Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

И.С. Ануреев

ЯЗЫК ATOMENT: СТАНДАРТНАЯ БИБЛИОТЕКА

Препринт
158

Новосибирск 2010

Язык Atoment является предметно-ориентированным языком разработки методов верификации программ. Он используется в мультиязыковой программной системе ускоренной разработки и апробации методов и техник верификации Спектр. Удобный специализированный язык позволяет пользователю системы описывать в естественной нотации методы и техники верификации, верифицировать алгоритмы в различных предметных областях, добавляя при необходимости свои языки для их представления, разделять методы и техники верификации с другими пользователями системы и комбинировать их. В работе описана стандартная библиотека языка Atoment, представляющая собой множество схем выполнения. Определена семантика этих схем выполнения. Схемы снабжены примерами их применения.

Siberian Branch of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

I.S. Anureev

THE ATOMENT LANGUAGE: STANDARD LIBRARY

Preprint
158

Novosibirsk 2010

The Atoment language is a domain-specific language of development of program verification methods. It is used in the multilanguage software system Spectrum of rapid development and testing of verification methods. The easy-to-use specialized language allows a user of the system to describe verification methods in natural notation, verify algorithms for different object domains, adding new languages for their representations as necessary, share verification methods with other users and combine them. In this paper the standard library of the Atoment language is described. It is presented by a set of execution schemes. Semantics of these schemes is defined. The schemes are illustrated by examples of their use.

1. ВВЕДЕНИЕ

Современная тенденция в области верификации программ — переход от разработки методов верификации программ, применяемых для небольших программ на модельных языках программирования, к верификации больших программных систем на индустриальных языках программирования. Эта тенденция состоит в выделении практически значимых свойств программ и построении специализированных методов и техник анализа и верификации. Унификация и формализация процессов описания таких свойств и построения для них методов и техник является важной открытой проблемой. Для индустриальной верификации также характерно использование комбинации различных методов верификации. Это приводит к появлению новых гибридных методов верификации программ. Создание средств накопления, анализа и формализации опыта, накопленного в области интеграции различных методов верификации — еще одна важная открытая проблема.

Цель проекта Спектр — разработка нового подхода к верификации императивных программ, который позволяет интегрировать, унифицировать и комбинировать методы и техники верификации программ, накапливать и использовать знания о них. Особенностью подхода является использование предметно-ориентированного языка Atoment [1] разработки средств верификации программ, который позволяет представить в едином унифицированном формате как методы и техники верификации, так и данные для них (программные модели, аннотации, логические формулы). Систему Спектр [2], базирующуюся на этом языке, можно рассматривать в качестве как специализированной среды разработки инструментов в области верификации программ, так и информационной системы, которая аккумулирует знания в этой области и обеспечивает доступ к ним. В частности, знаниями, представленными в этой информационной системе, являются методы и техники верификации программ. В настоящее время в систему Спектр интегрируются авторские разработки в области верификации программ [3, 4, 5, 6, 7, 8, 9].

В работе описана стандартная библиотека языка Atoment, представляющая собой множество схем выполнения. Схемы разбиты на группы в соответствии с сущностями, действия над которыми они определяют. В разделе 3 рассматриваются схемы, определяющие действия над атомами, переменными, свойствами, контекстами и исключениями. В разделе 4 представлены схемы, определяющие действия, которые создают,

удаляют и модифицируют атоменты, возвращают выборки атоментов и значений свойств атоментов. В разделе 5 описаны схемы, определяющие императивные действия, которые соответствуют императивным конструкциям в языках программирования, и их комбинацию с механизмом сопоставления с образцом. В разделе 6 рассматриваются схемы, определяющие действия, которые управляют выполнением атоментов и исполнимых файлов. В частности, в этом разделе рассматриваются схемы, определяющие действия, которые добавляют и удаляют схемы выполнения. В разделе 7 представлены схемы, определяющие действия, которые соответствуют стандартным операциям над формулами — логическим связкам и ограниченным кванторам, и логическое действие сопоставления с образцом. В разделе 8 описаны схемы, определяющие действия над целыми и вещественными числами. В разделе 9 рассматриваются схемы, определяющие действия над структурированными файлами, которые позволяют писать в файл и читать из него атоменты и состояния.

Работа частично поддержана грантом РФФИ 08-01-00899-а и интеграционным проектом РАН 2/12.

2. ПРЕДВАРИТЕЛЬНЫЕ ПОНЯТИЯ

Стандартная библиотека определяет схемы, описывающие действия над основными сущностями языка Atoment: атомами, элементами, свойствами, переменными, схемами выполнения, целыми и вещественными числами, формулами, структурированными файлами, исключениями, контекстами и состояниями, а также императивные действия и сопоставление с образцом. Пусть `THIS` обозначает действие, удовлетворяющее схеме выполнения, `ct` — текущий контекст.

Для описания семантики действий используется псевдокод. Программы на псевдокоде строятся из элементарных действий с помощью операторов композиции.

Элементарных действий шесть:

- действие `возвратить U` завершает программу, возвращая значение `U`;
- действие `A.P ::= B` изменяет состояние `st` на `st′`, где `st′(P, A) = B` для пары `(P, A)` и `st′(V) = st(V)` для любой другой пары `V`;
- действие `A.P := B` аналогично действию `A.P ::= B`, за исключе-

нием двух случаев:
  - если `A` — исключение, то действие ничего не делает и возвращает `A`;
  - если `A` — не исключение, `B` — исключение, то действие ничего не делает и возвращает `B`;
- действие `X ::= U` определяет переменную `X` и присваивает ей значение `U`.
- действие `X := U` аналогично действию `X ::= U` за исключением следующего случая: если `U` — исключение, то действие ничего не делает и возвращает `U`;
- действие **`имя с параметрами`** выполняет действие с именем **`имя с параметрами`**. Это действие является аналогом вызова функции.

Оператор последовательной композиции имеет вид:

`действие 1; действие 2; ...`

Он выполняет действия **`действие 1`**, **`действие 2`**, ... в порядке их следования.

Условный оператор имеет вид:

```
условие 1⟶ действие 1;
условие 2⟶ действие 2;
 ...
```

Он проверяет условия **`условие 1`**, **`условие 2`**, ... и выполняет первое действие, для которого выполнено соответствующее условие. Если ни одно из условий невыполнено, то оператор возвращает `void`. Если при проверке некторого условия возвращается исключение, то последующие проверки не выполняются, и действие возвращает это исключение. Действие **`действие i`** может быть последовательной композицией.

Составной оператор определяется круглыми скобками:

`(действие 1; действие 2; ...)`

Оператор цикла имеет вид:

`для i от 1 до n выполнить: действие(i)`

Он выполняет последовательность действий **`действие(1)`**; ...; **`действие(n)`**.

Действия могут помечаться меткой:

`L: действие`

Оператор **`перейти к L`** выполняет переход к действию, помеченному меткой `L`.

Комментарии записываются как

```
// комментарий
```

Пусть A, B — атоменты, P — свойство. Пусть A.P обозначает st(P, A). Пусть A = (P_1:B_1 ... P_n:B_n [Q_1] ... [Q_m]) означает, что A имеет порядковые и меточные свойства P_1, ..., P_n со значениями B_1, ..., B_n и логические свойства [Q_1], ..., [Q_m] и не имеет никаких других свойств.

Пусть new:(A_1:B_1 ... A_n:B_n) означает создать новый элемент D со свойствами A_1, ..., A_n со значениями B_1, ..., B_n и возвратить его, new:() — создать новый элемент D без свойств и возвратить его.

3. ДЕЙСТВИЯ НАД АТОМАМИ, ПЕРЕМЕННЫМИ, СВОЙСТВАМИ, КОНТЕКСТАМИ И ИСКЛЮЧЕНИЯМИ

В этом разделе рассматриваются схемы, определяющие действия над атомами, переменными, свойствами, контекстами и исключениями.

3.1. Действия над атомами

Схема быть атомом имеет вид ifpattern:(A.[atom]) var:A и семантику

```
A — атом⟶ возвратить true;
иначе⟶ возвратить void
```

3.2. Действия над переменными

Схема означить переменную имеет две подсхемы.

Схема ifpattern:(A:=B) var:($A B) имеет семантику

```
(B = void⟶
  (x — доступная переменная с именем A⟶ удалить x);
 B — чистое значение⟶
  (x — доступная переменная с именем A⟶ x.value := B;
   иначе⟶ new:([variable] context:ct value:B name:A)));
возвратить void
```

Определение переменной, доступной по имени в контексте, дано в [1].

Замечание. Специальная схема для получения значения переменной не нужна, так как получить значение переменной x можно через действие x.value, а получить значение переменной по ее имени A можно с помощью действия выполнить атомент A, описанного в [1].

Пример. Пусть x — имя переменной. Действие `x := 0` присваивает переменной с именем x значение 0 и возвращает void. Действие x возвращает значение 0 переменной с именем x. Действие `x := void` удаляет переменную с именем x и возвращает void. Пусть `E = (exception: "некоторое исключение")`. Действие `x := E` ничего не делает и возвращает E.

Пример. Действие `x:=new:([светофор])` возвращает новый светофор D (элемент с логическим свойством `[светофор]`) и присваивает значение D переменной с именем x. Действие x возвращает D.

Схема `ifpattern:(A::=B) var:($A $B)` имеет семантику

```
B′ ::= выполнить B;
(B′ = void⟶
  (x — доступная переменная с именем A⟶ удалить x);
 иначе⟶
  (x — доступная переменная с именем A⟶ x.value ::= B′;
   иначе⟶ new:([variable] context:ct value:B′ name:A)));
возвратить void
```

Пример. Пусть x — имя переменной. Действие `x ::= 0` присваивает переменной с именем x значение 0 и возвращает void. Действие x возвращает значение 0 переменной с именем x. Действие `x ::= void` удаляет переменную с именем x и возвращает void. Пусть `E = (exception: "некоторое исключение")`. Действие `x ::= E` присваивает переменной с именем x значение E и возвращает void.

3.3. Действия над свойствами

Схема `получить значение свойства` имеет вид `ifpattern:(A.B) [assoc] var:(A B)` и семантику

```
возвратить A.B.
```

Схема `присвоить значение свойству` имеет две подсхемы.

Схема `ifpattern:(A.B:=C) var:(A B C)` имеет семантику

```
A.B := C
```

Схема `ifpattern:(A.B::=C) var:(A B $C)` имеет семантику

```
C′ ::= выполнить C; A.B ::= C′
```

Схема `получить число свойств` имеет 4 подсхемы. Схема `ifpattern:(A.propnum) var:A` возвращает общее число свойств элемента A. Схема `ifpattern:(A.lognum) var:A` возвращает число логических

свойств элемента A. Схема ifpattern:(A.labnum) var:A возвращает число меточных свойств элемента A. Схема ifpattern:(A.ordnum) var:A возвращает число порядковых свойств элемента A.

Схема получить список свойств имеет 3 подсхемы.

Пусть u_1 < ... < u_n — порядковые свойства A, v_1, ..., v_m — меточные свойства A, w_1, ..., w_k — логические свойства A.

Схема ifpattern:(A.[ordlist]) var:A возвращает new:(1:u_1 ... n:u_n).

Схема ifpattern:(A.[lablist]) var:A возвращает new:(1:v_1 ... m:v_m).

Схема ifpattern:(A.[loglist]) var:A возвращает new:(1:w_1 ... 1:w_k).

Схема удалить контекст имеет вид ifpattern:(deletecontext:A) whereafter:A.[context] var:A и семантику

```
удалить контекст A
```

Схема породить контекст имеет вид ifpattern:(newcontext) и семантику

```
породить контекст
```

3.4. Действия над исключениями

Язык Atoment имеет встроенный механизм обработки исключений.

Схема быть исключением имеет вид ifpattern:(A.[is exception]) var:A и семантику

```
A — исключение⟶ возвратить true;
иначе⟶ возвратить void
```

Схема поймать исключение имеет вид ifpattern:(catch:A where:B var:C then:D else:E) [macro] var:($A $B $C $D $E) и семантику

```
(сопоставить ct.value с A where:B var:C) возвращает
 последовательность F от 1 до n,
 X — последовательность параметров образца A where:B var:C⟶
  (ct′ := породить контекст; ct′.parent := ct;
   ct′.[inheritvar] := true; ct′.[inherit] := true;
   G1 := new:([variable] context:ct′ name:X.1 value:F.1); ...;
   Gn := new:([variable] context:ct′ name:X.n value:F.n);
   ((выполнить D в ct′) возвращает H⟶
     (удалить ct′; возвратить H)));
```

```
иначе⟶
 (THIS имеет свойство else⟶
   (ct.value := void;
    ((выполнить E в ct′) возвращает H⟶
      (удалить ct′; возвратить H)));
  иначе⟶ возвратить ct.value)
```

Замечание. Действия `породить исключение` нет, поскольку любое действие может возвращать элемент, который является исключением.

4. ДЕЙСТВИЯ НАД АТОМЕНТАМИ

В этом разделе рассматриваются схемы, определяющие действия, которые создают, удаляют и модифицируют атоменты, возвращают выборки атоментов и значений свойств атоментов.

4.1. Порождение атоментов

Схема `породить элемент` имеет вид `ifpattern:(new:A where:B var:C) var:($A $B $C)` и семантику

```
D := new:();
модифицировать D относительно A where:B var:C;
возвратить D
```

Схема `породить элементы` имеет вид `ifpattern:(newall:A where:B var:C [void]) var:($A $B $C)` и семантику

```
V_1, ..., V_n — все последовательности значений параметров
образца A where:B var:C, для которых B возвращает чистое
значение, X — последовательность параметров этого образца⟶
 (THIS имеет свойство [void]⟶
   для i от 1 до n выполнить:
   (D := new:();
    модифицировать D относительно
     (заменить параметры из X в A на V_i);
    возвратить void);
  иначе⟶
   (L := new:();
    для i от 1 до n выполнить:
    (D := new:();
```

```
 модифицировать D относительно
  (заменить параметры из X в A на V_i);
 L.i := D);
возвратить L))
```

Пример. Действие `newall:([отрезок] длина:m) where:(m.[integer] and 1<=m and m<=10)` создает отрезки с целочисленными длинами от 1 до 10.

4.2. Удаление атоментов

Схема `удалить элемент` имеет вид ifpattern:(delete:A) var:A и семантику

```
A — элемент⟶ удалить элемент A
```

Пример. Пусть `x` — имя переменной. Пусть элемент `E` является значением этой переменной. Действие `delete:x` удаляет элемент `E` и возвращает `void`.

Пример. Пусть `x` — имя переменной. Пусть атом `A` является значением этой переменной. Действие `delete:x` ничего не делает и возвращает `void`.

Схема `удалить элементы` имеет вид `ifpattern:(deleteall:A where:B var:C) var:($A $B $C)` и семантику

```
(E1, ..., En — все актуальные элементы такие, что
 (сопоставить Ei с A where:B var:C) возвращает чистое
 значение⟶
  для i от 1 до n выполнить: удалить элемент Ei);
возвратить void
```

Пример. Действие `deleteall:([круг])` удаляет все круги.

Пример. Действие `deleteall:([фигура] длина:*)` удаляет все фигуры, имеющие длину.

Пример. Действие `deleteall:([фигура] длина:10 [!])` удаляет все фигуры, которые имеют длину, равную 10, и не имеют других свойств кроме длины.

Пример. Действие `deleteall:([круг] радиус:m) where:m>10` удаляет все круги радиуса, большего чем 10.

4.3. Выборка атоментов

Схема выбрать атомент имеет вид ifpattern:(get:A where:B var:C) var:($A $B $C) и семантику

```
(сопоставить E с A where:B var:C) возвращает чистое
 значение⟶ возвратить E;
иначе⟶ возвратить void
```

Пример. Действие get:([круг]) возвращает круг.

Пример. Действие get:([фигура] длина:*) возвращает фигуру, имеющую длину.

Пример. Действие get:([фигура] длина:* [!]) возвращает фигуру, имеющую длину и не имеющую других свойств кроме длины.

Пример. Действие get:([фигура] длина:10 ~ширина) возвращает фигуру, которая имеет длину 10 и не имеет ширины.

Схема выбрать атоменты имеет вид ifpattern:(getall:A where:B var:C) var:($A $B $C) и семантику

```
E1, ..., En — все актуальные элементы такие, что
(сопоставить Ei с A where:B var:C) возвращает чистое
значение⟶
 (n > 0⟶
   (D := new:();
    для i от 1 до n выполнить: D.i := Ei;
    возвратить D);
  иначе⟶ возвратить void)
```

Пример. Действие getall:([фигура]) возвращает все круги.

Пример. Действие getall:([фигура] длина:*) возвращает все фигуры, имеющие длину.

Пример. Действие getall:([фигура] длина:* [!]) возвращает все фигуры, имеющие длину и не имеющие других свойств кроме длины.

Пример. Действие getall:([фигура] длина:10 ~ширина) возвращает все фигуры, которые имеют длину 10 и не имеют ширины.

4.4. Модификация атоментов

Схема модифицировать элемент имеет вид ifpattern:(modify:A match:B where:C var:D) var:(A $B $C $D) и семантику

`модифицировать A относительно B where:C var:D`

Пример. Действие `modify:x match:(цвет:зеленый вкус:сладкий)` добавляет к элементу `x` свойства `цвет` и `вкус` со значениями `зеленый` и `сладкий`.

Пример. Действие `x.цвет` возвращает значение `зеленый` свойства `цвет`.

Пример. Действие `modify:x match:(~цвет вкус:горький)` удаляет свойство `цвет` и изменяет значение свойства `вкус` на `горький`.

Пример. Действие `x.цвет` возвращает `void`.

5. ИМПЕРАТИВНЫЕ ДЕЙСТВИЯ И СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ

В этом разделе рассматриваются схемы, определяющие императивные действия, которые соответствуют императивным конструкциям в языках программирования, и их комбинацию с механизмом сопоставления с образцом.

5.1. Последовательное выполнение

Схема `выполнить последовательно` имеет вид `ifpattern:(A;B) [assoc] [macro] var:($A $B)` и семантику

`выполнить A; выполнить B`

5.2. Итерация

Схема `выполнить итерацию` имеет 3 подсхемы.

Схема `ifpattern:(foreach:x [index] [rev] in:A do:B from:m to:n) [macro] var:($x $A $B m n)` имеет семантику

```
A не имеет порядковых свойств⟶ возвратить void;
A имеет порядковые свойства от m₁ до n₁⟶
 ((THIS имеет свойство from⟶ m₂ := A.from;
   иначе⟶ m₂ := m₁);
  (THIS имеет свойство to⟶ n₂ := A.to;
   иначе⟶ n₂ := n₁);
  (THIS не имеет свойства [rev]⟶
    (m′ := max(m₂, m₁); n′ := min(n₂, n₁));
   иначе⟶ (m′ := max(m₂, n₁); n′ := min(n₂, m₁)));
```

```
  ct′ := породить контекст; ct′.parent := ct;
  ct′.[inheritvar] := true; ct′.[inherit] := true;
  var := new([variable] context:ct′ name:x value:null);
  для i от m′ до n′ выполнить:
   ((THIS не имеет свойства [index]⟶ var.value := A.i;
     иначе⟶ var.value := i);
    val := выполнить B в ct′);
  удалить ct′; возвратить val
```

Схема `ifpattern:(while:A do:B) [macro] var:($A $B)` имеет семантику

```
anIterationHasBeenExecuted := false;
1: C := выполнить A;
(C — чистое значение⟶
  (D := выполнить B;
   (D — чистое значение⟶
     (anIterationHasBeenExecuted := true; перейти к 1)));
 иначе⟶
  (anIterationHasBeenExecuted = false⟶ возвратить void;
   иначе⟶ возвратить D))
```

Схема `ifpattern:(while:A match:B where:C var:D do:E) var:(A B C D E)` имеет семантику

```
anIterationHasBeenExecuted := false;
(1: (сопоставить A с B where:C var:D) возвращает
 последовательность V от 1 до n,
 X — последовательность параметров образца B where:C var:D⟶
  (anIterationHasBeenExecuted := true;
   ct′ := породить контекст;
   ct′.parent := ct; ct′.[inheritvar] := true;
   ct′.[inherit] := true;
   new([variable] context:ct′ name:X.1 value:V.1); ...;
   new([variable] context:ct′ name:X.n value:V.n);
   W ::= выполнить E в ct′;
   (W — исключение⟶ (удалить ct′; возвратить C);
    иначе⟶ (удалить ct′; перейти к 1))));
(anIterationHasBeenExecuted = false⟶ возвратить void;
 иначе⟶ возвратить W)
```

5.3. Условное выполнение

Схема выполнить при условии имеет 2 подсхемы.

Схема `ifpattern:(if:A then:B else:C) [macro] var:(A $B $C)` имеет семантику

```
A ≠ void⟶ выполнить B;
иначе⟶ (THIS имеет свойство else⟶ выполнить C)
```

Эта схема имеет синтаксическое расширение

`if:A then:B elseif:`C_1 `then:`D_1 `...` `elseif:`C_n `then:`D_n `else:E`

которое сводится к выражению

`if:A then:B elseiflist:(1:`C_1 `...` `n:`C_n`)`
`thenlist:(1:`D_1 `...` `n:`D_n`) else:E.`

Выполнение этого расширения эквивалентно выполнению

```
if:A then:B else:(if:C_1 then:D_1 else:(if:C_2 then:D_2 ...
else:(if:C_n then:D_n else:E) ... )).
```

Схема `ifpattern:(if:A match:B where:C var:D then:E else:F) [macro] var:(A $B $C $D $E $F)` имеет семантику

```
(сопоставить A с B where:C var:D) возвращает
 последовательность V от 1 до n,
 X — список параметров образца B where:C var:D⟶
  (ct′ := породить контекст; ct′.parent := ct;
   ct′.[inheritvar] := true; ct′.[inherit] := true;
   new([variable] context:ct′ name:X.1 value:V.1); ...;
   new([variable] context:ct′ name:X.n value:V.n);
   W ::= выполнить E; удалить ct′; возвратить W);
иначе⟶ (THIS имеет свойство else⟶ выполнить F)
```

Эта схема имеет синтаксическое расширение

`if:A match:`B_1 `where:`C_1 `var:`D_1 `then:`E_1 `...`
`match:`B_n `where:`C_n `var:`D_n `then:`E_n `else:F`

которое сводится к выражению

`if:A matchlist:(1:(match:`B_1 `where:`C_1 `var:`D_1`) ...`
`n:(match:`B_n `where:`C_n `var:`D_n`)) thenlist: (1:`C_1 `...` `n:`C_n`) else:F`

Выполнение этого расширение эквивалентно выполнению

`if:A match:`B_1 `where:`C_1 `var:`D_1 `then:`E_1 `else:( ...`
`match:`B_n `where:`C_n `var:`D_n `then:`E_n `else:F) ...)`

Пример. Действие `if:x match:([светофор] цвет:зеленый) then: иди else:жди` эквивалентно действию `иди`, если `x` — светофор, и горит зеленый свет. Иначе оно эквивалентно действию `жди`.

6. УПРАВЛЕНИЕ ВЫПОЛНЕНИЕМ АТОМЕНТОВ И ИСПОЛНИМЫХ ФАЙЛОВ

В этом разделе рассматриваются схемы, определяющие действия, которые управляют выполнением атоментов и исполнимых файлов. В частности, рассматриваются схемы, определяющие действия, которые добавляют и удаляют схемы выполнения.

6.1. Добавление и удаление схем выполнения

Схема `добавить схему выполнения` имеет вид `ifpattern: ([local] ifpattern:A where:B whereafter:C var:D id:E [assoc] [macro] then:F) var:($A $B $C $D $E $F)`. Она порождает новую д схему выполнения, определяемую свойствами `ifpattern`, `where`, `whereafter`, `var`, `id`, `[macro]` и `then` и возвращает ее в качестве значения. Свойство `[assoc]` используется для бинарных операций и указывает, что бинарная операция левоассоциативна, т. е. `a op b op c = (a op b) op c`. Оно относится к синтаксическим свойствам. Порождаемая схема выполнения `S` добавляется в контекст, который вычисляется по следующему правилу:

```
ct′ := ct;
(действие имеет свойство [local]⟶ S.context := ct′;
 иначе⟶
  (1: ct′.[scheme localizing] ≠ void⟶ S.context := ct′;
   ct′.parent = void⟶ S.context := ct′;
   иначе⟶ (ct′ := ct′.parent; перейти к 1)))
```

Схема `удалить схему выполнения` имеет вид `ifpattern: (deletescheme:A) var:A` и семантику

```
ct′ := ct; schemeNumber := 0;
(1: контекст ct′ содержит схему B с идентификатором A⟶
  (schemeNumber := schemeNumber + 1; удалить B; перейти к 1);
 ct′.[inherit]≠void, ct′.parent≠void⟶
  (ct′:=ct′.parent; перейти к 1);
 иначе⟶ возвратить schemeNumber)
```

Замечание. Для удаления схемы выполнения, не имеющей идентификатора, используется действие `удалить элемент`.

6.2. Возвращение значения

Схема `возвратить значение` имеет 2 подсхемы. Действие, удовлетворяющее этой схеме, должно иметь предком `B.then` для некоторой схемы выполнения `B`.

Схема `ifpattern:(return:A) var:A` имеет семантику

```
ct.value := A; возвратить A
```

Схема `ifpattern:return` имеет семантику

```
ct.value := void; возвратить void
```

Пример. Действие

```
ifpattern:x.площадь whereafter:x.[фигура] var:x id:площадь
then:
(if:x
 match:([прямоугольник] длина:A ширина:B) then:(return:A*B)
 match:([круг] радиус:A) then:(return:3,1415*A*A)
 ...)
```

добавляет новую схему выполнения `E`, которая вычисляет площади фигур.

Пример. Действие `(new:([фигура] [круг] радиус:10 ""единица измерения":см)).площадь` создает новый круг радиуса 10 см и возвращает площадь этого круга.

Пример. Действие `deletescheme:площадь` удаляет схему `E`.

6.3. Выполнение атоментов

Схема `выполнить атомент` имеет вид `ifpattern:(execute:A) var:A` и семантику

```
выполнить A
```

Схема `не выполнять атомент` имеет вид `ifpattern:(noexecute:A) var:$A` и семантику

```
возвратить A
```

Пример. Действие `y := (noexecute:""напиши письмо")` сохраняет атомент `напиши письмо` в переменной с именем `y`. Действие `execute:y`

эквивалентно действию напиши письмо.

6.4. Выполнение файлов

Схема выполнить файл имеет вид ifpattern:(executefile:A) var:A и семантику

```
(A — выполнимый файл⟶ выполнить A);
возвратить void
```

7. ДЕЙСТВИЯ НАД ФОРМУЛАМИ

Любой атомент может рассматриваться как формула. Формула A истинна, если A возвращает чистое значение. Формула A ложна, если A возвращает void. В этом разделе рассматриваются схемы, определяющие действия, которые соответствуют стандартным операциям над формулами — логическим связкам и ограниченным кванторам, и логическое действие сопоставления с образцом.

7.1. Логические связки

Стандартные логические связки моделируется с помощью схем и, или, не.

Схема и имеет вид ifpattern:(A and B) [assoc] var:(A $B) и семантику

```
A — чистое значение⟶ выполнить B;
иначе⟶ возвратить void
```

Схема или имеет вид ifpattern:(A or B) [assoc] var:(A $B) и семантику

```
A = void⟶ выполнить B;
иначе⟶ возвратить A
```

Схема не имеет вид ifpattern:(not:A) [assoc] var:A и семантику

```
A — чистое значение⟶ возвратить void;
иначе⟶ возвратить true
```

7.2. Кванторы

Кванторы моделируется с помощью схем для любого и существует.

Схема для любого имеет вид ifpattern:(forall:A in:B where:C var:D formula:E [exc]) var:($A $B $C $D $E) и семантику

```
M := {};
(1: V ∉ M, (сопоставить V с B where:C var:D) возвращает
  последовательность W от 1 до n,
  X — последовательность параметров образца B where:C var:D⟶
   (ct′ := породить состояние; ct′.parent := ct;
    ct.[inheritvar] := true; ct.[inherit] := true;
    new([variable] context:ct′ name:X.1 value:W.1); ...;
    new([variable] context:ct′ name:X.n value:W.n);
    W ::= выполнить E в ct′;
    (W — чистое значение⟶
      (удалить ct′; добавить V к M; перейти к 1);
     W = void⟶ (удалить ct′; возвратить void);
     W — исключение⟶
      (THIS имеет свойство [exc]⟶
        (удалить ct′; возвратить void);
       иначе⟶ (удалить ct′; возвратить W))));
 иначе⟶ возвратить true)
```

Схема существует имеет вид ifpattern:(exist:A in:B where:C var:D formula:E [exc]) var:($A $B $C $D $E) и семантику

```
M := {};
(1: V ∉ M, (сопоставить V с B where:C var:D) возвращает
 последовательность W от 1 до n,
 X — последовательность параметров образца B where:C var:D⟶
  (ct′ := породить состояние; ct′.parent := ct;
   ct.[inheritvar] := true; ct.[inherit] := true;
   new([variable] context:ct′ name:X.1 value:W.1); ...;
   new([variable] context:ct′ name:X.n value:W.n)
   W ::= выполнить E в ct′;
   (W — чистое значение⟶ (удалить ct′; возвратить true);
    W = void⟶ удалить ct′; (добавить V к M; перейти к 1);
    W — исключение⟶
     (THIS имеет свойство [exc]⟶
       (удалить ct′; добавить V к M; перейти к 1);
      иначе⟶ (удалить ct′; возвратить W))));
 иначе⟶ возвратить void)
```

7.3. Сопоставление с образцом

Схема `сопоставить с образцом` имеет вид `ifpattern:(element:A match:B var:C where:D) var:(A $B $C $D)` и семантику

```
(сопоставить A с B var:C where:D) возвращает void⟶
 возвратить void;
иначе⟶ возвратить true
```

8. ДЕЙСТВИЯ НАД ЦЕЛЫМИ И ВЕЩЕСТВЕННЫМИ ЧИСЛАМИ

В этом разделе рассматриваются схемы, определяющие действия над целыми и вещественными числами.

8.1. Предикаты

Схема `быть числом` имеет вид `ifpattern:(A.[number]) var:A` и семантику

```
A — целое или вещественное число⟶ возвратить true;
иначе⟶ возвратить void
```

Схема `быть целым числом` имеет вид `ifpattern:(A.[integer]) var:A` и семантику

```
A — целое число⟶ возвратить true;
иначе⟶ возвратить void
```

Схема `быть вещественным числом` имеет вид `ifpattern:(A.[real]) var:A` и семантику

```
A — вещественное число⟶ возвратить true;
иначе⟶ возвратить void
```

8.2. Арифметические операции

Схема `получить сумму чисел` имеет вид `ifpattern:(A + B) [assoc] whereafter:(A.[number] and B.[number]) var:(A B)`. Она возвращает сумму чисел `A` и `B`. Если хотя бы одно из чисел — вещественное, то схема возвращает вещественное число. В противном случае схема возвращает целое число.

Схема `получить разность чисел` имеет вид `ifpattern:(A - B) [assoc] whereafter:(A.[number] and B.[number]) var:(A B)`. Она возвращает разность чисел `A` и `B`. Если хотя бы одно из чисел — вещественное, то схема возвращает вещественное число. В противном случае схема возвращает целое число.

Схема `получить произведение чисел` имеет вид `ifpattern:(A * B) [assoc] whereafter:(A.[number] and B.[number]) var:(A B)`. Она возвращает произведение чисел `A` и `B`. Если хотя бы одно из чисел — вещественное, то схема возвращает вещественное число. В противном случае схема возвращает целое число.

Схема `получить частное чисел` имеет вид `ifpattern:(A * B) [assoc] whereafter:(A.[number] and B.[number]) var:(A B)`. Она возвращает частное чисел `A` и `B`. Схема всегда возвращает вещественное число.

Схема `получить целую часть от деления` имеет вид `ifpattern:(A div B) [assoc] whereafter:(A.[integer] and B.[integer]) var:(A B)`. Она возвращает целую часть от деления `A` на `B`.

Схема `получить остаток от деления` имеет вид `ifpattern:(A mod B) [assoc] whereafter:(A.[integer] and B.[integer]) var:(A B)`. Она возвращает остаток от деления `A` на `B`.

8.3. Отношения над числами

Операции сравнения чисел определяются схемами

```
ifpattern:(A < B) [assoc]
 whereafter:(A.[number] and B.[number]) var:(A B)
ifpattern:(A <= B) [assoc]
 whereafter:(A.[number] and B.[number]) var:(A B)
ifpattern:(A > B) [assoc]
 whereafter:(A.[number] and B.[number]) var:(A B)
ifpattern:(A >= B) [assoc]
 whereafter:(A.[number] and B.[number]) var:(A B)
ifpattern:(A != B) [assoc]
 whereafter:(A.[number] and B.[number]) var:(A B)
ifpattern:(A = B) [assoc]
 whereafter:(A.[number] and B.[number]) var:(A B)
```

8.4. Математические функции над числами

Математические функции над вещественными числами определяются схемами:

```
ifpattern:(abs:A) whereafter:A.[number] var:A
ifpattern:(acos:A) whereafter:A.[number] var:A
ifpattern:(asin:A) whereafter:A.[number] var:A
ifpattern:(atan:A) whereafter:A.[number] var:A
ifpattern:(ceil:A) whereafter:A.[number] var:A
ifpattern:(cos:A) whereafter:A.[number] var:A
ifpattern:(hcos:A) whereafter:A.[number] var:A
ifpattern:(exp:A) whereafter:A.[number] var:A
ifpattern:(floor:A) whereafter:A.[number] var:A
ifpattern:(log:A) whereafter:A.[number] var:A
ifpattern:(log10:A) whereafter:A.[number] var:A
ifpattern:(sin:A) whereafter:A.[number] var:A
ifpattern:(hsin:A) whereafter:A.[number] var:A
ifpattern:(sqrt:A) whereafter:A.[number] var:A
ifpattern:(tan:A) whereafter:A.[number] var:A
ifpattern:(htan:A) whereafter:A.[number] var:A
ifpattern:(number:A power:B) whereafter:(A.[number] and
 B.[number]) var:(A B)
```

Свойства `abs`, `acos`, `asin`, `atan`, `ceil`, `cos`, `hcos`, `exp`, `floor`, `log`, `log10`, `sin`, `hsin`, `sqrt`, `tan`, `htan` используются, чтобы вычислять абсолютное значение `A`, арккосинус `A`, арксинус `A`, арктангенс `A`, наименьшее целое, большее или равное `A`, косинус `A`, гиперболический косинус `A`, экспоненту от `A`, наибольшее целое, меньшее или равное `A`, натуральный логарифм `A`, десятичный логарифм `A`, синус `A`, гиперболический синус `A`, квадратный корень из `A`, тангенс `A`, гиперболический тангенс `A`.

Если аргумент — целое число, то оно приводится к соответствующему вещественному числу. Действие `(number:A power:B)` вычисляет `A` в степени `B`. Для целых чисел `A` и `B` оно возвращает целое число.

9. ДЕЙСТВИЯ НАД СТРУКТУРИРОВАННЫМИ ФАЙЛАМИ

Структурированный файл — это файл, содержимое которого является выражением. В этом разделе рассматриваются схемы, определяющие действия над структурированными файлами, которые позволяют

писать в файл и читать из него атоменты и состояния.

9.1. Открытие и закрытие файла

Схема открыть файл имеет вид ifpattern:(openfile:A [read] [write]) where:A.[atom] var:A и семантику

```
(нет доступа к файлу с именем A⟶
 возвратить new:(exception:""There is no access to file"
 value:THIS));
(файл с путем A не существует⟶
 (THIS имеет свойство [write]⟶ создать файл с именем A;
  иначе⟶
   возвратить (exception:""There is no file with this name"
    value:THIS)));
F := new:([file] path:A);
(THIS имеет свойство [write]⟶ F.[write] := true);
(THIS имеет свойство [read]⟶ F.[read] := true);
означить скрытые служебные свойства F;
возвратить F
```

Схема возвращает новый элемент с логическим свойством [file], меточным свойством path, специфицирующим путь к файлу, логическими свойствами [read] и [write], специфицирующими доступ к файлу, и другими скрытыми служебными свойствами, определяющими механизм взаимодействия этого элемента со структурированным файлом. Этотэлемент называется дескриптором указанного файла.

Схема закрыть файл имеет вид ifpattern:(closefile:A) where:A.[file] var:A и семантику

```
выполнить закрытие файла с путем A.path;
удалить элемент A
```

9.2. Сериализация атоментов

Схема записать атомент в файл имеет вид ifpattern:(write:A to:B [readability] [tree]) where:B.[file] var:(A B) и семантику

```
THIS имеет свойство [tree]⟶
  (A — дерево⟶
    (D := сериализовать дерево A;
     (THIS не имеет свойства [readability]⟶
```

```
      содержимое файла с путем A.path := D;
     иначе⟶
      содержимое файла с путем A.path :=
       преобразовать выражение D к более высокому порядку));
  иначе⟶
  возвратить new:(exception:""There is no tree" value:THIS));
иначе⟶ D := сериализовать A
```

Если THIS имеет свойство `[readability]`, результирующее выражение представляется в удобном для чтения виде. В противном случае, `A` транслируется в выражение порядка `0`.

Атомент `A` называется деревом, если

- `A` — атом или
- `A` — элемент и
  - любое меточное свойство атомента `A` имеет вид $\{P_1 \ \ldots \ P_n\}$, где P_i — атомы;
  - любое логическое свойство атомента `A` имеет вид $\{Q_1 \ \ldots \ Q_m\}$, где Q_j — атомы;
  - для любого меточного или порядкового свойства `P` атомента `A` атомент `A.P` является деревом.

Пусть `A` — дерево. Действие `сериализовать дерево A` имеет семантику:

```
A — атом⟶ возвратить A;
A имеет меточные и порядковые свойства P1, ..., Pn и
логические свойства Q1, ..., Qm⟶
 (A1 := сериализовать дерево A.P1;
  ...;
  An := сериализовать дерево A.Pn;
  возвратить (A1 ... An Q1 ... Qm))
```

Пример. Пусть `A := (U sin(x) V+W)`, `B` — дескриптор файла `""test"`. Действие `(write:A to:B [tree])` записывает выражение `(1:U (applyfun:sin 1:x) (applyfun:+ 1:V 2:W))` в файл `""test"`. Действие `(write:A to:B [tree] [readability])` записывает выражение `(U sin(x) V+W)` в файл `""test"`.

Пусть `A` — атомент. Действие `сериализовать A` имеет семантику:

```
ElemswithUname := new:(); ElemswithUid := new:();
NotSerializedElemSet := {}; сериализовать A в Env
```

Окружение Env, в котором сериализуется атомент, состоит из трех глобальных переменных ElemswithId, ElemswithUid и ElemswithoutId. Переменная ElemswithUname определяет элементы, которые пройдены процедурой сериализации, и которым сопоставлены новые уникальные имена: ElemswithUname.B = A тогда и только тогда, когда атомент A пройден процедурой сериализации, A не имеет уникального идентификатора и B — уникальное имя для A, присвоенное процедурой сериализации. Переменная ElemswithId определяет элементы, которые пройдены процедуру сериализации и имеют уникальные идентификаторы: ElemswithId.B = A тогда и только тогда, когда атомент A пройден процедурой сериализации и A.uid = B. Переменная NotSerializedElemSet определяет множество элементов, которые не пройдены процедурой сериализации: A ∈ NotSerializedElemSet тогда и только тогда, когда атомент A не пройден процедурой сериализации и встретился в свойствах атомента, который пройден процедурой сериализации.

Действие сериализовать A в Env имеет семантику:

```
A = {A_1 ... A_n}⟶
 (для i от 1 до n выполнить: A_i' := сериализовать A_i в Env;
  возвратить {A_1' ... A_n'});
A = [A_1 ... A_n]⟶
 (для i от 1 до n выполнить: A_i' := сериализовать A_i в Env;
  возвратить [A_1' ... A_n']);
A — атом⟶ возвратить A;
ElemswithUname.B = A⟶ возвратить (elem:B);
ElemswithUid.B = A⟶ возвратить (elem:B);
A = (P_1:A_1 ... P_n:A_n Q_1 ... Q_m)⟶
 (добавить A в NotSerializedElemSet;
  (A.uid ≠ void⟶
    (id := A.uid; ElemswithUid.id := A; idprop := uid);
   иначе⟶
    (id := породить атом;
     ElemswithUname.id := A; idprop := name));
  для i от 1 до n выполнить:
  (P_i' := сериализовать P_i в Env;
   A_i' := сериализовать A_i в Env);
  для i от 1 до m выполнить: Q_i' := сериализовать Q_i в Env;
  возвратить (idprop:id cont:(P_1':A_1' ... P_n':A_n' Q_1' ... Q_m')))
```

Пример. Пусть текущее состояние содержит актуальные элементы A, B, C со следующими свойствами:

```
A = (applyfun:f 1:B)
B = (applyfun:g 1:B 2:C)
C = (prop:B uid:id).
```

Пусть U — дескриптор файла ""test". Тогда действие write:A to:U записывает следующее содержимое в файл ""test":

```
(name:A1 cont:(applyfun:f 1:(elem:B1)))
(name:B1 cont:(applyfun:g 1:(elem:B1) 2:(elem:ui)))
(uid:ui cont:(prop:(elem:B1)))
```

где A1 и B1 — новые уникальные атомы, задающие имена элементам A и B, соответственно. Так как элемент C имеет уникальный идентификатор, задаваемый свойством uid, то для него новое уникальное имя не порождается, а используется в качестве имени уникальный идентификатор ui.

9.3. Десериализация атоментов

Схема прочитать атомент из файла имеет вид ifpattern:([atom] [serialized] read:A) where:A.[file] var:A и семантику

```
THIS имеет свойство [atom]⟶
 возвратить чистый атом, соответствующий содержимому файла
 с путем A.path;
THIS имеет свойство [serialized]⟶
 (содержимое файла с путем A.path является сериализованным
   выражением B⟶ десериализовать B;
  иначе⟶
   возвратить new:(exception:""File does not contain
    serialized expression" value:THIS));
содержимое файла с путем A.path является выражением B⟶
 (C := транслировать B; возвратить C);
иначе⟶
 возвратить new:(exception:""File does not contain expression"
  value:THIS)
```

Пусть A — сериализованное выражение. Действие десериализовать A имеет семантику:

```
A — последовательность от 1 до n⟶
```

```
(для i от 1 до n выполнить:
  (A.i.name ≠ void⟶ A.i.elem := породить элемент;
   B.uid = A.i.uid⟶ A.i.elem := B);
 для i от 1 до n выполнить:
  (A.i.cont = (P_1:A_1 ... P_k:A_k Q_1 ... Q_m)⟶
    (для j от 1 до k выполнить:
      (P_j = {C_1 ... C_r}⟶
        (для s от 1 до r выполнить:
          (C_s = (elem:D), A.v.name = D⟶ C_s' := A.v.elem;
           C_s = (elem:D), A.v.uid = D⟶ C_s' := A.v.elem;
           C_s — атом⟶ C_s' := C_s);
         P_j' := {C_1' ... C_r'}));
     для j от 1 до k выполнить:
      (C_s = (name:D), A.v.name = D⟶ A_j' := A.v.elem;
       C_s = (uid:D), A.v.uid = D⟶ A_j' := A.v.elem;
       C_s — атом⟶ A_j' := A_j);
     для j от 1 до k выполнить: A.i.elem.P_j' := A_j';
     для j от 1 до m выполнить:
      (Q_j = [C_1 ... C_r]⟶
        (для s от 1 до r выполнить:
          (C_s = (name:D), A.v.name = D⟶ C_s' := A.v.elem;
           C_s = (uid:D), A.v.uid = D⟶ C_s' := A.v.elem;
           C_s — атом⟶ C_s' := C_s);
         Q_j' := [C_1' ... C_r']);
     для j от 1 до m выполнить: A.i.elem.Q_j' := true))))
```

Пример. Пусть `A` — дескриптор файла `""Verification strategy"`. Действие `x := (read:A)` создает элемент по содержимому файла `""Verification strategy"` и сохраняет этот элемент в переменной с именем `x`. Действие `eval:x` выполняет стратегию верификации из этой переменной.

Пример. Пусть содержимое файла `""test"` имеет вид:

```
(green yellow red)
```

Пусть `A` — дескриптор этого файла. Действие `read:A` возвращает новый элемент вида `(1:green 2:yellow 3:red)`. Действие `([atom] read:A)` возвращает чистый атом `\(green\ yellow\ red\)`.

Пример. Пусть текущее состояние содержит актуальные элементы `A`, `B`, `C` со следующими свойствами:

```
A = (applyfun:f 1:B)
B = (applyfun:g 1:B 2:C)
C = (prop:B uid:id).
```

Пусть `U` — дескриптор файла `""test"`. Содержимое файла `""test"` имеет вид:

```
(name:A1 cont:(applyfun:f 1:(elem:B1)))
(name:B1 cont:(applyfun:g 1:(elem:B1) 2:(elem:ui)))
(uid:ui cont:(prop1:abc))
```

Тогда действие `read:A to:U` записывает следующее содержимое в файл `""test"` меняет текущее состояние на следующее:

```
A = (applyfun:f 1:B)
B = (applyfun:g 1:B 2:C)
C = (prop:B uid:id prop1:abc)
A2 = (applyfun:f 1:B2)
B2 = (applyfun:g 1:B2 2:C)
```

где `A2` и `B2` — новые актуальные элементы, порождаемые по именам `A1` и `B1`.

9.4. Сериализация и десериализация состояний

Схема `записать текущее состояние в файл` имеет вид `ifpattern: (writestate:A) where:A.[file]`. Семантика этой схемы аналогична семантики схемы `записать текущее состояние в файл` и выходной файл имеет такой же формат за исключением того, что во-первых, сериализуются все актуальные элементы текущего состояния и, во-вторых, сериализуются непредопределенные свойства атомов.

Формат выражения для атомов имеет вид `(atom:A cont:B)`, где `A` — атом, `B` — список непредопредленных свойств атома `A`, задаваемый так же, как и для элементов.

Схема `прочитать текущее состояние из файла` имеет вид `ifpattern:(readstate:A) where:A.[file] var:A`. Семантика этой схемы аналогична семантике схемы `прочитать атомент из файла`.

Пример. Пусть текущее состояние содержит актуальные элементы `A`, `B`, `C` и атом `at` со следующими свойствами:

```
A = (x:B y:at z:A)
B = (uid:ui z:A)
C = ()
```

```
at = (prop:A)
```

и атом `at` с непредпоределенным свойством Пусть `U` — дескриптор файла `""test"`. Тогда действие `(writestate:A)` записывает следующее содержимое в файл `""test"`:

```
(name:A1 cont:(x:(elem:ui) y:at z:(elem:A1)))
(uid:ui cont:(z:(elem:A1))
(uid:C1 cont:())
(atom:at cont(prop:(elem:A1)))
```

где `A1` и `C1` — новые уникальные атомы, задающие имена элементам `A` и `C`, соответственно. Так как элемент `B` имеет уникальный идентификатор, задаваемый свойством `uid`, то для него новое уникальное имя не порождается, а используется в качестве имени уникальный идентификатор `ui`.

10. ЗАКЛЮЧЕНИЕ

В работе описана стандартная библиотека языка Atoment, представляющая собой множество схем выполнения. Представлены схемы выполнения, определяющие действия над атомами, элементами, свойствами, переменными, схемами выполнения, целыми и вещественными числами, формулами, структурированными файлами, исполнимыми файлами, исключениями, контекстами и состояниями, а также императивные действия и сопоставление с образцом. Определена семантика этих схем выполнения. Схемы снабжены примерами их применения. Мы планируем использовать язык Atoment в рамках системы Спектр для разработки программных моделей широко используемых языков программирования (Java, C/C++, C# и т. п.), формальных выполнимых спецификаций этих языков на базе операционно-онтологического подхода [10], методов спецификации и верификации программных моделей и систем.

СПИСОК ЛИТЕРАТУРЫ

1. Ануреев И.С. Язык Atoment: синтаксис и семантика. — Новосибирск, 2010. — 40 с. — (Препр./РАН. Сиб. отд-ние. ИСИ; № 157).
2. Непомнящий В.А., Ануреев И.С., Атучин М.М., Марьясов И.В., Петров А.А., Промский А.В. Система анализа и верификации С-программ СПЕКТР-2 // Моделирование и анализ информационных систем. — 2010. — № 4. — (В печати).

3. Шилов Н.В., Анореев И.С., Бодин Е.В. О генерации условий корректности для императивных программ // Программирование. — 2008. — № 6. — С. 307–321.
4. Непомнящий В.А., Анореев И.С., Михайлов И.Н., Промский А.В. На пути к верификации С программ. Язык C-light и его формальная семантика // Программирование. — 2002. — № 6. — С. 314–323.
5. Непомнящий В.А., Анореев И.С., Промский А.В. На пути к верификации С программ. Аксиоматическая семантика языка C-kernel // Программирование. — 2003. — № 6. — С. 338–350.
6. Непомнящий В.А., Анореев И.С., Промский А.В., Дубрановский И.В. На пути к верификации C# программ: трехуровневый подход // Программирование. — 2006. — № 4. — С. 190–202.
7. Anureev I.S. A three-stage method of C program verification // Joint NCC&IIS Bulletin. Ser. Comput. Sci. — 2008. — Vol. 28 — P. 1–30.
8. Анореев И.С., Марьясов И.В., Непомнящий В.А. Верификация С-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. — 2010. — № 3. — С. 1–23.
9. Анореев И.С. Метод элиминации структур данных, основанный на системах переписывания формул // Программирование. — 1999. — № 4. — С. 5–15.
10. Анореев И.С. Операционно-онтологический подход к формальной спецификации языков программирования // Программирование. — 2009. — № 1. — С. 35–42.

И.С. Ануреев

ЯЗЫК ATOMENT: СТАНДАРТНАЯ БИБЛИОТЕКА

Препринт
158

Рукопись поступила в редакцию 11.11.2010
Рецензент А.В. Промский
Редактор Т. М. Бульонкова

Подписано в печать 24.12.2010
Формат бумаги 60×84 1/16 Объем 1,8 уч.-изд.л., 2,0 п.л.
Тираж 60 экз.

Центр оперативной печати "Оригинал 2"
г.Бердск, ул. Островского, 55, оф. 02, тел. (383) 214 45 35