УДК 519.682

TEMPLET: ЯЗЫК РАЗМЕТКИ ДЛЯ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

© 2015 С.В. Востокин

Самарский государственный аэрокосмический университет имени академика С. П. Королёва
(национальный исследовательский университет)

Поступила в редакцию 30.07.2015

В статье предлагается новый подход к описанию сети взаимодействующих процессов на традиционном языке программирования. Для выражения семантики параллельного выполнения обычно разрабатываются специальные языки программирования или расширения для последовательных языков. Библиотеки на C++, Java, C# и других языках также применяются как средство объектно-ориентированного моделирования параллельных вычислений. Однако этот способ приводит к увеличению трудоёмкости рукописного кодирования. Штатный компилятор не может обнаружить семантические ошибки, связанные с моделью программирования в подобных библиотеках. Новый язык разметки и специальная техника автоматизации программирования на основе размеченного кода позволяют решить указанные проблемы. В статье приводится подробная спецификация языка разметки без обсуждения его реализации. Язык разметки является средством описания параллельных вычислений в виде сети взаимодействующих процессов произвольной сложности. Он предназначен для программирования современных и перспективных многопроцессорных систем.
Ключевые слова: язык разметки, автоматизация программирования, языково-ориентированное программирование, параллельное программирование.

1. ВВЕДЕНИЕ

Язык Templet является предметно-ориентированным языком разметки. Он предназначен для разметки кода на последовательном процедурном или объектно-ориентированном языке программирования. Новое свойство языка разметки – явное описание семантики параллельных вычислений вида процесс-канал с использованием только размечаемого последовательного кода.

Статья написана в стиле первичной спецификации языка разметки. Она может использоваться как основа для его реализации и документирования. Намеренно пропущенные детали логически вытекают из приведённых в статье примеров кода или должны определяться в конкретной реализации.

Концепции, используемые в дизайне языка, в основном соответствуют стилю языково-ориентированного программирования [1, 2]. Применён близкий к алгебраическим конструкциям способ описания процессов в стиле CSP [3]. Идея компактного дизайна, включающего только базовые механизмы абстракции, взята из языка Oberon [4].

Язык предлагается для разработки приложений многопроцессорных параллельных систем. Код препроцессора языка и примеры программных каркасов размещены по адресу http://the-markup-language-templet.googlecode.com. Каркасы применяются в научном сервисе http://templet.ssau.ru для автоматизации программирования высокопроизводительных вычислений.

Востокин Сергей Владимирович, доктор технических наук, доцент, профессор кафедры информационных систем и технологий. E-mail: vostokin_sv@ssau.ru

2. СИНТАКСИС И СЛОВАРЬ

Язык – это бесконечное множество из цепочек символов, отвечающих заданным синтаксическим правилам. Цепочка символов в описываемом языке, называемая модуль, является файлом или группой логически связанных файлов. Кроме этого, содержимое файлов также принадлежит некоторому языку программирования, называемому базовым языком. Базовый язык является средством передачи семантики языка разметки, а именно модели процессов, обменивающихся сообщениями (см. п. 4.3).

Для описания синтаксиса используется расширенная форма Бекуса-Наура. Квадратные скобки [и] означают необязательность заключенной в них последовательности символов. Фигурные скобки { и } обозначают повторение (возможно 0 раз). Скобки (и) используются для группировки символов в правых частях синтаксических правил. Левая и правая части правил разделяются знаком =, конец правил обозначается точкой.

Нетерминальные символы (употребляемые в левый частях правил) обозначаются английскими словами, раскрывающими их значение.

Словарь языка включает следующие терминальные символы: символы блоков, разделители, идентификаторы. Символы блоков и соответству-

ющие лексические процедуры описаны в п. 3.

Символы-разделители – это одиночные знаки `'~ = ; . + ? ! | , * : & ( ) < >'` в кавычках, а также пара знаков `'->'`, находящихся в блоке схемы модуля (см. п. 3).

Идентификаторы, обозначаемые `ident`, это последовательность из букв, цифр и других знаков, не содержащая пробельных знаков и символов-разделителей. Идентификаторы также находятся в блоке схемы модуля.

3. СТРУКТУРА КОДА

Блочная структура модуля на языке разметки описывается следующими правилами.

```
   module = {base-language|user-block}
module-scheme {base-language|user-
block}.
   user-block = user-prefix base-
language user-postfix.
   module-scheme = scheme-prefix { channel
| process } scheme-postfix.
```

Код модуля состоит из одной секции схемы модуля и кода на базовом языке, в котором выделены секции *кода пользователя*. Секции размечаются в коде при помощи специальных комментариев базового языка. Например, разметка на языке C++ может иметь следующий вид. Справа показаны символы блоков языка разметки.

```
#include <runtime.h>          <-- base-language

/*templet$$include*/          <-- user-prefix
  #include <iostream>         <-- base-language
/*end*/                       <-- user-postfix

/*templet*                    <-- scheme-prefix
  *hello<function>.           <-- module scheme
*end*/                        <-- scheme-postfix
void hello(){                 <-- base-language
/*templet$hello$*/            <-- user-prefix
std::cout << "hello world!!!"; <-- base-language
/*end*/                       <-- user-postfix
}                             <-- base-language
```

Лексический анализатор языка разметки определяет границы блоков по сигнатурам, выделяя в потоке знаков строки из заданных символов. Например, блок схемы может предварять сочетание знаков `/*templet*`, а завершать – сочетание `*end*/`. Префиксы блоков пользователя содержат идентификаторы, связывающие блоки со схемой модуля: `/*templet$hello$*/` связан с *hello<function>. .

Модуль представляет собой скелет программы, а блоки пользователя – точки его расширения. Схема определяет структуру скелета программы.

Язык разметки предусматривает *алгоритм отображения*. Отображение – это преобразование модуля с синтаксически корректным блоком схемы, осуществляемое путем переписывания кода модуля. В результате этого преобразования код модуля становится изоморфным схеме модуля. Возможно появление новых блоков пользователя. Старые блоки пользователя попадают в новые места кода модуля или становятся комментариями.

4. КЛАССЫ

В схеме модуля определяются схемы *классов:* каналов и процессов. Схема *канала* имеет вид `~ name [body]. .` Определение схемы *процесса* имеет вид `* name [body]. .` Каналы описывают коммуникации процессов, выполняющих вычисления.

Например, в программе проверки тригонометрического тождества $\sin^2 x+\cos^2 x=1$ процесс класса `*Master.` выполняет рассылку значений `x` рабочим процессам класса `*Worker.` по каналам класса `~Link. .` В ответ он получает значения квадратов тригонометрических функций и вычисляет их сумму.

Классы идентифицируются по уникальным в пределах схемы модуля именам name , видимым после точки определения. Также используются идентификаторы, объявляемые в теле классов `body .` Такие идентификаторы видны в любом месте в пределах тела класса.

Реализация алгоритма отображения должна гарантировать следование кода классов на базовом языке в порядке определения классов в схеме модуля. Это необходимо для языков программирования, в которых объявление класса должно предшествовать ссылке на него.

4.1. Параметры

При помощи *параметров* можно определить специальный алгоритм отображения схем классов в код на базовом языке.

```
   params = '<' ident {',' ident} '>'.
```

Например, если при проверке тригонометрического тождества используется протокол запрос-ответ, канал можно определить как `~Link <request-reply>. .` Если для проверки тригонометрического тождества используется схема управляющий-рабочие, можно определить процесс `*Pythagorean-identity-test <master-worker, shared-memory>. .`

4.2. Правила и сообщения

Обработка *сообщений* в каналах и процессах описывается *правилами*.

```
rules = rule { '|' rule }.
rule = ident { ',' ident } '->' ident.
```

Правила – это множество пар `message -> state`, состоящих из идентификаторов сообщения и состояния, в которое происходит переход. Правила разделяются символом `'|'`. Для случая с одинаковыми состояниями `A->S | B->S` может использоваться сокращенная запись `A,B -> S`.

4.3. Отображение классов

Классы описывают базовую семантику модели программирования языка разметки, наследуемую от классов `BaseChannel` или `BaseProcess`. Язык разметки не определяет конкретный интерфейс и реализацию системы выполнения в классах `BaseChannel` и `BaseProcess`. Эти классы должны быть реализованы таким образом, чтобы обеспечить описанное ниже поведение классов `Channel` и `Process`.

```
class Channel: public BaseChannel{
 public:
  // проверка возможности чтения-записи состояния канала
   bool access_client(){...} // на стороне клиента
   bool access_server(){...} // на стороне сервера
   // отправка всего канала от клиента серверу
   void send_client(){...}
   // отправка всего канала от сервера клиенту
   void send_server(){...}
...
};

class Process: public BaseProcess{
 public:
  virtual void recv(BaseChannel*);// получаем данные по каналу
   // привязка канала к процессу в качестве клиента или сервера
   bool bind_client(BaseChannel*) {...}
   bool bind_server(BaseChannel*) {...}
...
}
```

Класс `BaseChannel` должен быть реализован так, чтобы на его основе можно было реализовать абстрактный канал `Channel` со следующими свойствами. Доступ к каналу могут иметь поочерёдно два процесса: клиент и сервер. Клиент имеет доступ в начале вычислений. Методы `access_client()` и `access_server()` позволяют проверить наличие доступа процессом-клиентом или процессом-сервером. Методы `send_client()` и `send_server()` служат для передачи доступа другому процессу процессом-клиентом или процессом-сервером.

Класс `BaseProcess` должен быть реализован так, чтобы на его основе можно было реализовать абстрактный процесс Process со следующими свойствами. Методы `bind_client()` и `bind_server()` устанавливают связь процесса с каналом в роли клиента или сервера. Метод `recv()` вызывается в момент получения процессом доступа к каналу. Этот канал передаётся в качестве аргумента вызова. Если процесс получает доступ к нескольким каналам, то происходит несколько последовательных вызовов метода `recv()` в произвольном порядке. Если некоторый процесс передаёт доступ к каналу другому процессу, то другой процесс рано или поздно получит доступ к этому каналу.

5. КАНАЛ

В схеме канала определяется протокол взаимодействия между процессом в роли клиента и процессом в роли сервера.

```
channel = '~' ident [params] ['=' state {';' state}] '.'.
```

Протокол задаётся в виде правил смены *состояний* канала под действием сообщений, отправляемых по нему.

5.1. Состояние

Состояние имеет уникальный в пределах определения канала идентификатор. Определение состояния содержит атрибуты и набор правил перехода в новое состояние.

```
state = ['+'] ident [ ('?'|'!') [rules] ].
```

Символ `'+'` является атрибутом начального состояния. Символ `'?'` обозначает, что *сообщение-вопрос* передаётся от клиента в сервер. Символ `'!'` обозначает, что *сообщение-ответ* передаётся от сервера клиенту.

Например, протокол запрос-ответ можно определить как показано ниже.

```
~Link = +BEGIN? Request -> PROCESSING;
        PROCESSING! Reply -> BEGIN.
```

Протокол канала для проверки тригонометрического тождества можно записать так.

```
~Link = +BEGIN ? ArgCos -> CALCCOS
```

```
| ArgSin -> CALCSIN;
              CALCCOS ! Cos2 -> END;
CALCSIN ! Sin2 -> END.
```

Допускается не объявлять состояния, из которых нет переходов. Состояние, объявленное без указания правил и атрибутов (‘?’ или ‘!’), считается состоянием клиента (подразумевается символ ‘?’).

5.2. Отображение канала

Отображение схемы канала в код на базовом языке программирования показано ниже. Представлен интерфейс для доступа к каналу в контексте вызова обработчика сообщений в процессах.

```
   class Channel : public BaseChannel{
   ...
   public:
     struct Message{// структура данных
сообщения Message
   /*templet$Channel$Message*/
     // блок кода пользователя
     ...
   /*end*/
     };

      // поле класса для хранения со-
общения Message
     Message Message_get;

     // проверка доступа для чтения-за-
писи сообщения Message
     // на стороне клиента и сервера
     bool Message_read_client(){...}
     bool Message_write_client(){...}
     bool Message_read_server(){...}
     bool Message_write_server(){...}

     // отправка сообщения Message сер-
вером или клиентом
     void Message_send_server(){...}
     void Message_send_client(){...}
   ...
   };
```

Подобные фрагменты повторяются для каждого сообщения канала.

6. ПРОЦЕСС

В схеме процесса определяется алгоритм обработки поступающих по каналам сообщений и выдачи ответных сообщений.

```
   process = ‘*’ ident [params] [‘=’
((ports [‘;’ actions]) | actions) ] ‘.’.
```

Определение процесса включает определения *портов* и *действий*, а также вызовы *функций пользователя* из действий.

6.1. Порты

Порт обозначает точку привязки канала и процесса. Порт имеет уникальный в пределах определения процесса идентификатор. Определение порта содержит атрибуты и набор правил перехода к выполнению действий.

```
   ports = port {‘;’ port}.
   port  = ident ‘:’ ident (‘?’|’!’)
[(rules [‘|’ ‘->’ ident])|( ‘->’
ident)].
```

Атрибут *канал порта* указывается после символа ‘:’ вслед за идентификатором порта. Символ ‘?’ является атрибутом *серверного порта*. Сообщения в правилах, следующих за символом ‘?’, являются сообщениями-вопросами. Символ ‘!’ является атрибутом *клиентского порта*. Сообщения в правилах, следующих за символом ‘?’, являются сообщениями-ответами.

Например, выбор вычисления квадрата синуса или квадрата косинуса в зависимости от сообщения может быть описан как `p:Link ? ArgSin -> sin2 | ArgCos -> cos2`. Может использоваться сокращенная форма записи `p : Link ? -> sin2`, позволяющая указать действие, выполняемое для всех сообщений, неуказанных в правилах явно.

6.2. Отображение порта

Отображение порта в код на базовом языке программирования показано ниже. Представлен фрагмент метода обработки сообщений `void recv(BaseChannel*c)` для порта `Port:Channel!Message1->Action1|Message2->Action2|->SomeAction`.

```
   class Process : public BaseProcess{
   public:
     // связывание порта Port с каналами
типа Channel
      bool  Port_bind_client(Channel*)
{...}

     // необязательная процедура порта
Port
     void Port_call(Channel*){
   /*templet$Process$Port*/
     // блок кода пользователя
   ...
   /*end*/
     }
      // поле порта для хранения при-
вязки к каналу
     Channel* Port_port;

     // обработчик сообщений для порта
Port в методе recv()
     void recv(BaseChannel* c){
```

```
      int sel;//селектор
        sel = c->selector;//в канале
хранится информация о порте

      switch(sel){
        case Port_label:
         {// статическое приведение к
типу Channel
              Channel* _c=static_
cast<Channel*>(c);

          Port_call(_c);  // необяза-
тельный вызов процедуры порта
          Port_port = _c; // необяза-
тельная привязка к порту
                          // если к
нему подключены несколько каналов
          sel=UNKNOWN;

           // проверка возможных со-
общений
            if(_c->Message1_read_cli-
ent()) then sel=Action1_label;
        else if(_c->Message2_read_cli-
ent())then sel=Action2_label;
          else sel=SomeAction_label;

         assert(sel!=UNKNOWN);// пре-
рывание, если поступило неизвестное
          break;              // со-
общение
        }//case
        ...
      }//switch
    }//recv
  };
```

Подобные фрагменты кода повторяются для каждого порта процесса. Если порт определён как серверный порт с символом `'?'`, то используются методы с суффиксами `_server` вместо `_client`.

6.3. Функция пользователя

Функция пользователя обозначает вызов метода с кодом пользователя. В вызове указывается имя и список аргументов.

```
  call = ident '(' [args] ')'.
  args = ident ('?'|'!') ident {',' 
ident ('?'|'!') ident}.
```

Список аргументов состоит из пар порт-сообщение, разделённых запятыми. Если имя порта и сообщение разделяет знак `'?'`, сообщение читается. Если имя порта и сообщение разделяет знак `'!'`, сообщение записывается для дальнейшей отправки.

Метод в коде процесса, реализующий функцию пользователя, должен возвращать признак успешного или неуспешного завершения.

6.4. Действия

Действие задаёт последовательность вызова функций пользователя и отправки сообщений. Оно имеет уникальный в пределах определения процесса идентификатор. Определение действия содержит атрибуты и одну или несколько функций пользователя.

```
  actions = action {';' action}.
  action = ['+'] [ident ':'] disjunction
['->' ([ident] '|' ident) | ident].

  disjunction = conjunction {  '|'
conjunction}.
  conjunction = call {'&' call}.
```

Атрибут `'+'` указывает действие, запускаемое в начале вычислений извне системы процессов и каналов. В качестве уникального идентификатора действия может использоваться метка перед знаком `':'`. Если метка опущена, идентификатором действия является идентификатор первой по порядку функции пользователя. После знака `'->'` может указываться действие, выполняемое в случае успешного завершения. После знаков `'->' '|'` может указываться действие, выполняемое в случае неуспешного завершения.

Например, процессы для проверки тригонометрического тождества можно определить следующим образом.

```
  *Master =
       p1:Link ! Sin2 -> join; p2:Link
! Cos2 -> join;
         +fork(p1!ArgSin,p2!ArgCos);
join(p1?Sin2,p2?Cos2).

  *Worker =
        p : Link ? ArgSin -> sin2 |
ArgCos -> cos2;
      sin2(p?ArgSin,p!Sin2);
cos2(p?ArgCos,p!Cos2).
```

Применяя атрибуты для построения цепочек действий, процесс `Worker` можно определить следующим образом.

```
  *Worker =
        p : Link ? -> DO;
     DO:sin2(p?ArgSin,p!Sin2)->|cos2;
cos2(p?ArgCos,p!Cos2).
```

Вызовы функций пользователя в действиях могут разделяться знаками «конъюнкция» `'&'` и «дизъюнкция» `'|'`. Знак `'&'` имеет более высокий приоритет, чем `'|'`, группировка выполняется слева-направо. При указании нескольких функций пользователя в одном действии ис-

пользуется семантика сокращенных вычислений (short-circuit evaluation semantics). То есть определение `A()&B()->C|D` эквивалентно `A()->B|D; B()->C|D`, а – `A()|B()->C|D` эквивалентно `A()->C|B; B()->C|D` .

Возможно определение процесса `Worker` с использованием группировки функций пользователя в одном действии.

```
*Worker =
       p : Link ? -> DO;
     DO:sin2(p?ArgSin,p!Sin2)|cos2(p
?ArgCos,p!Cos2).
```

6.5. Отображение действия

Отображение действия в код на базовом языке программирования показано ниже. Представлен фрагмент функции обработки сообщений `void recv(BaseChannel*c)` для действия `Action(p1?Message1,p2!Message2)->A1|A2`, а также объявлений портов `p1:Channel!` и `p2:Channel?` .

```
class Process : public BaseProcess{
public:
   // функция пользователя для дей-
ствия Action
    bool Action_call(Channel::Messag
e1*m1,Channel::Message2*m2){
/*templet$Process$Action*/
// блок кода пользователя
/*end*/
    }
   // обработчик сообщений для дей-
ствия Action в методе recv()
  void recv(BaseChannel* c){
    int  sel; // селектор
     bool res; // используется при
обработке действий
          ...
    // первое действие в цикле было
выбрано выше в обработчике порта (п.6.2)
    for(;;)switch(sel){
      case Action_label:
      { //проверка возможности чте-
ния-записи сообщений
        res=p1_port->Message1_read_
client()&&p2_port->Message2_write_
server();

         //если тест пройден, вызы-
ваем функцию пользователя для действия
Action
        if(res)res=Action_call(&p1_
port->Message1_get,&p2_port->Message2_
get);

       //если предыдущие шаги успеш-
ны, отправляем сообщения
         if(res){p2_port->Message2_
send_server();}

        //если все шаги успешно вы-
полнены, переходим к действию A1,
       //иначе переходим к действию A2
        if(res) sel=A1_label; else
sel=A2_label;
       break; // выполняем следующий
виток цикла
}//case
...
   }//for loop
  }//recv
};
```

Подобные фрагменты кода повторяются для каждого действия процесса. Сложные действия, включающие несколько функций пользователя, предварительно преобразуются в простые вида `A()->B|C` .

Реализация процесса на базовом языке должна обеспечивать следующие правила вычислений для действий. Функция пользователя запускается, если управление перешло к действию и возможно чтение (признак `'?'`) или запись (признак `'!'`) всех сообщений, указанных в списке аргументов. Отправка сообщений (с признаком `'!'`) происходит, если функция пользователя была запущена и вернула значение «истина». Переход к действию по ссылке `-> A` происходит, если функция пользователя была запущена и вернула значение «истина». Переход к действию по ссылке `-> |A` происходит, если функция не была запущена или вернула значение «ложь».

7. ПРОГРАММА

Программа представляет собой *сеть объектов*. Объекты являются экземплярами классов базового языка, в которые отображаются схемы каналов и процессов.

7.1. Сеть объектов

Сеть объектов описывается на базовом языке программирования. Пример сети объектов для проверки тригонометрического тождества выглядит следующим образом.

```
void main(){

  TempletProgram p;       // объект,
управляющий программой

  Link link1(p),link2(p);// объекты
каналов
  Master m;               // объекты
процессов
  Worker w1,w2;
```

```
    // привязка каналов к главному процессу Master
    m.p1_port(link1);m.p2_port(link2);

    // привязка каналов к рабочим процессам Worker
    w1.p_port(link1);w2.p_port(link2);

     // сеть из объектов построена, вводим данные и запускаем
     cout<<»input x:»; cin>>m.x;
     p.run();
     cout<<»sin2(x) + cos2(x) = «<<m.sin2x_plus_cos2x;
  }
```

Любая программа на языке разметки является сетевой структурой данных произвольной сложности.

7.2. Исполнение

Реализация программы на базовом языке программирования должна предусматривать возможность недетерминированного выполнения, что следует из определения модели программирования в п. 4.3. Недетерминизм выполнения моделируется при помощи псевдослучайных чисел.

```
  void TempletProgram::run()
  {
    size_t rsize;
    // выполняем, пока в очереди есть каналы
    while(rsize=ready.size()){
        //случайным образом выбираем канал, выполняющий передачу сообщения
       //извлекаем этот канал из очереди сообщений
      //изменяем его состояние на «нет передачи»
       int n=rand()%rsize;    a u t o it=ready.begin()+n;
       BaseChannel*c=*it;     r e a d y . erase(it); c->sending=false;
      //из канала извлекаем процесс, в который выполняется передача
       //запускаем метод обработки сообщения recv в этом канале
      //и передаём сам канал в качестве аргумента метода recv
       c->p->recv(c);
    }
  }
```

Для истинно параллельного исполнения кода используются необходимые API. При этом может потребоваться специальный алгоритм отображения схемы модуля в код.

8. РЕЗУЛЬТАТЫ

Разработка прототипа препроцессора и вспомогательных инструментов языка разметки показала следующие преимущества нашего подхода.

Для передачи смысла параллельного алгоритма не требуется дополнительных языковых конструкций в базовом языке, как и при использовании объектно-ориентированных библиотек STL [5], TBB [6], CCR [7], Boost [8], но применение языка разметки дополнительно уменьшает объём ручного кодирования.

Обеспечивается надежная защита от ошибок программирования, сопоставимая с известными языками Go [9], Occam [10], Limbo [11], Erlang [12], также использующими модель процесс-канал. Статический контроль типов в языке реализации помогает предотвратить неправильное соединение источника и получателя сообщений. Семантическая проверка может быть реализована на уровне препроцессора языка разметки. Например, по схеме модуля можно проверить достижимость всех коммуникационных состояний в каналах и возможность вызова методов обработки сообщений в процессах. Проверка может осуществляться также во время выполнения программы. Если связанные процессы не выполняют предписываемый каналом протокол обмена сообщениями, вычисления остановятся.

Поведение программы можно детальнее исследовать при помощи проблемно-ориентированного отладчика, если алгоритм отображения сформирует дополнительный код для передачи информации в отладчик [13].

Возможно предсказание производительности параллельной программы методом дискретно-событийного моделирования. Для этого применяется специальная среда исполнения.

Язык разметки является средством скелетно-ориентированного программирования и повторного использования кода [14, 15]. Можно спроектировать универсальный скелет для программ с похожим потоком управления и адаптировать его для конкретных приложений, изменяя код обработчиков сообщений и передаваемых в сообщениях данных. Данный метод эффективен для программирования современных многоядерных систем [16, 17].

Разметка может быть представлена в наглядной графической форме. В отличие от классических визуальных языков [13,18], визуальное представление в языке разметки является дополнительным средством предварительного проектирования и документирования. Визуально представлены отдельные объекты как, например, в [19].

В языке разметки применена техника описания параллелизма последовательным кодом.

Обычно она используется в средствах инкрементного распараллеливания путём разметки кода в промышленных [20, 21] и исследовательских системах [22, 23] для представления итеративного или рекурсивного параллелизма. Мы адаптировали эту технику для модели процесс-канал. Код в нашей системе также последовательный, но он генерируется, а не пишется вручную.

Язык разметки применим к разным языкам реализации и параллельным библиотекам времени исполнения. Он совместим с современными технологиями [24, 25], используемыми в промышленном программировании.

БЛАГОДАРНОСТИ

Автор выражает благодарность Самарскому государственному аэрокосмическому университету за поддержку этого исследования. Библиотека времени выполнения языка была реализована для использования на суперкомпьютере «Сергей Королёв» СГАУ при постоянной помощи со стороны обслуживающего персонала.

Работа выполнена при государственной поддержке Министерства образования и науки РФ в рамках реализации мероприятий Программы повышения конкурентоспособности СГАУ среди ведущих мировых научно-образовательных центров на 2013-2020 годы. Работа частично поддержана грантом РФФИ № 15-08-05934 А.

СПИСОК ЛИТЕРАТУРЫ

1. *Ward M.P.* Language-oriented programming // Software-Concepts and Tools. 1994. Т. 15. № 4. С. 147-161.
2. *Dmitriev S.* Language oriented programming: The next programming paradigm: [Электронный документ] // JetBrains onBoard. 2004. Т. 1. № 2. URL: http://www.onboard.jetbrains.com/articles/04/10/lop/ (дата обращения 16.11.2014).
3. *Hoare C. A. R.* Communicating sequential processes // The origin of concurrent programming. Springer New York, 2002. С. 413-443.
4. *Wirth N.* The programming language Oberon // Software: Practice and Experience. 1988. Т. 18. № 7. С. 671-690. DOI:10.1002/spe.4380180707.
5. *Stroustrup B.* The C++ programming language. Addison-Wesley, 2013.
6. *Reinders J.* Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc., 2007.
7. *Richter J.* Concurrent Affairs-Concurrency and Coordination Runtime //MSDN Magazine-Louisville. 2006. С. 117-128.
8. *Schling B.* The boost C++ libraries. Xml Press, 2011.
9. The Go Programming language specification - The Go programming language: [Электронный документ]. URL: http://golang.org/doc/go_spec.html). (дата обращения 28.05.2014).
10. INMOS Limited. Occam programming manual. Prentice Hall Direct, 1984.
11. *Ritchie D.M.* The Limbo programming language. In Inferno 3rd Edition Programmer's Manual, vol. 2. Vita Nuova Holdings Ltd, 2000.
12. *Larson J.* Erlang for concurrent programming // Communications of the ACM. 2009. Т. 52. № 3. С. 48-56. DOI:10.1145/1467247.1467263.
13. *Browne J. C. et al.* Visual programming and debugging for parallel computing // IEEE Concurrency. 1995. Т. 3. № 1. С. 75-83. DOI:10.1109/88.384586.
14. *Cole M.* Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming //Parallel computing. 2004. Т. 30. № 3. С. 389-406. DOI:10.1016/j.parco.2003.12.002.
15. *González-Vélez H., Leyton M.* A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers //Software: Practice and Experience. 2010. Т. 40. №12. С. 1135-1160. DOI:10.1002/spe.1026.
16. *Karasawa Y., Iwasaki H.* A parallel skeleton library for multi-core clusters //Parallel Processing, 2009. ICPP'09. International Conference on. IEEE, 2009. С. 84-91. DOI:10.1109/ICPP.2009.18.
17. *Aldinucci M., Danelutto M., Kilpatrick P.* Skeletons for multi/many-core systems //PARCO. 2009. С. 265-272. DOI:10.3233/978-1-60750-530-3-265.
18. *Johnston W.M., Hanna J.R., Millar R.J.* Advances in dataflow programming languages //ACM Computing Surveys (CSUR). 2004. Т. 36. № 1. С. 1-34. DOI:10.1145/1013208.1013209.
19. *Philippi S.* Visual programming of concurrent object-oriented systems // Journal of Visual Languages & Computing. 2001. Т. 12. № 2. С. 127-143. DOI:10.1006/jvlc.2000.0192.
20. *Dagum L., Menon R.* OpenMP: an industry standard API for shared-memory programming //Computational Science & Engineering, IEEE. 1998. Т. 5. № 1. С. 46-55. DOI:10.1109/99.660313.
21. *Blumofe R. D. et al.* Cilk: An efficient multithreaded runtime system // Journal of parallel and distributed computing. 1996. Т. 37. № 1. С. 55-69. DOI:10.1006/jpdc.1996.0107.
22. *Konovalov N. A., Krukov V. A., Sazanov Y. L.* C-DVM-A Language for the Development of Portable Parallel Programs //Programming and Computer Software. 1999. Т. 25. № 1. С. 46-55.
23. *Abramov S. et al.* OpenTS: an outline of dynamic parallelization approach //Parallel Computing Technologies. Springer Berlin Heidelberg, 2005. С. 303-312.

24. *Selic B.* The pragmatics of model-driven development // IEEE software. 2003. T. 20. №. 5. C. 19-25. DOI:10.1109/MS.2003.1231146.

25. *Atkinson C., Kuhne T.* Model-driven development: a metamodeling foundation //Software, IEEE. 2003. T. 20. № 5. C. 36-41. DOI:10.1109/MS.2003.1231149.

TEMPLET: A MARKUP LANGUAGE FOR CONCURRENT PROGRAMMING

© 2015 S.V. Vostokin

Samara State Aerospace University named after Academician S.P. Korolyov
(National Research University)

In this paper we propose a new approach to the description of a network of interacting processes in a traditional programming language. Special programming languages or extensions to sequential languages are usually designed to express the semantics of parallel execution. Libraries in C++, Java, C#, and other languages are also used as a means of object-oriented modeling of parallel computations. However, this method leads to an increase in workload of a manual coding. Besides, stock compilers can not detect semantic errors related to the programming model in such libraries. The new markup language and a special technique of automatic programming based on the marked code can solve these problems. The article provides a detailed specification of the markup language without discussing its implementation details. The markup language is designed for the description of parallel computation as a network of interacting processes of arbitrary complexity. It is used for programming of current and prospective multiprocessor systems.
Key words: markup language, automatic programming, language-oriented programming, concurrent programming.

Sergey Vostokin, Doctor of Technics, Professor at the Information Systems and Technologies Department.
E-mail: vostokin_sv@ssau.ru