения, язык программирования Java, методы оптимизации*

Введение

Оптимизация параметров многоконтурных теплоснабжающих систем представляет сложную задачу нелинейного программирования. Для ее решения в рамках развиваемой в Институте систем энергетики им. Л. А. Мелентьева теории гидравлических цепей предложены эффективные методы и алгоритмы, реализованные в виде программных комплексов [1]. Для оптимизации параметров разветвленных сетей разработан метод, основанный на применении динамического программирования [2], для расчета кольцевых сетей предложен метод многоконтурной оптимизации (МКО), основанный на идее последовательного улучшения решений [3].

Развитие рынка теплопроводов, оборудования и технологий, применяемых при строительстве теплоснабжающих систем (ТСС), значительно расширяет возможности реализации технических решений. В свою очередь, расширение этих возможностей требует их соответствующего учета при моделировании и оптимизации ТСС.

Каждый вид оборудования имеет свои характеристики и описывается своим набором математических моделей, характеризующих его параметры и технико-экономические зависимости. Традиционно в разрабатываемых программных комплексах для решения задач проектирования ТСС знания о свойствах конкретного оборудования закладывались непосредственно в программный код с помощью алгоритмических языков, оставаясь недоступными для понимания и корректировки инженерами-энергетиками. В современных условиях, когда необходимо поддерживать различный набор технологического оборудования и иметь возможность гибкого его использования при расчетах различных типов трубопроводных систем, реализация программного обеспечения (ПО) в рамках применяемых ранее парадигм программирования представляется невозможной. В связи с этим требуются разработка и применение новых адаптивных подходов, которые позволяют настраивать ПО на расчет любого типа систем с широким набором оборудования. Кроме того, при разработке и апробации новых методов и алгоритмов для математического моделирования и оптимизации ТСС необходимо ориентироваться на расширяемую архитектуру ПО, которая позволит гибко встраивать новые компоненты в существующую схему вычисления.

Выполненные исследования позволили разработать технологию проведения вычислений, обеспечивающую оптимизацию параметров ТСС с допустимым для конкретной схемы диапазоном состава оборудования и заданными типами реконструкции существующих элементов. Предложенный подход ос-

нован на применении метапрограммирования и отличается простотой добавления вновь разрабатываемых компонентов и адаптации унаследованного ПО. Эти, а также излагаемые далее новые предложения реализованы в программном комплексе (ПК) СОСНА-М.

Архитектура программного комплекса

Принципы построения и схема реализации ПК СОСНА-М базируются на применении компонентного подхода, предполагающего декомпозицию программной системы на части, которые можно многократно использовать при решении различных задач, и позволяющей наращивать функциональность программного обеспечения без внесения изменений в уже существующую часть ПО. В качестве основного языка программирования в разработанном ПК принят язык Java, который обеспечивает переносимость ПО и предоставляет широкие возможности применения объектно-ориентированного программирования (ООП) [4]. Часть вычислительных компонентов, от программных реализаций которых требуется повышенное быстродействие (реализация алгоритма определения диаметров трубопроводов в оптимизаторе параметров ТСС, решение систем уравнений в компоненте для гидравлического расчета), написаны на языках C, C++ и Fortran и подключаются к ПК через стандартный интерфейс JNI (*Java Native Interface*) платформы Java [5]. ПК позволяет организовать взаимодействие с базами данных нескольких разработчиков, имеющих разную структуру построения и работающих под управлением различных систем управления базами данных (СУБД). Для этих целей использована технология JDBC (*Java DataBase Connectivity*) — платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД. Предложенная архитектура ПК представлена на рис. 1.

При реализации ПК использованы формализованные знания в виде онтологий. В терминах, связанных с искусственным интеллектом, онтология — это эксплицитная (явная) спецификация концептуализации знаний [6]. Формально онтология состоит из иерархии понятий, их определений и атрибутов, а также из связанных с ними аксиом и правил вывода [7].

При разработке ПК СОСНА-М использованы онтологии двух видов: онтология теплоснабжающей системы (онтология ТСС) и онтология программного обеспечения (онтология ПО). Онтология ТСС содержит описание свойств подсистем ТСС, типы и параметры оборудования, иерархию структуры элементов сети, их свойства, связи между элементами, их математическое описание (например, замыкающие уравнения, формулы расчета сопротивлений и др.). Фрагмент онтологии теплоснабжающей системы приведен на рис. 2. Онтология ПО содержит описание программных компонентов и их свойств (пакеты, классы, подпрограммы), входные и выходные параметры, форматы данных, технологии доступа к компонентам системы. Для формализации онтологии используется язык описания предметной области, созданный на базе языка XML.

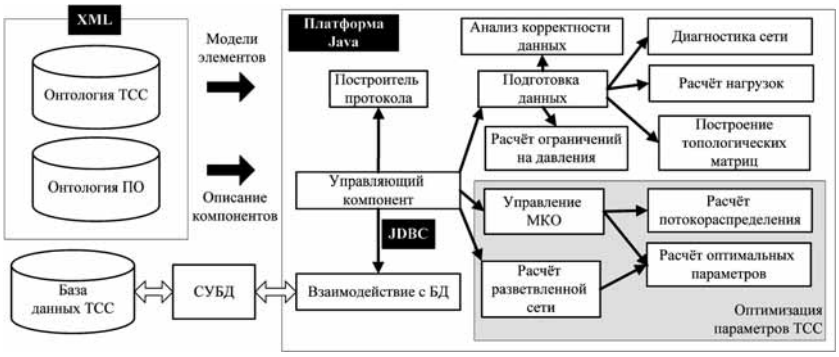

Рис. 1. Архитектура программного комплекса

Рис. 2. Фрагмент онтологии теплоснабжающей системы

Метапрограммирование

В данной работе в целях создания открытой архитектуры ПК, гибко настраиваемой на конкретную задачу в процессе ее решения, предлагается использовать *метапрограммирование*. Технологии метапрограммирования позволяют разрабатывать ПО, которое в процессе своей работы изменяет или создает другое ПО. При реализации ПК использованы два аспекта применения метапрограммирования:

1. Применение самомодифицирующегося программного кода за счет использования соответствующих механизмов современных высокоуровневых языков программирования (рефлексия, интерпретация кода на уровне программы и т. д.).

2. Генерация программного кода в процессе работы ПК путем применения генераторов программного кода из собственных языков описания предметных областей.

Первый аспект применения метапрограммирования реализует динамическое подключение новых вычислительных компонентов к работающей программе, что позволяет собирать программную систему из готовых компонентов во время ее работы для оптимизации параметров конкретной схемы ТСС. Для решения данной задачи используется подход, основанный на применении онтологии ПО, механизмов рефлексии языка программирования Java и паттернов проектирования.

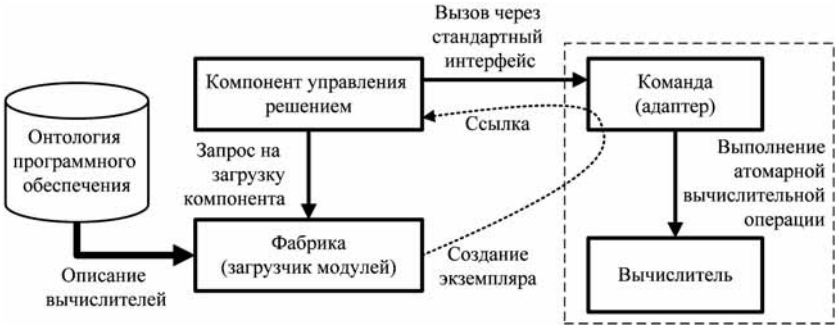

Рис. 3. Схема интеграции вычислительного компонента в программный комплекс с использованием онтологии программного обеспечения

Средства рефлексии языка Java позволяют получать информацию о компонентах, создавать экземпляры и манипулировать ими во время работы программы. Соответствующие средства находятся в пакете *java.lang.reflect* стандартной библиотеки платформы Java.

Паттерны проектирования (*design pattern*) — типовые решения некоторых характерных задач проектирования программного обеспечения. Паттерны позволяют стандартизировать интерфейсы компонентов системы и использовать их многократно при разработке программного обеспечения. В ПК СОСНА-М использованы паттерны проектирования Фабрика, Адаптер и Команда [9]. Фабрика (*Factory*) — паттерн проектирования, предполагающий использование программного компонента, предоставляющего интерфейс для создания *других* компонентов (данный компонент на основе логического имени запрашиваемого компонента загружает в память его экземпляр и возвращает ссылку на него вызвавшей подпрограмме). Команда (*Command*) — паттерн проектирования, представляющий собой подпрограмму, оформленную в виде компонента. Адаптер (*Adapter*) — паттерн проектирования, предназначенный для организации использования компонента, недоступного для модификации, через специально созданный интерфейс.

В ПК СОСНА-М все программные компоненты являются командами[1] (что позволило стандартизировать их интерфейсы и сделать компоненты взаимозаменяемыми) и разделяются на две группы, включающие системные (предназначеные в основном, для взаимодействия с базами данных, автоматической экспертизы и подготовки данных) и вычислительные компоненты (содержат реализации вычислительных алгоритмов). Вычислительные команды являются или компонентами-вычислителями, или адаптерами, предоставляющими единый интерфейс для доступа к другим компонентам-вычислителям. Данный подход позволяет достаточно просто решить задачу адаптации унаследованного ПО, например, подключить унаследованный ПК и предоставить единый интерфейс для доступа к его подпрограммам.

Схема интеграции вычислительного компонента в ПК с использованием онтологии ПО показана на рис. 3. Управляющий компонент формирует последовательность решения задач, запрашивая у фабрики по логическому имени задачи ссылку на экземпляр соответствующего компонента. Фабрика, загружая из онтологии ПО описание компонентов системы, находит компонент, соответствующий решаемой задаче, и подготавливает необходимые для его вызова структуры данных. Для создания экземпляра компонента фабрика использует средства рефлексии языка Java. После загрузки экземпляра компонента в память, соответствующая ссылка на него передается фабрикой в управляющий компонент, который, в свою очередь, передав параметры вычисления проводит через стандартизированный интерфейс вызов подпрограммы, с которой начинается работа компонента.

Второй аспект применения метапрограммирования — компиляция компонентов из сгенерированного кода во время работы программы. Язык программирования Java имеет набор необходимых средств для генерации исходного Java-кода как результата работы самой программы, его компиляции в программные компоненты и подключения их к программной системе на этапе ее выполнения. Для этого используются компоненты из пакета *javax.tools*, добавленного в Java весии SE 6 как стандартный интерфейс прикладного программирования (API) для компиляции исходного кода на языке Java.

Применение технологий метапрограммирования позволяет настраивать ПК на схемы с разным набором оборудования (стальные и металлополимерные трубы разных диаметров, насосы и арматура) и типами прокладок сети (подземная, надземная и т. п.), представляемыми различными математическими зависимостями. На основе моделей элементов сети, содержащихся в онтологии ТСС, динамически генерируются компоненты-модели под набор оборудования рассчитываемой ТСС. В качестве примера рассмотрим модель стандартного стального теплопровода. Модель включает стандартный ряд параметров: диаметр трубы, ее удельный вес, параметры и стоимость

[1] Далее компонент, реализующий паттерн проектирования Фабрика, будем называть "фабрика"; компонент, реализующий паттерн Команда — "команда"; компонент, реализующий паттерн Адаптер — "адаптер".

прокладки, теплопотери, гидравлическое сопротивление и т. д. Кроме того, модель содержит математические зависимости между параметрами. Например, значение потери напора на стальной трубе рассчитывается по формуле [1]:

$$h = s|x|x,$$

где h — потеря напора на участке, м вод. ст.; s — гидравлическое сопротивление трубопровода м/(т/ч)^2; x — расход теплоносителя, т/ч.

Сопротивление s при плотности теплоносителя $\rho = 0{,}975$ т/м^3 рассчитывается по формуле [1]:

$$s = 0{,}74 \cdot 10^{-9} \frac{(1+\alpha) l k_э^{0,25}}{d^{5,25}},$$

где α — коэффициент местных потерь; l — длина трубы, м; d — диаметр трубы, м; $k_э$ — эквивалентная шероховатость трубопровода.

Применение технологий метапрограммирования позволяет автоматизировать процесс создания компонентов-моделей для расчета конкретной ТСС. Данный подход обеспечивает новые возможности для решения задач математического моделирования и оптимизации ТСС, предоставляя инженеру инструмент, который позволяет проводить исследования, изменяя только параметры моделей. Его универсальность заключается в том, что программист избавляется от работы по ручному созданию множества компонентов.

Каждому типу элемента ТСС присваивается свой уникальный номер, который также имеет соответствующая модель элемента, хранящаяся в онтологии ТСС. В разработанном ПК используется двухуровневая модель представления данных (рис. 4): первый уровень соответствует абстрактной модели сети, где каждый элемент хранит номер соответствующей модели элемента, а второй уровень содержит компоненты, представляющие модели элементов. Отображение с первого уровня на второй осуществляется через механизмы хеширования [10]. Данный подход позволяет, используя номер модели в качестве ключа, получать ссылку на компонент-модель. Пользователь, формируя исходные данные, выбирает из списка оборудование, используемое на конкретном элементе системы (участке, насосной станции, арматуре и т. д.), а среда ПК для текущего элемента подставляет уникальный номер соответствующей модели. В результате однажды реализованная модель может быть многократно использована для описания элементов одинаковых типов.

Во время работы ПК оптимизатор получает номера используемых на схеме моделей элементов и запрашивает у фабрики компоненты, соответствующие данным моделям (рис. 5). Фабрика запускает генератор Java-кода, который создает исходный код для компиляции, используя загруженную из онтологии

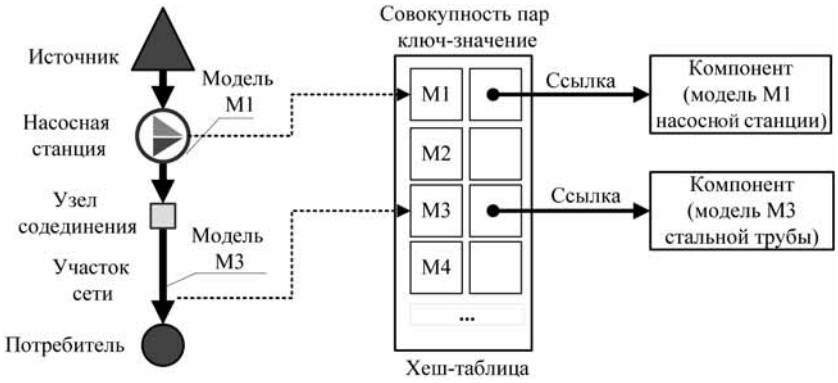

Рис. 4. Схема двухуровневой модели представления данных

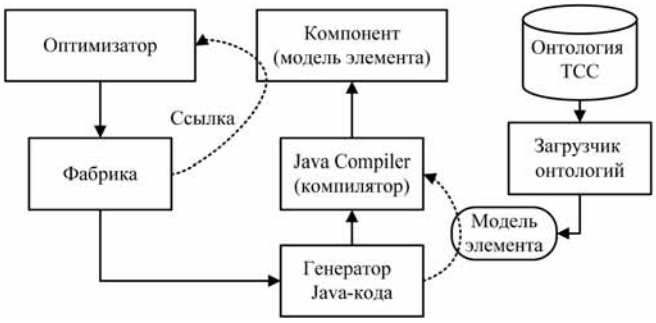

Рис. 5. Схема интеграции компонента-вычислителя и компонента, представляющего модель элемента

ТСС модель элемента. Затем запускается Java-компилятор (Java Compiler), который создает компонент, реализующий модель данного элемента. Фабрика, используя механизмы виртуальной машины Java, загружает созданный компонент в память и возвращает вычислителю ссылку на готовый для проведения вычислений компонент-модель.

Применение ПК СОСНА-М

Реализованный ПК состоит из 62 программных компонентов. Объем написанного программного кода более 4000 строк. Данный ПК используется для исследовательских целей и применяется для расчета реальных ТСС городов и населенных пунктов. На рис 6. приведена принципиальная схема ТСС одного из населенных пунктов Иркутской области. В связи с ростом тепловых нагрузок возникла необходимость решения вопросов реконструкции и развития системы. С помощью разработанного ПК была проведена оптимизация параметров теплопроводов и насосных станций ТСС. На основе выполненных расчетов были определены:

— узкие места в системе с недостаточной пропускной способностью и значительными потерями давления теплоносителя;

— типы реконструкции перегруженных участков (перекладка теплопроводов существующих участков, докладка нового теплопровода);

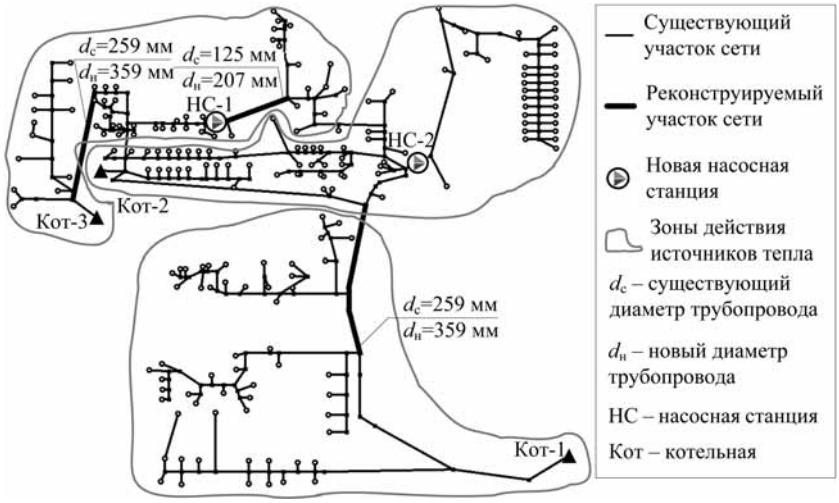

Рис. 6. Схема теплоснабжающей системы одного из населенных пунктов Иркутской области

— оптимальные давления теплоносителя в подающей и обратной магистралях (в источниках, узлах соединения участков и потребителях);

— места расположения новых насосных станций и их параметры.

Основные решения по развитию ТСС показаны на рис. 6. Новая насосная станция отмечена соответствующим условным обозначением, участки, требующие реконструкции в связи с недостаточной их пропускной способностью, выделены жирными линиями, определены типы их реконструкции и новые диаметры трубопроводов. Оптимальные зоны действия источников теплоты, полученные в результате расчета, также показаны на рис. 6.

Выводы

1. Проблема оптимального развития ТСС в связи с формированием рынка современных технологий и оборудования, ужесточением требований по качеству и надежности теплоснабжения приобретает все более высокое социально-экономическое значение. Создание эффективного методического и программного инструмента по управлению развитием ТСС является актуальной задачей.

2. Для реализации этой задачи предложена технология построения ПК на основе применения метапрограммирования. Данная технология использована при реализации ПК СОСНА-М, который позволяет находить оптимальные параметры системы как для разветвленных так и кольцевых схем ТСС с множеством источников, сложным рельефом местности, произвольным числом узлов, участков и контуров.

3. Предложенный подход позволяет специалисту прикладной области, не обладающему знаниями в области программирования, самостоятельно создавать или изменять специализированное ПО для решения задач математического моделирования и оптимизации ТСС.

4. Предложенная технология организации вычислительного процесса позволяет осуществлять оптимизацию параметров ТСС с дифференциацией допустимого набора трубопроводов, оборудования и видов реконструкции элементов конкретной теплоснабжающей системы с учетом ее особенностей и условий развития.

5. ПК СОСНА-М позволяет подготавливать рекомендации по проектированию реальных ТСС с определением: диаметров новых участков тепловой сети, видов реконструкции существующих участков, мест установки насосных станций, располагаемых напоров теплоносителя на насосных станциях и источниках, осуществлять проверку работоспособности системы и получать экономическую оценку предлагаемого решения.

Список литературы

1. **Меренков А. П., Сеннова Е. В., Сумароков С. В., Сидлер В. Г., Новицкий Н. Н., Стенников В. А., Чупин В. Р.** Математическое моделирование и оптимизация систем тепло-, водо-, нефте- и газоснабжения. Новосибирск: ВО "Наука", 1992. 407 с.

2. **Меренков А. П., Хасилев В. Я.** Теория гидравлических цепей. М.: Наука, 1985. 280 с.

3. **Сумароков С. В.** Метод решения многоэкстремальной сетевой задачи // Экономика и мат. методы. 1976. Т. 12. № 5. С. 1016—1018.

4. **Эккель Б.** Философия Java. 4-е изд. СПб.: Питер, 2009. 640 с.

5. **Монахов В. В.** Язык программирования Java и среда NetBeans. 3-е изд. СПб.: БХВ-Петербург, 2011. 704 с.

6. **Гаврилова Т. А., Хорошевский В. Ф.** Базы знаний интеллектуальных систем. СПб: Питер, 2000. 384 с.

7. **Евгенев Г. Б.** Интеллектуальные системы проектирования: учеб. пособие. М: Изд-во МГТУ им. Н. Э. Баумана, 2009. 334 с.

8. **Bartlett J.** The art of metaprogramming, Part 1: Introduction to metapro-gramming. Режим доступа: URL: http://www-128.ibm.com/developerworks/linux/library/1-metaprogl.html 17.05.2011.

9. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2008. 366 с.

10. **Кнут Д.** Искусство программирования. Т. 3. Сортировка и поиск. 2-е изд. М.: Вильямс, 2007. 824 с.

УДК 004.4

В. А. Васенин, проф., д-р физ.-мат. наук, НИИ механики МГУ им. М. В. Ломоносова, e-mail: vasenin@msu.ru, **М. А. Занчурин**, аспирант Механико-математического факультета МГУ им. М. В. Ломоносова, e-mail: maxim.zanchurin@gmail.com, **А. А. Коршунов**, канд. техн. наук, вед. науч. сотр., НИИ механики МГУ им. М. В. Ломоносова, e-mail: korsh@msu.ru

К созданию средств мониторинга состояния работоспособности элементов Грид-систем

Рассматриваются вопросы, связанные с мониторингом состояния работоспособности элементов Грид-систем. Предлагается авторское решение задачи интеграции программного комплекса FLAME, предназначенного для мониторинга состояния работоспособности с информационными сервисами пакета программ Globus Toolkit 4 и отдельно с находящимися в стадии разработки информационными сервисами пакета программ Globus Toolkit 5.

Ключевые слова: *мониторинг работоспособности Грид-систем, globus toolkit, flame*

Введение

В последние годы во всех сферах человеческой деятельности активно используются вычислительные комплексы высокой и сверхвысокой производительности, объединяющие в рамках единой системы большое число однородных вычислительных узлов, каждый из которых содержит от одного до нескольких процессоров. Следующим шагом в развитии подобных высокопроизводительных систем является создание территориально-распределенных комплексов, включающих ресурсы, вообще говоря, разнородных в аппаратно-программном плане установок в единую информационно-вычислительную среду с помощью каналов передачи данных сетей связи различного назначения. В качестве базовой методологии создания распределенных вычислительных комплексов предлагается методология Грид [1—2] (англ. — решетка), которая активно развивается в мире последние 10 лет. Такие распределенные вычислительные комплексы, аккумулирующие в себе новейшие достижения многопротокольных высокоскоростных сетей передачи данных, высокопроизводительных вычислений и современных парадигм и средств управления данными, являются новым этапом в развитии информационных технологий. С использованием Грид-систем и инфраструктур в настоящее время уже решаются сложнейшие научно-технические задачи, совершенствуется процесс подготовки необходимых для этого кадров.

Распределенные вычислительные системы способны кардинально изменить возможности многих объектов хозяйственной, научно-технической и других видов деятельности, в том числе — критически важных с позиций национальной безопасности инфраструктур [3]. Последнее обстоятельство, и это подчеркивается в заявлениях многих руководителей развитых государств, не только обеспечивает технологическое и экономическое могущество страны, но и состояние ее национальной безопасности. Такие комплексы могут найти широкое применение для решения самых разных задач — от фундаментальных научных исследований до решения задач поддержки обороноспособности страны и интеллектуальной обработки больших объемов данных, как вспомогательного средства для принятия адекватных управленческих решений органами государственной власти.

В соответствии с классическим определением, тремя ключевыми характеристиками Грид-систем являются [4]:

- децентрализованное управление входящими в состав таких систем ресурсами по причине их принадлежности независимым организациям;
- применение открытых, стандартизованных протоколов и интерфейсов взаимодействия компонентов программного обеспечения, реализующих функции, необходимые для совместного использования ресурсов;
- возможность реализации нетривиальных политик качества обслуживания пользователей при совместной эксплуатации аккумулируемых в составе таких систем ресурсов.

Globus Toolkit

В настоящее время "де-факто" стандартным инструментарием, служащим для построения Грид-систем, выступает программный пакет Globus Toolkit [5].

Globus Toolkit представляет собой набор служб, созданный партнерством Globus Alliance[1]. Последняя версия этого пакета Globus Toolkit 4 (GT4) включает средства разработки служб в соответствии с набором спецификаций WSRF[2], среду их исполнения, а также базовый набор служб, таких как управления заданиями, передачи файлов, информационного обслуживания и безопасности и ряд других. Программный пакет GT4 выполняет следующие основные функции:

- управление заданиями (*Execution Managernent*) для мониторинга и координации удаленного выполнения заданий;
- поддержка информационных служб (*Information Services*) для описания сервисов и ресурсов;
- управление данными (*Data Management*), позволяющее пользователям получать доступ к распределенным данным, передавать их и управлять ими.

Архитектура GT4 изображена на рис. 1, см. третью сторону обложки.

Работа Грид-систем по назначению, в соответствии с предъявляемыми к ним требованиями, в силу сложности и существенной распределенности составляющих ее ресурсов невозможна без постоянного контроля за их работоспособностью и готовностью к выполнению заданий. Решение этой задачи в GT4 возложено на сервис MDS, который будет рассмотрен далее более подробно.

Мониторинг состояния и обнаружение ресурсов

Мониторинг в широком толковании — специально организованное систематическое наблюдение за состоянием объектов, явлений и процессов в целях его оценки с тех или иных заранее заданных позиций (аномальный характер поведения, работоспособность, надежность, эффективность использования). В данном случае под объектами мониторинга будем понимать аппаратно-программные компоненты информационно-вычислительного комплекса высокой производительности, поддерживающую его сетевую инфраструктуру, а также исполняющиеся при этом процессы с позиции оценки состояния их работоспособности и возможности использовать эти компоненты в соответствии с принятым регламентом.

Грид-система представляет собой совокупность вычислительных ресурсов. Здесь и далее под ресурсом будем понимать отдельную вычислительную систему, которая может быть использована для решения различных вычислительных задач. Ресурс может быть составным и включать в себя несколько компонентов. Под процессом обнаружения понимается поиск ресурсов, необходимых для выполнения текущего задания. Процесс обнаружения может состоять из двух этапов, а именно: поиск ресурсов — потенциальных кандидатов для выполнения задания и выбор ресурса из списка найденных, на котором и будет выполняться задание.

Компонент MDS (*Monitoring and Discovery System*) предназначен для решения указанных задач. Сервисы, входящие в состав MDS, предоставляют механизмы для сбора параметров состояния ресурсов, а также набор триггеров, которые могут быть настроены на выполнение заданных действий при возникновении определенных условий. Более подробно эти механизмы будут рассмотрены далее.

При работе в Грид-системах можно выделить следующие две стадии мониторинга.

- **Мониторинг состояния работоспособности ресурсов.** На данной стадии осуществляется мониторинг внутри каждого кластера. Для этого могут использоваться системы анализа состояния работоспособности, настраиваемые администратором, который отвечает за поддержку регламентированных режимов работы узлов кластера. Администратор имеет доступ и соответствующие полномочия, позволяющие выявлять, локализовать и устранять неисправности внутри кластера. Некоторые интегральные характеристики, такие как состояние отдельного узла или число работоспособных узлов, могут передаваться сервисам MDS.
- **Мониторинг готовности ресурсов.** Главной целью на этой стадии является определение готовности ресурсов к выполнению заданных наборов функций. При этом не возникает необходимости решать задачи локализации и устранения неполадок в работе компонентов, входящих в состав ресурсов.

Разделение процесса мониторинга на две стадии позволяет явно распределить функции систем мониторинга состояния ресурсов на отдельном кластере и грид-системе в целом. Администраторы, имеющие доступ и соответствующие полномочия, должны устранять неисправности внутри подконтрольного кластера. В свою очередь, для работы Грид-сервисов, например, планировщика заданий, необходима информация о ресурсах, которые способны выполнять определенные функции в рамках реализации этих заданий. При этом, однако, не ставится задача обнаружения и локализации неисправностей внутри кластера.

Система MDS была спроектирована для предоставления информации о состоянии ресурсов. Она использует внешние источники для получения этой информации и обеспечивает доступ к сводной информации о состоянии ресурсов, зарегистрированных в составе Грид-системы, используя стандартные интерфейсы веб-сервисов.

Для выполнения задач мониторинга и обнаружения необходимо анализировать информацию о состоянии распределенных аппаратно-программных компонентов Грид-системы. Система MDS предоставляет так называемые сервисы агрегации, которые проводят сбор актуальной информации о состоянии от зарегистрированных в этой системе источников данных. Более подробную информацию о механизмах регистрации ресурсов можно найти по адресу http://globus.org/toolkit/docs/4.2/4.2.1/info/aggregator/pi/#mds-servicegroup-add.

[1] http://www.globus.org/alliance/about.php.

[2] Web Services Resource Framework — семейство спецификаций для веб-сервисов.

Эволюция сервиса MDS

В виде отдельного сервиса система MDS была выделена, начиная со второй версии пакета Globus Toolkit. Система MDS2 представляет собой набор информационных сервисов, построенных на базе протокола LDAP[3]. Сбор информации о состоянии осуществлялся с помощью сервиса поставщика информации GRIS (*Grid Resource Information Service*). В состав MDS2 входит также сервис GIIS (*Grid Index Information Service*), который является конфигурируемым и реализует механизмы хранения информации о состоянии. Доступ к хранимой в системе информации осуществляется по протоколу LDAP.

В дальнейшем развитие пакета GT шло по пути открытых стандартов веб-сервисов. В 2002 г. была сформулирована концепция OGSA (*Open Grid Service Architecture*). Согласно этой концепции было введено понятие Грид-служб как основной формы программных компонентов распределенных систем, а также поставлена задача стандартизации взаимодействия Грид-служб. Соответствующие спецификации были разработаны в предложениях по инфраструктуре Грид-сервисов OGSI (*Open Grid Service Infrastructure*) и реализованы в Globus Toolkit 3, вышедшем в 2003 г. Таким образом, сервисы MDS3 основываются на архитектуре OGSA и удовлетворяют спецификациям OGSI.

Далее основные положения OGSI были модифицированы и новый подход получил название WSRF — *Web-services Resource Framework*.

Основными мотивами трансформации OGSI в WSRF стали:

- введение концепции WS-ресурсов;
- улучшенное разделение функций и поддержка новых спецификаций веб-сервисов;
- более широкий взгляд на уведомления, которые являются необходимой составляющей веб-сервисов.

Решение, предложенное в WSRF, представляет собой надстройку над веб-сервисами, расширяющими их возможности средствами работы с запоминающими состояние (*stateful*) ресурсами. Подход WSRF — это пять дополняющих стандарты веб-сервисов спецификаций, которые позволяют устанавливать связь между веб-сервисами и ресурсами[4].

Версии пакета Globus Toolkit с первой по третью содержали лишь сервисы для выполнения дистанционных операций. Однако Грид-система является территориально-распределенной и выполняет роль операционной среды для различных приложений. Как подтверждает практика, чтобы эта среда сохраняла свойства обычных компьютерных систем, в ней необходимо решать задачи обеспечения надежности, детерминированности, качества обслуживания, управляемости. Они начинают решаться с использованием сервисов планирования, мониторинга заданий и устройств, учета, протоколирования и ряда других [6]. Сводные различия версий MDS можно найти в руководстве по миграции на MDS4[5]. В последней стабильной реализации GT4 применяется версия MDS4, удовлетворяющая спецификациям веб-сервисов WSRF.

MDS

Компонент MDS4 [7] пакета GT4 призван упростить решение задачи мониторинга и обнаружения ресурсов в распределенной системе. Этот компонент состоит из следующих трех различных сервисов со схожими интерфейсами:

- **MDS-Index.** Сервис индексирования осуществляет сбор и хранение данных от зарегистрированных источников Грид-системы. Информация передается в виде XML-документов. Сервис MDS-Index предоставляет доступ по HTTP-протоколу к хранимым значениям параметров, описывающих состояние ресурса. Он предоставляет веб-интерфейсы для выполнения пользовательских запросов к хранимой в системе информации, которая описывается на языке XPath. Информация, которая собирается сервисом MDS-Index, может быть использована внешними по отношению к GT приложениям для ее последующего анализа.
- **MDS-Trigger.** Данный сервис осуществляет сбор данных и выполняет различные управляющие действия в зависимости от заранее заданных критериев. Сбор данных осуществляется также, как и с использованием сервиса индексирования. Для настройки сервиса на выполнение различных управляющих действий необходимо использовать конфигурационную информацию, которая представляется в виде XML-файла.
- **MDS-Archiver.** Этот сервис осуществляет хранение информации, которая собирается сервисом MDS-Index. Сервис MDS-Archiver предоставляет интерфейсы веб-сервисов, которые используются для обработки клиентских запросов к хранимой в системе информации.

Все три перечисленных выше сервиса построены на базе среды агрегации, более подробную информацию о которой можно найти по адресу http://globus.org/toolkit/docs/4.2/4.2.1 /info/aggregator/.

Основываясь на определении источников данных вводится понятие "поставщик информации". Поставщик информации (*information provider*) — некоторое приложение, осуществляющее хранение информации о состоянии компонентов вычислительных систем и реализующее стандартные интерфейсы веб-сервисов, которые используются для получения доступа к этой информации по протоколу HTTP.

На настоящее время поставщики информации для MDS реализованы в следующих системах мониторинга:

- *Hawkeye Information Provider*;
- *Ganglia Information Provider*.

[3] Lightweight Directory Access Protocol http://tools.ietf.org/html/rfc4511.

[4] Описание спецификаций WSRF http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf.

[5] Руководство по миграции на MDS4 http://www.globus.org/toolkit/docs/4.2/4.2.0/info/mig/

Более подробную информацию о поставщиках информации можно найти по адресу http://globus.org/toolkit/docs/4.2/4.2.1/info/key/.

В настоящее время в разработке находится пятая версия пакета GT, в состав которой система MDS входить не будет. В крупнейших проектах, таких как TeraGrid, caBIG, и BIRN, сервисы системы MDS использовались в основном для создания единого хранилища информации о состоянии различных компонентов Грид-системы. Для решения этих задач в пакете GT5 планируется использовать встроенные информационные RESTful сервисы IIS (*Integrated Information Services*). Веб-сервисы, использующие стиль REST (*Representational State Transfer*) [8] вместе с протоколом HTTP, называются RESTful веб-сервисами. В результате IIS предоставляет:

- общие описания схем данных, а также механизмы хранения, поиска и передачи информации;
- работающий информационный сервис, который может использоваться организациями, не планирующими создавать свои собственные информационные службы.

В настоящее время IIS находится в разработке и полную информацию по этому проекту можно найти по адресу http://confluence.globus.щrg/display/IIS

Используемые системы мониторинга

Далее рассмотрим существующие системы мониторинга, которые выступают в роли источников данных для MDS.

Ganglia

Система мониторинга работоспособности Ganglia создавалась для распределенных высокопроизводительных вычислительных систем. Для представления данных в системе используется широко распространенная технология XML, XDR[6] — для передачи данных, и RRDtool[7] — для хранения и визуализации данных. Система Ganglia является проектом с открытым исходным кодом под лицензией BSD, который начинался в Калифорнийском университете. Она представляет собой набор модулей, из которых для каждого конкретного приложения создается система мониторинга состояния работоспособности. На рис. 2, изображена стандартная конфигурация модулей системы мониторинга Ganglia, которая используется при логическом объединении кластеров. Механизм логического объединения кластеров рассмотрен далее.

Система Ganglia предоставляет механизмы сбора и хранения информации о состоянии узлов подконтрольной системы, в частности — одного или нескольких вычислительных кластеров. В дальнейшем под понятием "состояние узла" будем понимать набор значений параметров, описывающих текущее состояние вычислительного узла, таких как доступность узла, загруженность

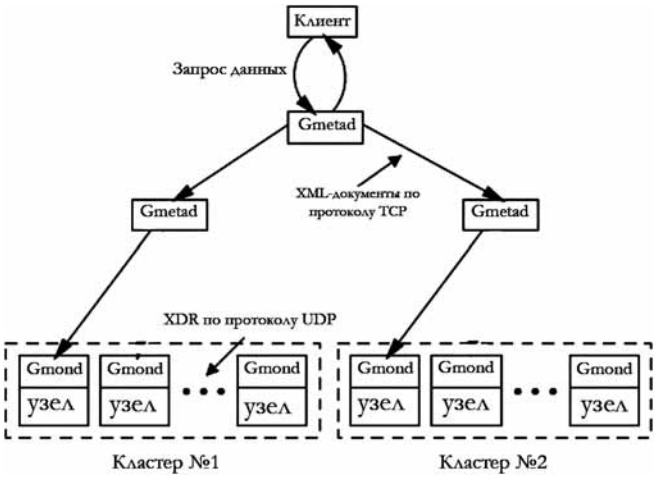

Рис. 2. Логическое объединение вычислительных кластеров

процессора, количество свободной оперативной памяти и др. В системе мониторинга Ganglia информация о состоянии узла представляется в виде XML-документов.

Система мониторинга работоспособности Ganglia состоит из следующих модулей.

- **Gmond.** Данный модуль устанавливается на каждом узле кластера, входящего в состав подконтрольной среды. Модуль Gmond предоставляет доступ к значениям параметров, описывающих состояние узла кластера, на котором данный модуль установлен. Доступность узла определяется с помощью механизма тактовых сообщений. Каждый модуль Gmond поддерживает хранение информации о состоянии всех узлов кластера. Данная избыточность необходима для восстановления состояния всех узлов кластера при условии работоспособности хотя бы одного модуля Gmond.
- **Gmetad.** Модуль Gmetad осуществляет сбор информации о состоянии подконтрольных объектов от других Gmetad- и Gmond-модулей и сохраняет их на дисковом носителе в циклической базе данных. Данный модуль используется для логического объединения кластеров. Под логическим объединением понимается хранение информации о состоянии всех узлов подконтрольной среды на одном узле, в качестве которого используется модуль Gmetad.
- **Web.** Этот модуль реализован на языке php и используется для графического отображения информации, которая хранится в модуле Gmetad.

Система Ganglia создавалась для решения задач мониторинга работоспособности узлов распределенных вычислительных систем. Для реализации логического объединения нескольких кластеров используется объединение модулей Gmond и Gmetad в виде дерева. Листовыми узлами деревьев являются модули Gmond, которые, как упоминалось ранее, устанавливаются на каждый узел вычислительных кластеров. Всеми остальными узлами являются модули Gmetad, которые осуществляют сбор информации о состоянии от своих потомков. Таким образом, корневым элементом дерева является модуль Gmetad, осуществляющий хранение информации о состоянии всех узлов подконтрольной среды.

[6] Международный стандарт передачи данных в Интернете. http://docs.freebsd.org/44doc/psd/25.xdrrfc/paper.pdf

[7] Набор утилит для работы с циклической базой данных. http://oss.oetiker.ch/rrdtool/index.en.html

Система мониторинга Ganglia реализует механизмы сбора и хранения информации о состоянии узлов, не осуществляя при этом никакой обработки собранной информации.

Более подробную информацию об архитектуре данной системы мониторинга и о механизмах сбора данных можно найти в статье [9].

Hawkeye

Приложение Hawkeye позиционируется как средство для мониторинга распределенных систем. Оно создано в университете штата Винсконсин, США. Последний релиз состоялся 26 октября 2006 г. Этот факт свидетельствует о том, что на настоящее время система не развивается.

Данное приложение имеет клиент-серверную архитектуру. На каждом вычислительном узле устанавливается клиентская часть программы Hawkeye, которая периодически отправляет по протоколу UDP hawkeye-серверу сообщения, содержащие информацию о состоянии этого узла. Информация о состоянии узла записывается на языке classified advertisements[8] (classads), который также разрабатывается в университете штата Винсконсин. Термин Classiaied Advertisements можно перевести как тематические объявления. Каждое тематическое объявление представляет собой пару атрибут-значение, например, "free_mem" и 512M. Таким образом, информация о состоянии узла представляется в виде массива таких тематических объявлений.

Серверная часть приложения Hawkeye осуществляет сбор, индексирование и хранение информации о состоянии подконтрольных узлов. Список атрибутов, описывающих состояние узла, можно расширять, создавая собственные модули для клиентской части Hawkeye. После установки по умолчанию можно осуществлять сбор значений таких атрибутов как свободное дисковое пространство, используемая оперативная память, число открытых файлов, загрузка процессора.

Средства мониторинга FLAME в системе управления ресурсами в Грид-средах

Далее опишем разработанные коллективом, в состав которого входят авторы настоящей публикации, средства (программный комплекс) мониторинга работоспособности элементов информационно-телекоммуникационных систем FLAME (*FunctionaL Active Monitoring Environment*) [10] и возможности их применения для управления ресурсами в Грид-средах.

Такой программный комплекс предназначен для контроля состояния работоспособности элементов больших вычислительных систем (кластеров), а также сложной сетевой инфраструктуры, которая является критичным фактором для потребителей ресурсов подобных систем. Далее такие средства или комплексы будем для краткости именовать средствами мониторинга, опуская иногда указание на предмет контроля и подлежащие анализу свойства. Совокупности контролируемых событий задаются путем формирования соответствующих функций. Например, если необходимо отслеживать работоспособность некоторого набора устройств и программ, то соответствующая им функция может иметь вид "логическое И от работоспособности отдельных устройств и программ". Набор всех этих функций составляет описание конфигурации контролируемой системы. Компонент, реализующий каждую функцию рассматриваемого комплекса, обращается к контролируемым устройствам и программам по протоколам SNMP или HTTP, сохраняя затем вычисленное значение в течение некоторого интервала времени. Алгоритмы управления вызовами функций следят за тем, чтобы не было перегрузки запросами контролируемых устройств, гарантируя таким образом необходимую степень актуальности информации, которую они выдают по запросам на подсистему визуализации.

Рассматриваемый программный комплекс обеспечивает мониторинг: аппаратно-программных средств вычислительных кластеров; коммуникационного оборудования (маршрутизаторов, коммутаторов, концентраторов, шлюзов и др.); работающих в сети приложений и сервисов (WEB-серверов, FTP-серверов, DNS-серверов, DHCP-серверов, серверов баз данных, WEB-сервисов); параметров операционных систем на вычислительных узлах (число процессов, число пользователей в системе и пр.); параметров сетевого трафика (число принятых/потерянных пакетов, количество байт, длина очереди на портах оборудования и т. п.); состояния устройств на вычислительных узлах (загруженность дисковых накопителей, частота их использования); физических характеристик оборудования (температура процессора, материнской платы, скорость вращения вентиляторов).

Средства мониторинга обеспечивают отображение (визуализацию) текущего состояния программно-аппаратных компонентов информационно-телекоммуникационных систем, информация о которых получается посредством опроса значений соответствующих функций. Средства автоматического определения конфигурации контролируемой среды позволяют существенно упростить работу администраторов по настройке программного комплекса.

В средствах мониторинга реализована возможность проверки получаемых параметров на удовлетворение заданным администратором условиям. В случае соответствия результата такой проверки шаблону порождается событие, которое может быть передано администратору (отображено визуализатором) или использовано другими средствами. С помощью этой методики могут быть выявлены такие события, как выход параметра за предопределенные границы регламентированного режима функционирования (перегрев узла, чрезмерная загруженность процессора или памяти, превышения границ параметрами трафика). Могут быть также обнаружены события, описываемые более сложными конструкциями. В программном комплексе для мониторинга реализована возможность порождения статусных событий, которые описывают состояние того или иного объекта контролируемой среды. Данные события поступают в подсистему корреляции. Механизмы корреляции собы-

[8] http://www.cs.wisc.edu/condor/classad/index.html

тий позволяют более точно определять место возникновения сбоя в контролируемой среде и снижают число сообщений, поступающих администратору, связанных со второстепенными событиями.

Архитектура средств мониторинга

Рассматриваемые в данной публикации средства мониторинга имеют модульную архитектуру, которая представлена на рис. 3.

Алгоритм работы каждого модуля предназначен для решения определенной задачи. Отдельные программные модули обмениваются различной информацией с другими модулями. В настоящее время в составе комплекса для мониторинга состояния работоспособности реализованы следующие модули:

- модуль управления (обеспечение взаимодействия модулей программного комплекса);
- модуль ядра (опрос компонентов подконтрольной среды, вычисление функций от значений параметров, определение факта и времени возникновения событий);
- модуль корреляции (определение источника неблагоприятного события, фильтрация дублирующих событий);
- модуль обработки клиентских запросов (обеспечение взаимодействия с внешними приложениями, например, консолью администратора).

Кроме указанных выше модулей в программном комплексе присутствует компонент автоконфигурации, реализующий алгоритм, согласно которому определяется топология подконтрольной среды. На ее основе создается конфигурационный файл, используемый модулями системы.

В составе средств мониторинга присутствует графическая консоль администратора, реализованная в виде отдельной программы. Консоль выполняет функции визуализации собранной информации и администрирования программного комплекса.

Центральное место в программном комплексе занимает модуль управления, который поддерживает маршрутизацию сообщений между другими модулями.

Сравнительный анализ

В данном подразделе представлены результаты сравнительного анализа средств мониторинга FLAME с системой мониторинга Ganglia, которая была описана выше. В данном сравнении не учитывается система мониторинга работоспособности Hawkeye, так как ее поддержка была прекращена в 2004 г.

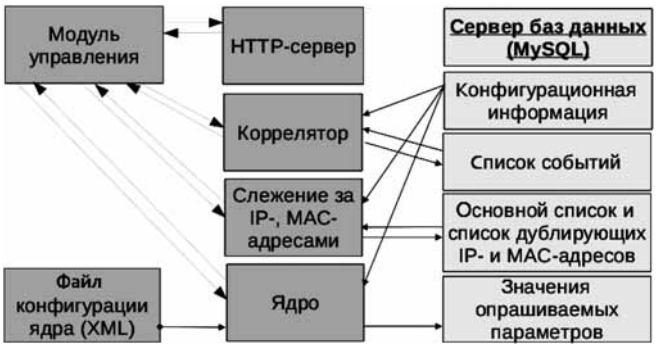

Рис. 3. Модули программного комплекса FLAME для мониторинга работоспособности элементов информационно-вычислительных и телекоммуникационных систем

Начальное конфигурирование

Для установки и первоначальной настройки системы мониторинга состояния работоспособности Ganglia необходимо установить модуль Gmond на каждый контролируемый узел кластера. Для мониторинга состояния работоспособности узлов кластера необходимо установить модуль Gmetad на один из узлов вне кластера. Архитектура и взаимодействие модулей системы кратко описаны ранее. Для получения информации о состоянии работоспособности узлов кластера необходимо использовать механизм запросов модуля Gmetad. Модуль Gmetad осуществляет сбор информации о состоянии работоспособности всех узлов, на которых установлен модуль Gmond. Таким образом при добавлении узла в кластер необходимо установить Gmond на этот узел.

Для установки и первоначальной настройки программного комплекса FLAME для мониторинга необходимо провести его установку на один узел кластера. Создание конфигурации контролируемой системы может проводиться как в автоматизированном, так и в ручном режимах. Модули автоконфигурирования программного комплекса FLAME описаны в данной работе ранее. На настоящее время пользователь может получать информацию о состоянии работоспособности с помощью графического интерфейса комплекса FLAME.

Решение задачи мониторинга состояния работоспособности

Далее представлены результаты сравнительного анализа возможностей Ganglia и FLAME по сбору и дальнейшей обработке значений параметров работоспособности контролируемых узлов.

- **Ganglia.** Сбор значений параметров осуществляется модулем Gmond. Параметры представляют собой некоторые количественные или качественные характеристики узла. Модуль осуществляет сбор значений таких параметров как загрузка процессора, количество свободной памяти, число открытых файлов, свободное дисковое пространство и др. Существует возможность создания внешних программ для сбора значений других параметров узла. Система Ganglia не осуществляет обработку собранной информации.
- **FLAME.** Сбор параметров осуществляется по протоколу SNMP. Таким образом можно собирать значения большого числа различных параметров, таких как тип процессора, тактовая частота и др. Существует также возможность создания новых скриптов и получения результатов их запуска с помощью протокола SNMP. В средствах мониторинга FLAME реализован опрос программных сервисов, таких как HTTP, FTP, SMTP и др. Программный комплекс FLAME проводит сбор значений параметров и извещает администратора о произошедших неисправностях. В некоторых случаях возможно в автоматическом режиме восстанавливать регламентированные режимы работы узлов.

Масштабируемость

Далее опишем действия, которые необходимо выполнить при добавлении в контролируемый вычислительный кластер одного узла и, отдельно, добавлении в грид-систему одного кластера.

- **Ganglia.** При добавлении одного узла в кластере необходимо установить на этот узел модуль Gmond. При добавлении одного кластера в контролируемую систему необходимо осуществить установку и первоначальную настройку Ganglia на новом кластере. Далее необходимо изменить параметры сбора данных модулей Gmetad. Последнее необходимо для получения и анализа данных о новом кластере. Понятие дерево связей было определено ранее при рассмотрении архитектуры системы мониторинга Ganglia. Система является хорошо масштабируемой, что было показано при ее тестировании, результаты которого можно найти в статье [9].
- **FLAME.** При добавлении одного узла в кластере необходимо запустить утилиту автоматического создания конфигурации или добавить новый узел в конфигурацию системы вручную. При ручном добавлении необходимо также добавить связи нового узла со старыми узлами системы.

Корреляция событий

Далее опишем возможности корреляции событий в средствах мониторинга FLAME и в системе мониторинга Ganglia. Под корреляцией событий в данном случае будем понимать поиск зависимостей между событиями, которые возникают в средствах и системах мониторинга.

Система мониторинга Ganglia поддерживает сбор информации о состоянии работоспособности, хранит собранную информацию в циклической базе данных и предоставляет пользователям возможность получать хранимую в системе информацию. Система Ganglia не предоставляет возможностей анализа хранимой информации. Аналитическая обработка хранимых данных должна осуществляться внешними системами. Таким образом, Ganglia не реализует механизмов корреляции событий.

В состав комплекса FLAME для мониторинга входит модуль корреляции событий, который используется для нахождения связей между событиями. Информация об этих связях необходима для нахождения первопричины неисправности. Корреляция событий позволяет также уменьшить число событий, которые будут показаны администратору. Таким образом, при большом числе возникающих событий модуль корреляции событий значительно упрощает работу администратора.

Результаты сравнения программного комплекса FLAME и системы мониторинга Ganglia приведены в таблице.

Возможность применения в системе управления ресурсами в Грид-средах

Учитывая возможности средств мониторинга состояния работоспособности FLAME, возможны описанные далее два варианта их использования в системе управления ресурсами в грид-средах. Рассмотрим отрицательные и положительные моменты предлагаемых вариантов.

1. Средства мониторинга состояния работоспособности, которые используются внутри вычислительного кластера.

В рассматриваемом варианте программный комплекс для мониторинга состояния работоспособности FLAME выступает в роли поставщика данных для информационных сервисов пакета GT.

На рис. 4 (см. третью сторону обложки) показана архитектура взаимодействия информационных сервисов пакета GT и программных комплексов для мониторинга состояния работоспособности.

Данная модель обеспечивает решение следующих задач.

- Сбор информации о состоянии и локализация неисправностей внутри вычислительного кластера. Эти задачи выполняют сторонние системы и средства мониторинга состояния работоспособности на кластерах. Перед администратором Грид-системы не ста-

Сравнение возможностей программного комплекса FLAME и системы мониторинга Ganglia

Параметры сравнения	Ganglia	FLAME
Начальная настройка	Установка модуля Gmond на все узлы. Установка модуля Gmetad, собирающего информацию о состоянии от модулей Gmond	Установка программного комплекса на один узел кластера. Автоматическое создание конфигурации контролируемой среды
Решение задачи мониторинга	+ Большое число собираемых параметров состояния ресурсов, простота создания собственных приложений для сбора дополнительных параметров (состояние программных сервисов в том числе) − Отсутствует какая-либо обработка собираемой информации	+ Программный комплекс FLAME проводит обработку собираемой информации (например, корреляция событий). Сбор значений большого числа параметров по протоколу SNMP. Опрос программных сервисов. Возможность выполнения сторонних приложений для получения значений параметров состояния − Необходимость настройки доступа по протоколу SNMP к контролируемым ресурсам
Масштабируемость	Система хорошо масштабируема на множестве кластеров	Программный комплекс не масштабируем и используется для мониторинга внутри одного кластера
Корреляция событий	− Какой-либо анализ собираемой информации отсутствует	+ В состав программного комплекса входит модуль корреляции событий

вится задача локализации неисправностей внутри вычислительных кластеров.

- Предварительная обработка информации о состоянии компонентов кластера. Сторонние системы и средства мониторинга предоставляют информационным сервисам пакета GT только статистическую информацию. Например, информацию о числе узлов кластера и их загруженности. Предварительная обработка информации о состоянии компонентов позволяет значительно уменьшить нагрузку на информационные сервисы программного пакета GT.
- Предоставление доступа к обработанной информации. Системы мониторинга работоспособности предоставляют доступ к обработанной информации о состоянии компонентов вычислительного кластера. Любые сторонние системы, в том числе информационные сервисы программного пакета GT, могут получить доступ к информации о состоянии, используя стандартные механизмы веб-сервисов, что упрощает интеграцию средств мониторинга с другими информационными системами.

Организация, управляющая Грид-системой, в общем случае не имеет доступа и возможностей решать задачи локализации и устранения неисправностей на кластерах, входящих в состав Грид-системы. В свою очередь, основной задачей администратора вычислительного кластера является устранение неисправностей внутри этого кластера. Таким образом происходит логичное разделение функций в решении задачи мониторинга и использование предлагаемой модели реализует это разделение.

Для использования средств мониторинга FLAME в качестве сторонней системы мониторинга в рамках предлагаемой модели необходимо реализовать дополнительный сервис. Его интерфейсы будут использоваться сторонними приложениями для получения доступа к информации о состоянии работоспособности подконтрольных узлов вычислительного кластера, которая собирается средствами мониторинга.

Указанный сервис должен обеспечить выполнение следующих функций.

- Представление информации о состоянии узлов подконтрольной среды в виде XML-документов фиксированного формата. Вся информация о состоянии работоспособности ресурсов подконтрольной среды в информационных сервисах пакетов GT 4 и GT 5 хранятся в виде XML-документов, отвечающих схеме[9] GLUE (*Grid Laboratory Uniform Environment*) версии 2.0[10].
- Предоставление стандартных интерфейсов веб-сервисов. Для интеграции с системой MDS4 создаваемый сервис должен поддерживать GET-метод, при вызове которого он возвращает XML-документ, содержащий информацию о состоянии работоспособности ресурсов. Для интеграции с сервисами IIS требуется поддержка POST-метода, с помощью вызова которого будет проводиться передача данных в хранилище IIS. Таким образом необходимо реализовать все стандартные интерфейсы веб-сервиса.

[9] http://www.w3.org/TR/xmlschema-0/
[10] http://www.ogf.org/documents/GFD.147.pdf

В случае программной реализации данного сервиса появляется возможность использования средства мониторинга FLAME в качестве поставщика информации для системы MDS и сервисов IIS. Информация о состоянии вычислительных ресурсов при этом будет передаваться информационным сервисам пакета GT в виде XML-файлов, отвечающих GLUE-схеме.

2. Средства мониторинга реализуют полную функциональность информационных сервисов пакета GT.

Информационные сервисы пакетов GT4 и GT5 проводят сбор и хранение обобщенной информации. Информационные сервисы не решают задачи локализации и устранения неисправностей, которые ставятся перед системами мониторинга состояния работоспособности. Учитывая это обстоятельство, можно сделать вывод об отсутствии необходимости замены существующих сервисов средствами мониторинга FLAME. С учетом изложенного, данный подход считается менее нецелесообразным.

Заключение

В настоящее время широкое распространение получают комплексы, использующие Грид-вычисления. Стандартным инструментом, служащим для построения грид-систем, выступает программный пакет Globus Toolkit, который эволюционирует в сторону открытых веб-сервисов.

В данной статье рассмотрена система мониторинга и обнаружения MDS, являющаяся подсистемой пакета GT4. Анализируются варианты применения средств мониторинга состояния работоспособности FLAME в Грид-системах и возможности их интеграции с информационными сервисами пакета GT4. Проведен анализ последней версии информационных сервисов IIS, которые будут использоваться в пакете GT5. Предложен вариант интеграции средств мониторинга состояния работоспособности FLAME с сервисами IIS.

Список литературы

1. **Tuecke S., Foster I., Kesselman C.** The Anatomy of the GRID: Enabling scalable virtual organizations. 2001. URL: http://www.globus.org/alliance/publications/papers/anatomy.pdf
2. **Nick J., Tuecke S., Foster I., Kesselman C.** The physiology of the GRID: An open grid services architecture for distributed systems integration. 2002. http://www.globus.org/alliance/publications/papers/ogsa.pdf
3. **Андреев О. О.** и др. Критически важные объекты и кибертерроризм. Часть 2. Аспекты программной реализации средств противодействия / под ред. В. А. Васенина. — М.: МЦНМО, 2008.
4. **Foster I.** What is the GRID? A three point checklist // GRIDToday. 2002. July.
5. **Foster I.** Globus Toolkit Primer. 2005. URL: http://globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf
6. **Эволюция** информационных сервисов GT. URL: http://www.gridclub.ru/practice/new_software.html
7. **Gt 4.2.1:** Information. URL: http://www.globus.org/toolkit/docs/4.2/4.2.1/info/
8. **Fielding R.** Architectural styles and the design of network-based software architectures. 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
9. **Culler D., Massie M., Chun B.** The ganglia distributed monitoring system: design, implementation, and experience. URL: http://ganglia.info/papers/science.pdf
10. **Ландина М. Ю., Роганов В. А., Васенин В. А., Корнеев В. В.** Система функционального активного мониторинга FLAME. // Программирование. 2003. № 3. С. 57—72.

УДК 004.78:004.056

Ю. К. Язов, д-р техн. наук, проф., **В. Б. Кадыков**, канд. техн. наук, стар. науч. сотр., **А. Ю. Енютин**, канд. техн. наук, стар. науч. сотр., **А. С. Суховерхов**, стар. науч. сотр.
ГНИИИ ПТЗИ ФСТЭК России,
e-mail: mostom84@mail.ru

Использование технологии фаззинга для поиска уязвимостей в программно-аппаратных средствах автоматизированных систем управления технологическими процессами

Рассматривается возможность применения фаззинга для выявления уязвимостей программного обеспечения автоматизированных систем управления технологическими процессами. Приводится классификация методов фаззинга и указываются основные программно-аппаратные средства их реализации.

Ключевые слова: *обеспечение безопасности информации, фаззинг, уязвимость, автоматизированные системы управления технологическими процессами*

Одной из ключевых задач обеспечения безопасности информации (ОБИ) в автоматизированных системах управления технологическими процессами (АСУ ТП) является получение оценок защищенности циркулирующей в них информации от угроз деструктивных информационных воздействий (ДИВ). Нарушение функционирования такого рода систем вследствие реализации угроз ДИВ может приводить к возникновению и развитию чрезвычайных ситуаций различного масштаба (гибели людей, крупным экономическим потерям и др.). Необходимость получения оценок защищенности информации в АСУ ТП обусловлена значительными сложностями (а порой и невозможностью на практике) организации и проведения работ по ОБИ в АСУ ТП только на основе "обобщенных" сведений о структуре и состоянии функционирования таких систем. Вместе с тем, получение такого рода оценок возможно путем проведения экспериментальных исследований по оценке уровня защищенности АСУ ТП от угроз ДИВ. Одним из направлений таких исследований является выявление уязвимостей программных и программно-аппаратных компонентов (ПАК) АСУ ТП и определение на основе этого возможных ДИВ на такие системы.

В настоящее время выявление уязвимостей программного и программно-аппаратного обеспечения проводится, как правило, только в отношении компьютерных систем "офисного" типа. При этом вопросы выявления уязвимостей в ПАК АСУ ТП освещены крайне мало. Лишь в некоторых публикациях, посвященных защите информации и автоматизации производств, указывается на актуальность выявления уязвимостей в ПАК АСУ ТП для обеспечения их безопасного функционирования [1, 2].

Одним из перспективных направлений выявления уязвимостей в программном обеспечении автоматизированных систем является применение метода фаззинга. Данный метод выявления уязвимостей, называемый также методом "серого ящика", появился на стыке методов структурного и функционального тестирования. При тестировании данным методом исследователь не имеет полной спецификации программы и ее исходных кодов, как при тестировании методом "белого ящика". Однако знает о системе больше, чем при тестировании методом "черного ящика". Сутью фаззинга является определение допустимых диапазонов входных значений параметров, используемых в программах, а также тестирование программ со значениями параметров, которые выходят за эти диапазоны либо находятся на их границах.

У каждой программы есть определенный набор выполняемых ею функций. При этом каждая программа осуществляет преобразование своего набора входных данных и формирует набор выходных данных. Формально реализация программой каждой из выполняемых ею функций $W_i(X_i)$ с преобразованием множества входных параметров X_i в множество выход-

Рис. 1. Схема реализации функций программы

ных параметров Y_i, можно представить в виде схемы, изображенной на рис. 1.

С учетом изложенного, задачей фаззинга является определение множества X входных значений для каждой реализуемой программой функции, при котором множество Y состоит из допустимых значений выходных параметров, а также определение таких значений входных параметров, при которых множество Y содержит хотя бы одно недопустимое значение. Таким образом фаззинг направлен на выявление уязвимостей, связанных с некорректной обработкой программным обеспечением сформированных для него нестандартным образом входных данных.

Изначально фаззинг использовался для тестирования качества и отказоустойчивости программного обеспечения [3]. В 1998 г. профессор Б. Миллер применил этот метод для тестирования приложений, работающих под управлением ОС Unix. Несмотря на то, что поиск уязвимостей не был приоритетом подобных исследований, их результаты показали его пригодность для решения и этого класса задач. Последний факт стал причиной популярности метода. На свет появилось большое число коммерческих и свободно распространяемых продуктов для фаззинга. Применение данного метода для выявления уязвимостей в специальном программном обеспечении, функционирующем в составе АСУ ТП (SCADA[1] — системы, программные средства OPC[2], прикладные программы контроля и управления ТП и др.) становится очень актуальным. Это обусловлено рядом обстоятельств:

- спецификой функционирования АСУ ТП является то, что ее программное обеспечение должно поддерживать значения параметров ТП в определенном диапазоне и их выход за его пределы может привести к нарушению функционирования АСУ ТП и, как следствие, к аварии на производстве;
- прекращение работы или некорректное функционирование программного обеспечения АСУ ТП, причиной которых может быть отсутствие возможности обработки им нестандартного набора входных данных, также могут привести к нарушению нормального режима функционирования АСУ ТП с недопустимыми последствиями;
- применение фаззинга позволит заранее выявить слабые места (уязвимости) в программном обеспечении АСУ ТП и исключить преднамеренное (путем реализации ДИВ с использованием данных уязвимостей) или непреднамеренное (случайным вводом некорректных данных оператором) нарушение штатного режима его функционирования путем внесения изменений в программное обеспечение АСУ ТП.

В зависимости от методов генерации данных и типов воздействия различают несколько методов фаззинга, классификация которых приведена на рис. 2.

Рассматривая методы фаззинга, классифицируемые по способу генерации данных, необходимо отметить следующее. В случае применения метода мутации данных новые данные получаются за счет незначительных изменений существующих данных, а при использовании метода генерирования данные подготавливаются "с нуля" на основе протоколов или в соответствии с заданными правилами. В этом случае при ручном изменении данных исследователю известен протокол и он пытается добиться аномального поведения исследуемого программного обеспечения за счет внесения ошибочных данных.

Для проведения осознанного внесения изменений в данные, подготовленные в соответствии с протоколом, должны быть проведены дополнительные исследования в целях определения, какие части данных должны оставаться постоянными, а какие изменяться.

Методы использования заранее подготовленных тестовых данных применяются для тестирования реализаций протоколов. Вместе с формальным описанием протокола разработчик может подготовить ряд тестовых данных, которые должны соответствующим образом обрабатываться программой, реализующей протокол.

Использование случайных данных является наименее эффективным из возможных подходов. Целевой программе передается большое количество случайных данных и при возникновении сбоя очень сложно определить настоящую его причину. Однако данный метод может быть эффективен для определения вероятности успешных реализаций различного рода сетевых атак на ПАК АСУ ТП, если она подключена к корпоративной сети предприятия и тем более к сетям общего пользования.

Методы мутации данных подразумевают наличие у исследователя определенных знаний об объекте исследования (протоколе, программном обеспечении и др.). Соответственно, они требуют большего времени на исследования. Методы генерирования данных могут быть применены для автоматизированных исследований, так как содержат стандартные наборы тестов, общих для различных объектов исследования.

Рассматривая методы фаззинга, классифицируемые по объекту воздействия, необходимо отметить следующее.

Как показано на рис. 2, к методам локального фаззинга относится фаззинг командной строки, переменных окружения и файловый фаззинг. Фаззинг команд-

[1] SCADA — Supervisory Control and Data Acquisition (диспетчерское управление и сбор данных).

[2] OPC — OLE (Object Linking and Embedding — связывание и внедрение объектов) for Process Control (OLE для управления процессами).

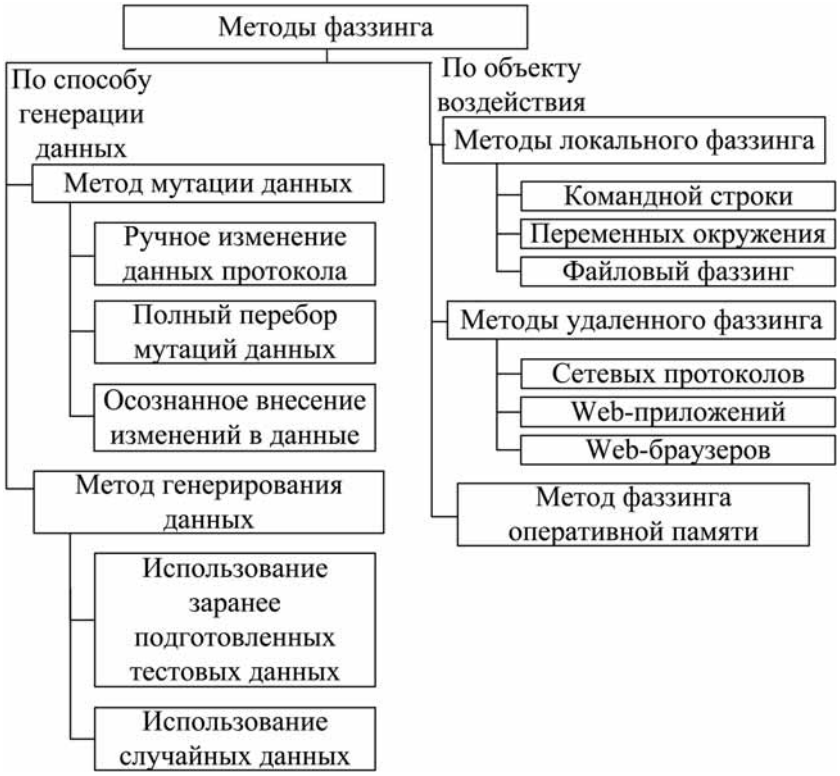

Рис. 2. Классификация методов фазззинга

ной строки используется для выявления ошибок, связанных с разбором входных параметров программ, фаззинг переменных окружения — для выявления ошибок, связанных с обработкой данных, получаемых через переменные окружения. Файловый фаззинг используется для тестирования программного обеспечения, принимающего файлы в качестве входных данных. Методы локального фаззинга могут быть использованы при выявлении уязвимостей программного обеспечения автоматизированных рабочих мест операторов ТП, программируемых логических контроллеров и различного рода серверов АСУ ТП (SCADA-серверов, архивных серверов, серверов-шлюзов и некоторых других).

К удаленному фаззингу относятся фаззинг сетевых протоколов, фаззинг wtb-приложений и фаззинг web-браузеров. Фаззинг сетевых протоколов обмена данными в АСУ ТП в настоящее время наиболее актуален. Причина в том, что исследований, посвященных выявлению уязвимостей, характерных для АСУ ТП сетевых протоколов (OPC, PROFIBUS, MODBUS и подобных им), до настоящего времени практически не проводилось. Фаззинг web-приложений и web-браузеров наиболее актуален для распределенных АСУ ТП, в которых осуществляется передача данных через сети связи общего пользования (Интернет). Однако для локальных АСУ ТП данные методы не утрачивают актуальность, поскольку большинство применяемых в АСУ ТП SCADA-систем имеют в своем составе web-клиенты, позволяющие удаленно (на корпоративном уровне) контролировать ход ТП.

Фаззинг оперативной памяти заключается в "заморозке" и моментальном снимке процесса и быстрого вброса ошибочных данных в один из шаблонов анализа ввода. После каждого случая тестирования делается новый снимок и вбрасываются новые данные. Так повторяется до тех пор, пока все случаи для тестирования не закончатся. Данный метод очень сложен для выполнения, поскольку требует хорошего знания низкоуровневого языка сборки, структуры процессной памяти и процессного инструментария.

На практике лучшие результаты приносит сочетание различных подходов к фаззингу, поскольку один метод может дополнять другой. Для автоматизации процедур фаззинга применяются программные средства, которые называются интегрированными средами фаззинга. Среди таких интегрированных сред можно выделить свободно распространяемые (Antiparser, Dfuz, SPIKE, Peach) и коммерческие (beSTORM, BPS-1000, Codenomicon, GLEG Proto Ver Professional, Mu Security Mu-4000, Security Innovation Holodeck) программно-аппаратные средства фаззинга.

Поскольку описанные методы и средства фаззинга разработаны применительно к программному обеспечению компьютерных систем в основном "офисного" типа, применение фаззинга в АСУ ТП требует учета некоторых особенностей процессов их функционирования. В офисных системах ввод некорректных данных может привести к сбою в работе программного обеспечения, что и является фактом наличия уязвимости. В программном обеспечении АСУ ТП ввод некорректных данных может не привести к сбою, однако в этом случае будут сформированы такие управляющие воздействия, которые выведут параметры ТП за пределы допустимого диапазона.

Кроме отмеченной выше возможна ситуация, когда некоторые сочетания корректных для программного обеспечения входных данных приведут к тем же последствиям. Это обусловлено возможным наличием уязвимостей в алгоритмах, на основе которых функционирует специальное программное обеспечение АСУ ТП (например, программ контроля и управления ТП). Такие уязвимости достаточно сложно обнаруживаются на этапе проектирования. В основном их выявление происходит на этапах пуско-наладочных работ или в процессе функционирования АСУ ТП при возникновении нештатных ситуаций. Досто-

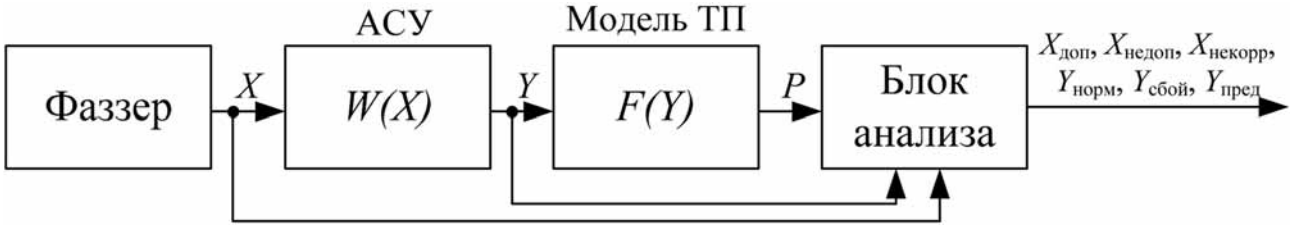

Рис. 3. Структурная схема применения фаззинга в программном обеспечении АСУ ТП

инством применения метода фаззинга в АСУ ТП является возможность выявления не только уязвимостей, приводящих к сбою программного обеспечения, но и уязвимостей построения таких систем, а также алгоритмов управления ТП, реализованных в АСУ ТП. Таким образом, множество выходных параметров программного обеспечения АСУ ТП можно представить совокупностью подмножеств:

$$Y = Y_{норм} \cup Y_{сбой} \cup Y_{пред},$$

где $Y_{норм}$ — подмножество выходных параметров, значения которых соответствуют нормальной работе программного обеспечения и регламентированному режиму функционирования ТП; $Y_{сбой}$ — подмножество выходных параметров, значения которых являются следствием сбоя в программном обеспечении; $Y_{пред}$ — подмножество выходных параметров, значения которых соответствуют нормальной работе программного обеспечения, но приводят к выходу параметров ТП за допустимые пределы.

Множество входных параметров также можно разделить на подмножества:

$$X = X_{доп} \cup X_{недоп} \cup X_{некорр},$$

где $X_{доп}$ — подмножество входных параметров, значения которых допускаются техническим регламентом ТП; $X_{недоп}$ — подмножество входных параметров, значения которых выходят за допустимые пределы или находятся на их границах; $X_{некорр}$ — подмножество явно некорректных для программного обеспечения входных данных (текстовые значения вместо числовых, сочетания текстовых с числовыми значениями и подобные им).

Таким образом, применение фаззинга в программном обеспечении АСУ ТП не ограничивается вводом некорректных данных, а охватывает весь диапазон возможных входных данных. Это обстоятельство позволяет выявить не только уязвимости конкретного программного обеспечения, но и уязвимости построения всей системы.

Следует, однако, отметить, что применение фаззинга на работающих в режиме промышленной эксплуатации АСУ ТП невозможно, так как может привести к аварийным ситуациям. Для определения выходных параметров системы, приводящих к выводу параметров ТП за допустимые пределы, необходимо наличие модели ТП, взаимодействующей с АСУ (или с адекватной моделью АСУ), осуществляющей расчет всех основных параметров процесса. Структурная схема, поясняющая применение метода фаззинга для выявления уязвимостей программного обеспечения АСУ ТП, представлена на рис. 3.

На рис. 3 фаззер — интегрированная среда фаззинга, обеспечивающая ввод необходимых данных в АСУ, $F(Y)$ — функция, преобразующая управляющие параметры АСУ (множество Y) в параметры ТП (множество P). Блок анализа определяет выходные параметры программного обеспечения АСУ ТП, являющиеся следствием сбоя, либо приводящие к выводу ТП за предельные режимы функционирования, а также входные параметры, которые к этому привели.

Таким образом, если блок анализа определит, что подмножества $Y_{сбой}$ или $Y_{пред}$ содержат хотя бы один элемент (данная ситуация соответствует нарушению штатного функционирования АСУ ТП), он зафиксирует параметры множества X, которые к этому привели. Как следствие, будет определена уязвимость программного обеспечения, заключающаяся в его неспособности сохранения штатного режима функционирования при вводе определенных входных параметров.

Применение в представленном виде фаззинга для программного обеспечения АСУ ТП позволит выявлять уязвимости и обеспечивать возможность предоставления производителям программного обеспечения рекомендаций по их устранению. Последнее обстоятельство будет способствовать повышению безопасности функционирования АСУ ТП и управляемого ею производства в целом.

Список литературы

1. **Официальный сайт** ЗАО "НТЦ "Станкоинформзащита" [Электронный ресурс]. URL: http://www.itdefence.ru.

2. **Андреев Е. Б., Кунцевич Н. А., Синенко О. В.** SCADA-системы: взгляд изнутри. М.: Издательство "РТСофт", 2004. 176 с.

3. **Саттон М., Грин А., Амини П.** Fuzzing: исследование уязвимостей методом грубой силы. Пер. с англ. СПб.: Символ-Плюс, 2009. 560 с.

CONTENTS

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

Дизайнер *Т.Н. Погорелова*. Технический редактор *Е.М. Патрушева*. Корректор *Т.В. Пчелкина*

Сдано в набор 12.09.2011 г. Подписано в печать 25.10.2011 г. Формат 60×88 1/8. Бумага офсетная. Печать офсетная. Усл. печ. л. 5,88. Уч.-изд. л. 6,79. Цена свободная.

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