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

И.С. Ануреев

ЯЗЫК ATOMENT: СИНТАКСИС И СЕМАНТИКА

Препринт
157

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 представлены синтаксис и семантика языка Atoment.

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

I.S. Anureev

THE ATOMENT LANGUAGE: SYNTAX AND SEMANTICS

Preprint
157

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 syntax and semantics of the Atoment language are described.

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 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 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 [1], 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 [2, 3, 4, 5, 6, 7, 8].

Работа имеет следующую структуру. В разделе 2 сформулированы требования, которым должен удовлетворять предметно-ориентированный язык разработки методов верификации программ, и показано, какие составляющие языка Atoment обеспечивают выполнение этих требований. В разделе 3 описаны предварительные понятия, которые используются в последующем изложении. В частности, этот раздел включает

нотацию для описания семантики конструкций языка Atoment, которая представляет собой некоторый вариант псевдокода. Основные семантические сущности языка Atoment представлены в разделах 4-9. Синтаксические конструкции этого языка и алгоритмы их трансляции в семантические сущности описаны в разделах 10-13.

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

2. ТРЕБОВАНИЯ К ПРЕДМЕТНО-ОРИЕНТИРОВАННОМУ ЯЗЫКУ РАЗРАБОТКИ МЕТОДОВ ВЕРИФИКАЦИИ ПРОГРАММ

В этом разделе мы определим требования, которым должен удовлетворять предметно-ориентированный язык разработки методов верификации программ (далее VOL — Verification-Oriented Language). Достаточно большое число практических методов анализа и верификации программ укладываются в следующую схему их использования:

1. Получить метод верификации, программу на целевом языке программирования и аннотацию, которая описывает анализируемое свойство этой программы.
2. Транслировать аннотированную программу в графовую модель.
3. Применить трансформации на графах, которые реализуют метод.
4. Возвратить интерпретацию результирующего графа.

Методы статического анализа, методы операционной и аксиоматической семантики, методы проверки на моделях, конечно-автоматные методы, бисимуляционные методы и методы трансформационной семантики хорошо укладываются в эту схему.

При дальнейшем изложении мы будем называть графовую модель аннотированной программы или ее (графовой модели) текстовое представление на языке VOL программной моделью.

Трансляция аннотированной программы в программную модель должна быть один к одному, чтобы обеспечить корректность трансляции на синтаксическом уровне и легкость обращения с этой программной моделью. Это требует гибкого синтаксиса языка VOL.

Более детальный анализ трансформаций на графах показывает, что следующие задачи часто решаются при описании этих трансформаций: задачи комбинирования трансформаций (например, последовательная

комбинация), локальное сопоставление с образцом на графах, локальная модификация графа относительно образца.

Локальное сопоставление с образцом на направленных графах — это задача проверки, сопоставляется ли некоторая вершина графа с образцом или нет. Образец зависит от предков и потомков этой вершины указанной глубины. Локальная модификация графа относительно образца — это задача модификации вершины графа (и, может быть, ее потомков) в соответствии с образцом.

Современный язык программирования имеет сложную концептуальную структуру, включающую сотни понятий. Поэтому спецификация метода верификации на языке VOL должна обеспечивать категоризацию сущностей целевого языка программирования так же, как создание и удаление экземпляров этих категорий (понятий).

Суммируем требования, которым язык VOL должен удовлетворять. Он должен

1) описывать программы, аннотацию и другие данные в графовой форме;
2) специфицировать задачи комбинирования трансформаций;
3) специфицировать локальное сопоставление с образцом на графах;
4) специфицировать локальную модификацию графа относительно образца;
5) иметь гибкий синтаксис для представления один к одному синтаксических конструкций целевого языка программирования;
6) разбивать на категории сущности целевого языка программирования;
7) обеспечивать создание и удаление экземпляров этих категорий.

Предлагаемый язык Atoment удовлетворяет всем этим требованиям:

1. На этом языке вершины графов представляются атоментами, а помеченные направленные дуги — меточными и порядковыми свойствами атоментов (Раздел 4).
2. Задачи комбинирования трансформаций представляются контекстами (Раздел 5), схемами выполнения (Раздел 8) и императивными действиями стандартной библиотеки [9] языка Atoment.
3. Локальное сопоставление с образцом на графах специфицируется образцами поиска (Раздел 6).
4. Локальная модификация графа относительно образца специфицируется образцами модификации (Раздел 7) и декларациями

свойств (Раздел 9).

5. Гибкий синтаксис обеспечивается раличными способами построения атомов из символов Unicode (Раздел 10) и иерархическим способом определения выражений (Раздел 11).
6. Категоризация сущностей целевого языка программирования обеспечивается логическими свойствами атоментов (Раздел 4).
7. Создание и удаление экземпляров этих категорий обеспечивается логическим свойством `[potential]` (Раздел 4).

3. ПРЕДВАРИТЕЛЬНЫЕ ПОНЯТИЯ

Язык Atoment базируется на понятиях атом, элемент и свойство. Множество атомов и элементов языка Atoment образует универсум, элементы которого называются атоментами. Атоменты выполняют роль данных и действий. В качестве действия атомент может выполняться, изменять состояние и возвращать значение. В качестве данных атомент используется как значение, возвращаемое действием, и как структура данных, содержание которой определяется как отображение из свойств в атоменты. Тотальная функция `st` из пар `(P, A)`, где `P` — свойство, `A` — атомент, в атоменты называется состоянием универсума. Атомент `st(P, A)` называется значением свойства `P` относительно атомента `A` в состоянии `st`. Специальный атом `void` является признаком того, имеет атомент `A` свойство `P` или нет. Атомент `A` имеет свойство `P`, если `st(P, A)` $\neq$ `void`. Атомент `A` не имеет свойства `P`, если `st(P, A) = void`. Атом `void` также используется в качестве возвращаемого значения действий, возвращаемое значение которых не важно.

3.1. Язык описания семантики действий

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 $\mathtt{st}'$, где $\mathtt{st}'$`(P, A) = B` для пары `(P, A)` и $\mathtt{st}'$`(V) = st(V)` для любой другой пары `V`;

- действие `A.P := B` аналогично действию `A.P ::= B`, за исключением 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.

Исключение — это элемент специального вида, который определяется в разделе 3.

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`

4. АТОМЕНТЫ И СВОЙСТВА

Свойства делятся на меточные, логические и порядковые.

Множество меточных свойств совпадает с множеством последовательностей атоментов вида {A_1 ... A_n}, где A_i — атоменты. В случае {A_1}, где A_1 — не целое число, скобки могут опускаться. Значение меточного свойства `P` атомента `A` определяется как `st(P, A)`.

Множество логических свойств совпадает с множеством последовательностей атоментов вида [A_1 ... A_n], где A_i — атоменты. Логическое свойство не имеет значения. Логическое свойство `[element]` означает, что атомент является элементом. Логическое свойство `[atom]` означает, что атомент является атомом.

Пример. Пусть атомент `A` не имеет меточного свойства `color` и логического свойства `[traffic lights]`. Действие `A.[traffic lights] := true` добавляет логическое свойство `[traffic lights]` к `A`. Это действие можно интерпретировать как добавление экземпляра `A` к содержимому понятия `[traffic lights]`, определяющему множество светофоров. Действие `A.color := green` добавляет меточное свойство `color` к `A` и присваивает этому свойству значение `green`. Действие `A.color := red` изменяет значение свойства `color` с `green` на `red`. Действие `A.color := void` удаляет меточное свойство `color` у `A`. Действие `A.[traffic lights] := void` удаляет логическое свойство `color` у `A`. Это действие можно интерпретировать как удаление экземпляра `A` из содержимого понятия `[traffic lights]`. Атомент `A` больше не является светофором после выполнения этого действия.

Множество порядковых свойств совпадает с множеством целых чисел. Целые числа являются атомами. Значение порядкового свойства `P` атомента `A` определяется как `st(P, A)`. Число порядковых свойств атомента называется его длиной. Атомент должен удовлетворять свойству непрерывности: если `i < j`, `A.i` $\neq$ `void`, `A.j` $\neq$ `void`, то `A.k` $\neq$ `void` для любого `i < k < j`. Атомент с порядковыми свойствами от `Min` до `Max` называется последовательностью (от `Min` до `Max`). Меточные

свойства left и right определяют различные виды последовательностей. Последовательность A со свойством left не может иметь порядковые свойства, меньшие чем A.left. Последовательность A со свойством right не может иметь порядковые свойства, большие чем A.right.

Пусть left(A) = A.left, если A.left $\neq$ void и left(A) = $-\infty$ иначе, right(A) = A.right, если A.right $\neq$ void и right(A) = $+\infty$ иначе. Пусть min(A) обозначает минимальное порядковое свойство A, max(A) обозначает максимальное порядковое свойство A. Действие A.m := B для порядкового свойства m зависит от вида последовательности A:

```
B ≠ void⟶
((A.m ≠ void⟶ (A.m присвоить B; возвратить void);
  A не имеет порядковых свойств и свойств left и right⟶
  (m > 0⟶ (A.left := 1; A.1 присвоить null);
   m = 0⟶ (A.left := 0; A.0 присвоить null);
   m < 0⟶ (A.right := -1; A.-1 присвоить null)));
 (left(A) ≤ m ≤ right(A)⟶
  (A.m присвоить B;
   для i от m+1 до min(A) выполнить: A.i присвоить null;
   для i от max(A) до m-1 выполнить: A.i присвоить null)));
иначе⟶
 (m = min(A) или m = max(A)⟶ A.m присвоить void;
  min(A) < m < max(A)⟶ A.m присвоить null)
```

Специальный атом null используется для заполнения дыр в последовательности A чтобы сохранить свойство непрерывности.

Пример. Пусть атомент A имеет порядковые свойства 1, 2, 3 со значениями 1, 4, 9. Пусть A.left = 0 и A.right = void. Действие A.0 := 0 добавляет порядковое свойство 0 к A и присваивает этому свойству значение 0. Действие A.=-1 := 0 ничего не делает, так как A.left = 0. Действие A.6 := 36 добавляет порядковые свойства 4, 5, 6 к A и присваивает им значения null, null, 36. Как уже было сказано, специальный атом null используется для заполнения дыр в последовательности A, чтобы сохранить свойство непрерывности. Действие A.3 := void не делает ничего, так как удаление порядкового свойства 3 нарушает свойство непрерывности. Действие A.3 := 36 удаляет порядковое свойство 6 у A.

Порядковые символические константы используются для ссылок на порядковые свойства атомента A. Пусть n — натуральное число. Поряд-

ковая символическая константа имеет один из следующих видов: `(min)`, `(min-n)`, `(min+n)`, `(max)`, `(max-n)`, `(max+n)`. Пусть порядковые свойства `A` образуют последовательность от `Min` до `Max`, `M` — порядковая символическая константа. Действие `получить значение M в A` имеет семантику:

`M = (min)` $\longrightarrow$ `возвратить Min;`
`M = (min-n)` $\longrightarrow$ `возвратить Min-n;`
`M = (min+n)` $\longrightarrow$ `возвратить Min+n;`
`M = (max)` $\longrightarrow$ `возвратить Max;`
`M = (max-n)` $\longrightarrow$ `возвратить Max-n;`
`M = (max+n)` $\longrightarrow$ `возвратить Max+n`

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.

Атомент `A` называется родителем `B` относительно `P`, если `A.P = B`. Атомент `A` называется родителем `B`, если `A.P = B` для некоторого `P`. Атомент `A` называется ребенком `B` относительно `P`, если `B.P = A`. Атомент `A` называется ребенком `B`, если `B.P = A` для некоторого `P`. Атомент `A` называется предком `B`, если `A.P = B` или `A.P` — предок `B` для некоторого `P`. Атомент `A` называется предком `B` относительно `P`, если `A.P = B` или `A.P` — предок `B` относительно `P`. Атомент `A` называется потомком `B`, если `B` — предок `A`. Атомент `A` называется потомком `B` относительно `P`, если `B` — предок `A` относительно `P`. Атомент `A` называется прародителем `B`, если `A` — предок `B` и `A` не имеет родителей. Атомент `A` называется прародителем `B` относительно `P`, если `A` — предок `B` относительно `P` и `A` не имеет родителя относительно `P`.

Замечание. Атоменты могут быть родителями (потомками) друг друга.

Атомент `A` называется копией `B`, если `A` и `B` имеют одни и те же свойства с одинаковыми значениями. Последовательность `A` называется копией последовательности `B`, если длины последовательностей `A` и `B` совпадают и `A.i` является копией `B.i` для каждого порядкового свойства `i`.

Действие `копировать B в A` добавляет к `A` свойства `B`. Оно имеет семантику:

`B = (`P_1`:`A_1 `...` P_n`:`A_n `[`Q_1`] ... [`Q_m`])` $\longrightarrow$
`(X.`P_1 `:=` A_1`; ...; X.`P_n `:=` A_n`;`

```
X.[Q1] := true; ...; X.[Qm] := true;
возвратить void)
```

Если A не имеет свойств, то после выполнения этого действия A становится копией B.

Элементы делятся на актуальные и потенциальные. Элемент называется потенциальным, если он имеет свойство [potential]. Элемент называется актуальным, если он не имеет свойства [potential]. Множество актуальных элементов называется актуальным универсумом. Потенциальные элементы не могут иметь свойств, кроме свойства [potential], и выполняться. Только актуальные элементы могут иметь свойства и выполняться. Таким образом, все действия над состоянием универсума выполняются в актуальном универсуме.

Потенциальные элементы могут добавляться к актуальному универсуму. Добавление потенциального элемента к актуальному универсуму называется порождением (актуального) элемента. Порождение актуального элемента состоит из нетедерминированного выбора потенциального элемента и удалении у него свойства [potential]. Действие породить элемент имеет семантику:

```
A.[element] ≠ void и A.[potential] ≠ void⟶
 (A.[potential] := void; возвратить A);
иначе⟶ возвратить void
```

Пусть new:(A1:B1 ... An:Bn) означает создать новый элемент D со свойствами A1, ..., An со значениями B1, ..., Bn и возвратить его, new:() — создать новый элемент D без свойств и возвратить его.

Актуальные элементы могут удаляться из актуального универсума. Удаление актуального элемента состоит в удалении всех его свойств и свойств родителей этого элемента, которые ссылаются на него, и добавлении к нему свойства [potential]. Пусть A — актуальный элемент. Действие удалить A имеет семантику:

```
(1: B.P = A⟶ (B.P := void; перейти к 1));
A = (P1:B1 ... Pn:Bn [Q1] ... [Qm])⟶
 (A.P1 := void; ...; A.Pn := void;
  A.[Q1] := void; ...; A.[Qm] := void;
  A.[potential] := true; возвратить void)
```

Действие возвращает либо чистое значение, либо void, либо исключение. Атом void является признаком того, что действие не возвращает чистого значения и не возвращает исключения. Исключения являются

специальными видами элементов. Атомент A называется чистым значением, если A $\neq$ void, и A не является исключением.

Исключение — это элемент с обязательным свойством exception и опциональным свойством value. Значением свойства exception является тип возвращаемого исключения или его описание. Значением свойства value является обычно действие, которое возвращает это исключение.

5. КОНТЕКСТЫ

Действия должны выполняться в контекстах выполнения.

Контекст выполнения ct — это элемент с обязательным свойством [context] и опциональным свойством value. В свойстве value хранятся значения, возвращаемые действиями, которые выполняются в контексте ct. Если действие A возвращает значение V в контексте ct, то ct.value := V. Атомент ct.value называется текущим значением в ct. Контекст ct может иметь свойство parent, значением которого является контекст. Контекст ct.parent называется родителем ct. Контекст может наследовать от родительского контекста данные и действия. Контекст может содержать элементы: A $\in$ ct тогда и только тогда, когда A.context = ct.

Так как контексты являются элементами, они могут добавляться в актуальный универсум (порождаться) и удаляться из него.

Порождение контекста состоит в порождении элемента и добавлении к нему свойства [context]. Действие породить контекст имеет семантику:

```
X := породить элемент; X.[context] := true;
копировать defaultcontext в X; возвратить X
```

Свойства специального атома defaultcontext наследуются контекстами, которые добавляются в универсум.

Удаление контекста состоит в удалении всех элементов, которые принадлежат ему, и самого контекста как элемента. Действие удалить контекст A имеет семантику:

```
(1: B.context = A⟶ (удалить элемент B; перейти к 1));
удалить элемент A
```

Действия могут иметь свойство executioncontext. Значение этого свойства является контекстом. Если действие A имеет это свойство, то

выполнение A в произвольном контексте сводится к выполнению A в контексте A.executioncontext.

Доступ к атоменту может осуществляться через специальное свойство uid. Атом A.uid называется уникальным идентификатором элемента A. Если атоменты A и B имеют свойство uid и A $\neq$ B, то A.uid $\neq$ B.uid.

Замечание. Свойство uid не является обязательным для атоментов.

Переменные — это специальный вид элементов, которые должны принадлежать контекстам. Переменные используются для ссылок на атоменты.

Переменная — это элемент с обязательными свойствами [variable], context, value и name. Пусть A — переменная. Элемент A.context является контекстом, которому принадлежит переменная A. Атомент A.value называется значением переменной A. Значение переменной A — это атомент, на который ссылается A. Атомент A.name называется именем переменной A. Имя переменной используется для доступа к ее значению.

Контекст может иметь свойство [inheritvar]. Если ct.[inheritvar] $\neq$ void, ct.parent $\neq$ void, x — переменная из ct.parent, переменная с именем x.name не принадлежит ct, то из контекста ct можно ссылаться по имени на переменную x из контекста st.parent.

Переменная x называется доступной в контексте ct, если для нее выполнено одно из свойств:

- x $\in$ ct;
- x $\notin$ ct, ct.[inheritvar] $\neq$ void и x доступна в ct.parent.

Действие получить значение переменной по имени A в ct имеет семантику:

```
A — имя переменной x из ct⟶ возвратить x.value;
ct.parent ≠ void⟶
 получить значение переменной по имени A в ct.parent;
иначе⟶ возвратить void
```

Пусть X — последовательность имен переменных, V — последовательность значений этих переменных.

Действие заменить переменные из X в A на V имеет семантику:

```
P1, ..., Pn — порядковые и меточные свойства A⟶
(M := {};
```

```
(A = X.r⟶ возвратить V.r;
 A — атом, не являющийся именем переменной из X⟶
  возвратить A;
 A ∉ M⟶
  (добавить A в M;
   ((заменить переменные из X в A.P_i на V) возвращает R_i
    для всех i⟶ (A.P_1 := R_1; ... A.P_n := R_n; возвратить A)))
 иначе⟶ возвратить A))
```

Переменная `M` хранит потомков элемента `E`, для которых уже была выполнена замена переменных.

Действие `заменить переменные из X в свойствах A на V` имеет семантику:

```
P_1, ..., P_n — порядковые и меточные свойства атомента A,
Q_1, ..., Q_m — логические свойства A⟶
(для i от 1 до n выполнить:
 (Pi = {B_1 ... B_k}⟶
   (для j от 1 до k выполнить:
    (B_j = X.r⟶ B_j' := V.r; иначе⟶ B_j' := B_j)
    P_i' := {B_1' ... B_k'});
  иначе⟶ P_i' := P_i);
 для i от 1 до m выполнить:
 (Qi = [C_1 ... C_s]⟶
   (для j от 1 до s выполнить:
    (C_j = X.r⟶ C_j' := V.r; иначе⟶ C_j' := C_j)
    Q_i' := [C_1' ... C_k']);
  иначе⟶ Q_i' := Q_i);
 D_1 := A.P_1; ...; D_n := A.P_n;
 A.P_1 := void; ...; A.P_n := void;
 A.P_1' := D_1; ...; A.P_n' := D_n;
 A.Q_1 := void; ...; A.Q_m := void;
 A.Q_1' := true; ...; A.Q_m' := true)
```

6. ОБРАЗЦЫ ПОИСКА

Образцы используется для поиска атоментов, удовлетворяющих определенным свойствам, создания и модификации атоментов. С образцом может быть связана последовательность, элементы которой называются параметрами образца.

Образцы делятся на образцы поиска и образцы модификации.

Образец поиска задает класс атоментов, которые имеют определенные свойства и определенные значения этих свойств. Про атомент, который принадлежит классу, говорят, что он сопоставляется с образцом. Образец поиска используется, чтобы проверять, сопоставляется ли некоторый атомент образцу, и, в случае сопоставления, извлекать значения свойств этого атомента и его потомков и сохранять их в параметрах образца.

Пусть `X` — последовательность параметров образца. Атомент `A` называется образцом поиска относительно `X`, если для него выполнено одно из свойств:

- `A = (compoundpattern:B)`, где `B` — последовательность и `B.i` — проекции поиска относительно `X`. В этом случае `A` называется составным образцом. Последовательность `B` может быть пустой;
- `A` $\neq$ `(compoundpattern:B)`. В этом случае `A` называется простым образцом.

Элемент `A` называется проекцией поиска относительно `X`, если для него выполнено одно из свойств:

- `A = [B]`;
- `A = (not:[B])`;
- `A = (prop:B value:C)`, где `B` — меточное или порядковое свойство, `C` — образец поиска;
- `A = (notprop:B)`, где `B` — меточное или порядковое свойство;
- `A = (from:B to:C value:D)`, где `B`, `C` — порядковые свойства, `D` $\in$ `X`;
- `A = (notfrom:B to:C)`, где `B`, `C` — порядковые свойства.

Действие `сопоставить E с A относительно X` возвращает либо `void`, что означает, что сопоставление невозможно, либо последовательность значений параметров из `X`. Оно имеет семантику:

```
(детально сопоставить E с A относительно X) возвращает void⟶
 возвратить void;
(детально сопоставить E с A относительно X) возвращает
 ({X1 = e1, ..., Xn = en}, G)⟶
  возвратить new:(1:e1 ... n:en)
```

Действие `детально сопоставить E с A относительно X` возвращает либо `void`, либо пару `(F, G)`, где `F` — множество равенств вида `x = e`, где `x` — переменная из `X`, `e` — атомент, который является значением `x`, `G` — множество свойств атомента `E`, которые были использованы в

сопоставлении. Оно имеет семантику:

```
A = (compoundpattern:B)⟶
 (заменить порядковые символические константы их значениями
  для проекций A;
 ((детально сопоставить E с проекцией B.i относительно X)
  возвращает (G_i, H_i) для каждого i⟶
   (G := объединение множеств G_1, ..., G_n;
    (для любого j если X.j = d и X.j = e принадлежат G,
      то d=e⟶
       (H := объединение множеств H_1, ..., H_n;
        (A имеет проекцию [!]⟶
         (H содержит все свойства E⟶ возвратить (G, H);
          иначе⟶ возвратить void);
         иначе⟶ возвратить (G, H)));
     иначе⟶ возвратить void));
   иначе⟶ возвратить void));
A ∈ X⟶ возвратить ({A=E}, {});
A = *⟶ возвратить ({}, {});
\\ Звездочка означает, что E может быть любым
A = E⟶ возвратить ({}, {});
иначе⟶ возвратить void
```

Действие детально сопоставить E с проекцией A относительно X имеет семантику:

```
A = [!]⟶ возвратить ({}, {});
A = [B]⟶
 (E.[B] ≠ void⟶ возвратить ({}, [B]);
  иначе⟶ возвратить void);
A = (not:[B])⟶
 (E.[B] = void⟶ возвратить ({}, [B]);
  иначе⟶ возвратить void);
A = (prop:B value:C)⟶
 (E.B ≠ void, (детально сопоставить E.B с C относительно X)
   возвращает (F, G)⟶ возвратить (F, B);
  иначе⟶ возвратить void);
A = (notprop:B)⟶
 (E.B = void⟶ возвратить ({}, [B]);
  иначе⟶ возвратить void);
A = (from:B to:C value:D)⟶
```

```
 (B ≤ C и B ≤ m_1 < ... < m_k ≤ C — порядковые свойства E⟶
   (D:=new:([rightsidesequence] 1:E.m_1 ... k:E.m_k);
    возвратить (x=D, m_1, ..., m_k));
  иначе⟶ возвратить ({}, {}));
A = (notfrom:B to:C)⟶
 (B>C или E не имеет порядкового свойства D такого, что
   B ≤ D ≤ C⟶ возвратить ({}, {});
  иначе⟶ возвратить void)
```

Атомент `A` называется условным образцом поиска, если `A = (searchpattern:B where:C var:D [show])`, где `B` — образец поиска, `C` — атомент, `D` — последовательность. Атомент `C` называется условием `A`, а последовательность `D` — спецификатором параметров `A`. Спецификатор `D` определяет последовательность параметров `A`. Условие `C` определяет условие на параметры `A`.

Действие `получить последовательность параметров образца по D` имеет семантику:

```
D = (D_1 ... D_n)⟶
 (F := new:();
  для i от 1 до n выполнить:
   (D_i = (exec:x)⟶ F.i := x; иначе⟶ F.i := D_i);
  возвратить F)
```

Действие `сопоставить E с A в ct` имеет семантику:

```
(получить последовательность параметров образца по A.var)
 возвращает последовательность X от 1 до n⟶
  ((сопоставить E с A.searchpattern относительно X в ct)
    возвращает чистое значение H⟶
     (породить ct′; ct′.parent := ct;
      new:([variable] context:ct′ value:H.1 name:X.1); ...;
      new:([variable] context:ct′ value:H.n name:X.n);
      (A.where возвращает чистое значение в ct′⟶
        (удалить ct′; возвратить H)));
   иначе⟶ (удалить ct′; возвратить void));
иначе⟶ возвратить void
```

Атомент может иметь скрытые свойства, которые не учитываются при сопоставлении этого элемента с образцом. Свойство `[show]` является индикатором того, учитывать скрытые свойства элемента при сопоставлении с образцом или нет. Скрытые свойства учитываются, если образец

имеет это свойство.

Пример. Образец `([figure])` определяет атоменты, которые имеют логическое свойство `[figure]`, т. е. являются геометрическими фигурами.

Пример. Образец `([figure] ~[triangle])` определяет атоменты, кторые имеют логическое свойство `[figure]` и не имеют логического свойства `[triangle]`, т. е. все геометрические фигуры за исключением треугольников.

Пример. Образец `([figure] length:5)` определяет атоменты, которые имеют логическое свойство `[figure]` и меточное свойство `length` со значением `5`, т. е. все геометрические фигуры длины 5.

Пример. Образец `(1:`$\mathtt{a}_1$ `... n:`$\mathtt{a}_n$`)` определяет атоменты, которые имеют порядковые свойства `1`, ..., `n` со значениями $\mathtt{a}_1$, ..., $\mathtt{a}_n$.

Пример. Образец `(1:`$\mathtt{x}_1$ `... n:`$\mathtt{x}_n$`)` с параметрами $\mathtt{x}_1$, ..., $\mathtt{x}_n$ определяет атоменты, которые имеют порядковые свойства `1`, ..., `n`. Значения этих свойств присваиваются при сопоставлении с образцом параметрам $\mathtt{x}_1$, ..., $\mathtt{x}_n$.

Пример. Образец `([figure] length:x ~width)` с параметром `x` определяет атоменты, которые имеют логическое свойство `[figure]`, меточное свойство `length` и не имеют меточного свойства `width`, т. е. все геометрические фигуры, имеющие длину и не имеющие ширины (в этот класс попадают, например, отрезки). Значение свойства `length` присваивается при сопоставлении с образцом параметру `x`.

Пример. Образец `if:x then:y else:z` с параметрами `x`, `y` и `z` определяет множество условных операторов. Сопоставление условного оператора `E` с этим образцом присваивает значения значения `E.if`, `E.then` и `E.else` параметрам `x`, `y` и `z`, соответственно.

Пример. Пусть `E = (1:1 2:2 3:3 4:4 5:5)`. Образец `(2#4:x) (-3#1:y) (7#10:z)` с параметрами `x`, `y` и `z` присваивает новый элемент `new:(2:2 3:3 4:4)` параметру `x`, новый элемент `new:(1:1)` — параметру `y`, и новый элемент `new:()` с пустым множеством свойств — параметру `z`.

7. ОБРАЗЦЫ МОДИФИКАЦИИ

Образец модификации применяется к элементу и определяет следующие преобразования:

- добавление свойств к элементу и его потомкам;

- удаление свойств у элемента и его потомков;
- изменение значений свойств элемента и его потомков.

Образец модификации используется также для порождения элементов с определенными свойствами. В этом случае порождаются новые элементы с пустым множеством свойств, а затем они модифицируются в соответствии с образом модификации.

Атомент A называется образцом модификации, если для него выполнено одно из свойств:

- A = (compoundpattern:B), где B — последовательность и B.i — проекции модификации. В этом случае A называется составным образцом. Последовательность B может быть пустой;
- A $\neq$ (compoundpattern:B). В этом случае A называется простым образцом.

Элемент называется проекцией модификации, если для него выполнено одно из свойств:

- (M:B), где B — образец модификации, M $\in$ {inslab, inslog, inslabn, inslogn, insleft, insright};
- (del:M), где M $\in$ {lab, log, ord};
- [B];
- (not:[B]);
- (prop:B value:C), где B — меточное или порядковое свойство, C — образец модификации;
- (notprop:B), где B — меточное или порядковое свойство;
- [new].

Действие модифицировать E относительно A имеет семантику:

```
A = (compoundpattern:B), где B — последовательность
от 1 до n⟶
 (заменить порядковые символические константы их значениями
   для проекций A;
  для i от 1 до n выполнить:
   модифицировать E относительно проекции B.i)
```

Действие модифицировать E относительно проекции A имеет семантику:

```
A = (M:B)⟶
 ((B — составной образец⟶
    (D := new:(); модифицировать D относительно B);
   иначе⟶ D := B);
  (M = inslab, P1, ..., Pn — меточные свойства D⟶
```

```
    (E.P1 := D.P1; ...; E.Pn := D.Pn);
   M = inslog, P1, ..., Pn — логические свойства D⟶
    (E.P1 := D.P1; ...; E.Pn := D.Pn);
   M = inslabn, P1, ..., Pn — меточные свойства D⟶
    для i от 1 до n выполнить: (E.Pi = void⟶ E.Pi := D.Pi);
   M = inslogn, P1, ..., Pn — логические свойства D⟶
    для i от 1 до n выполнить: (E.Pi = void⟶ E.Pi := D.Pi);
   M = insleft, P1, ..., Pn — порядковые свойства D⟶
    для i от 1 до n выполнить: E.(max+1) := D.Pi;
   M = insleft, P1, ..., Pn — порядковые свойства D⟶
    для i от n до 1 выполнить: E.(min-1) := D.Pi));
A = (del:lab), P1, ..., Pn — меточные свойства E⟶
 (E.P1 := void; ..., E.Pn := void);
A = (del:log), P1, ..., Pn — логические свойства E⟶
 (E.P1 := void; ..., E.Pn := void);
A = (del:ord), P1, ..., Pn — порядковые свойства E⟶
 (E.P1 := void; ..., E.Pn := void);
A = [B]⟶ E.[B] := true;
A = (not:[B])⟶ E.[B] := void;
A = (prop:B value:C)⟶
 (C — составной образец⟶
   (E.B ≠ void⟶
     (C имеет проекцию [new]⟶
       (C′ := удалить из C проекцию [new]; D := new:();
        модифицировать D относительно C′; E.B := D);
      иначе⟶ модифицировать E.B относительно C);
    иначе⟶
     (C′ := удалить из C проекцию [new]; D := new:();
      модифицировать D относительно C′; E.B := D));
  иначе⟶ E.B:=C);
A = (notprop:B)⟶ E.B := void
```

Атомент A называется условным образцом модификации, если A = (modificationpattern:B where:C var:D), где B — образец модификации, C — атомент, D — последовательность. Атомент C называется условием A, а последовательность D — спецификатором параметров A. Спецификатор D определяет последовательность параметров A. Условие C определяет условие на параметры A.

Действие получить последовательность параметров образца по D имеет семантику:

```
D = (D_1 ... D_n)⟶
 (F := new:();
  для i от 1 до n выполнить: F.i := D_i;
  возвратить F)
```

Действие модифицировать E относительно A в контексте ct имеет семантику:

```
(получить последовательность параметров образца по A.var)
возвращает последовательность X от 1 до n⟶
 (M := {};
  (1: F — последовательность от 1 до n, F ∉ M⟶
    (добавить F к M; ct′ := породить контекст;
     ct′.parent := ct; ct′.[inheritvar] := true;
     new:([variable] context:ct′ value:F.1 name:X.1); ...;
     new:([variable] context:ct′ value:F.n name:X.n);
     (A.where возвращает чистое значение в ct′⟶
       (модифицировать E относительно A.modificationpattern;
        удалить ct′; возвратить void);
      иначе⟶ удалить ct′; перейти к 1));
   иначе⟶ возвратить void))
```

Пример. Действие modify:E match:([figure] length:5 ~width) эквивалентно последовательному выполнению действий E.[figure] := true; E.length := 5; E.width := void.

Пример. Действие modify:E match:(B:(C:3)) эквивалентно действию E.B.C := 3, которое изменяет свойство C потомка E.B атомента E.

Пример. Действие modify:E match:(B:(C:3) [new]) эквивалентно последовательному выполнению действий x := new:(C:3); E.B := x.

Пример. Действие modify:E match:(inslab:A inslog:B insright:C inslabn:D) добавляет к E меточные свойства атомента A, логические свойства атомента B, порядковые свойства атомента C и меточные свойства атомента D. Порядковые свойства атомента C добавляются справа от порядковых свойств E. Значения меточных свойств A изменяют значения меточных свойств E. Меточное свойство атомента D добавляется к E только в случае, если E не имеет этого свойства. Пусть

E = (u:1 v:2 w:3 [c] 1:1 2:2), A = (u:2 a:3), B = ([b]), C = (1:3 2:4) и D = (v:5 r:6). Тогда E′ = u:2 v:2 w:3 a:3 r:6 [c] [b] 1:1 2:2 3:3 4:4), где E′ — результат модификации E.

Пример. Действие modify:E match:(del:lab del:log) удаляет все меточные и логические свойства атомента E.

8. СХЕМЫ ВЫПОЛНЕНИЯ

Выполнение атоментов (действий) описывается схемами выполнения. Схема выполнения определяет множество атоментов и способ выполнения атоментов этого множества. Говорят, что атомент удовлетворяет схеме, если он принадлежит этому множеству.

Замечание. Атомент может удовлетворять нескольким схемам выполнения.

Схема выполнения — это элемент с обязательными свойствами ifpattern и context и опциональными свойствами where, whereafter, var, id, [macro] и then такой, что элемент new:(match:P var: S.ifpattern where:S.where), где P — результат добавления к S.ifpattern свойства [!], является условным образцом поиска. Схема выполнения должна принадлежать контексту, который определяется свойством context.

Пусть S — схема действия, CSP — условный образец поиска, соответствующий S.

Атомент S.id называется идентификатором схемы S. Он используется для ссылки на S. Свойства ifpattern, where, whereafter и var используются для проверки удовлетворяет ли элемент схеме или нет. Параметры CSP называются параметрами S. Они используются для хранения значений свойств атоментов, удовлетворяющих схеме S.

Параметры S делятся на выполняемые и невыполняемые, а их значения — на предварительные и окончательные.

Последовательность предварительных значений параметров получаются при сопоставлении атомента E с CSP. Для выполняемых параметров окончательные значения суть результат выполнения предварительных значений этих параметров. Для невыполняемых параметров окончательные значения совпадают с предварительными значениями этих параметров. Выполняемый параметр определяется в S.var как значение свойства exec.

Действие сопоставить E с S в ct возвращает либо void, что озна-

чает, что сопоставление невозможно, либо возвращает пару (Y, ct′), где Y — последовательность переменных, которые соответствуют параметрам S (имеют эти параметры в качестве имен) и имеют значения, полученные в результате сопоставления с образцом CSP, ct′ — новый контекст, которому принадлежат эти переменные. Оно имеет семантику:

```
X — последовательность переменных схемы S от 1 до n⟶
 ((сопоставить E с CSP) возвращает чистое значение V⟶
   (ct′ := породить контекст; ct′.parent := ct;
    ct′.[inheritvar] := true;
    Y_1 := new:([variable] context:ct′ value:V.1 name:X.1);
    ...;
    Y_n := new:([variable] context:ct′ value:V.n name:X.n);
    Y := new:(context:ct′ 1:Y_1 ... n:Y_n);
    для i от 1 до n выполнить:
     (X.i — выполняемая переменная⟶
       (V.i возвращает чистое значение D в ct′⟶
         Y.i.value := D;
        V.i возвращает исключение D в ct′⟶
         (удалить ct′; возвратить D));
      иначе⟶ (удалить ct′; возвратить void));
    (S.whereafter в ct′ возвращает чистое значение⟶
      возвратить (Y, ct′);
     иначе⟶ (удалить ct′; возвратить void)));
  иначе⟶ возвратить void)
```

Свойства where, whereafter и var могут отсутствовать. Отсутствие свойств where и whereafter означает, что нет ограничений на предварительные и окончательные значения параметров схемы S, соответственно. Отсутствие свойства var означает, что схема S не имеет параметров.

Схемы делятся на определяемые и предопределенные. Определяемые схемы — это схемы, имеющие свойство then. Свойства then и [macro] определяют способ выполнения элементов, которые удовлетворяют схеме. Предопределенные схемы не имеют свойства then. Способ выполнения элементов, удовлетворяющих предопределенным схемам, определяется внешним образом, но параметры схемы S по-прежнему являются параметрами этого способа.

Среди свойств выделяют функциональные свойства. Функциональное свойство A имеет схему с образцом E.A и параметрами E и A. Элемент

E.A, удовлетворяющий этой схеме, возвращает либо значение свойства элемента E, либо void, если E не имеет свойства A.

Действие выполнить атомент A в ct имеет семантику:

```
A.executioncontext = void⟶
 ((A — имя переменной x из ct⟶ возвратить x.value;
   иначе⟶
    (1: ct.[inheritvar] ≠ void и ct.parent ≠ void⟶
      (ct := ct.parent;
       (A — имя переменной x из ct⟶ возвратить x.value;
        иначе⟶ перейти к 1))));
  (X — последовательность доступных переменных в ct,
   V — последовательность значений переменных из X⟶
    A′ := заменить переменные из X в свойствах A на V);
  (2: (сопоставить A′ с S в ct) возвращает (X, ct′)⟶
    ((S имеет свойство [macro],
      X.i1, ..., X.im — невыполняемые параметры S⟶
       (X.i1.value.executionstate := st; ...;
        X.im.value.executionstate := st));
     (S имеет свойство then⟶
       (V := выполнить S.then в ct′; удалить ct′;
        возвратить V);
      иначе⟶
       (V := реализация схемы S относительно X; удалить ct′;
       возвратить V)));
   A′ удовлетворяет нескольким схемам действий⟶
    (выбрать любую из этих схем; перейти к 2);
   иначе⟶
    ((ct.parent ≠ void, ct.[inherit] ≠ void⟶
       (ct := ct.parent; перейти к 2));
     (A — атом⟶ возвратить A;
      A — элемент⟶ возвратить new:(exception:
       ""there is no execution scheme" value:THIS)))));
иначе⟶ выполнить A в A.executioncontext
```

Пример.

```
ifpattern:f(x) [macro] var:$x then:(z := 5; execute:x);
z := 0;
f(z:=z+1)
```

Здесь x — выполняемая переменная образца ifpattern:f(x) var:$x, так как x имеет префикс $. Действие f(z:=z+1) в контексте ct эквивалентно действию z := 5; z:=z+1 в новом контексте ct′, причем действие z:=5 выполняется в контексте ct′, а действие z:=z+1 выполняется в контексте ct. Действие execute:x из стандартной библиотеки языка Atoment [9] выполняет атомент, который является значением действия x.

9. ДЕКЛАРАЦИИ СВОЙСТВ

Декларации свойств позволяют переопределять базовые операции для конкретного свойства: присвоить значение свойству, получить значение свойства, удалить свойство, проверить наличие свойства. Дополнительно декларация свойства определяет множество атоментов, к которым применимы переопределенные операции.

Атомент A называется декларацией свойства, если он имеет обязательное свойство prop и опциональные свойства var, set, get, where и [readonly]. Атомент A.prop определяет свойство P, для которого специализируются базовые операции. Атомент A.var определяет параметры для базовых операций и имеет вид (A B). Атоменты A.set и A.get определяют базовые операции: присвоить значение B свойству P вида A.P := B и получить значение свойства P вида A.P. Атомент A.where определяет множество атоментов, к которым применимы базовые операции, определенные свойствами set и get. Логическое свойство [readonly] означает, что нельзя изменять значение свойства P.

Базовая операция удаления свойства P выражается посредством A.P := void, а базовая операция проверки наличия свойства P у элемента A выражается посредством A.P ≠void.

Свойство, для которого имеется декларация, называется функциональным.

Пример. Определим свойство stack следующим образом:

```
(prop:stack var:(A B) where:(A.[stack])
 set:
 (if:(B = void)
  then:
  (if:(A.max != void)
   then:(x := A.max; A.max := x - 1; A.x := void))
  else:
```

```
  (if:(A.max = void)
   then:(A.max := 0; A.0 := B)
   else:(x := A.max + 1; A.max := x; A.x := B))
 get:(if:(A.max != void) then:(x := A.max; return(A.x))
```

Это свойство реализует стек на последовательности от 0 до $+\infty$. Свойство max определяет номер верхнего элемента стека, а порядковые свойства 0, ..., max — элементы стека. Ниже приведен пример создания, использования и удаления стека.

```
A.[stack] := true; // A = ([stack])
A.stack := 0; // A = ([stack] max:0 0:0)
A.stack; // возвращает 0
A.stack := 1; // A = ([stack] max:1 0:0 1:1)
A.stack; // возвращает 1
A.stack := void; // A = ([stack] max:0 0:0)
A.stack; // возвращает 0
A.stack := void; // A = ([stack])
A.stack; // возвращает void
A.stack := void; // A = ([stack])
A.stack; // возвращает void
A.[stack] := void; // A = ()
A.stack // возвращает void
```

В следующих разделах описаны синтаксические конструкции языка Atoment и алгоритмы их трансляции в семантические сущности.

10. АТОМЫ И СВОЙСТВА

Конструкции языка Atoment строятся из символов Unicode, среди которых выделяются пробельные символы, специальные символы и скобки. Пробельный символ определяется как любой символ Unicode класса Zs (который включает пробел) или символ из набора, включающего символ горизонтальной табуляции, символ вертикальной табуляции и символ возврата каретки. Множество специальных символов включает

```
" ~ ‘ @ # $ % ^ & * - + : ; ′ . < > / , | \ = ? !
```

Множество скобок включает

```
( ) { } [ ]
```

Конструкции языка включают атомы, свойства и выражения.

Последовательность Unicode символов называется атомом, если она имеет один из видов:

1) последовательность Unicode символов, в которой каждому вхождению скобки, пробельного символа или специального символа предшествует символ \;
2) символ : или последовательность специальных символов за исключением символов " и \, которая не заканчивается символом :;
3) A"B"CB, где A — (возможно пустая) последовательность Unicode символов вида (1), B — непустая последовательность Unicode символов, не содержащая символа ", C — последовательность Unicode символов, не содержащая B" в качестве подпоследовательности;
4) A""C, где A — (возможно пустая) последовательность Unicode символов вида (1), C — последовательность Unicode символов, каждому вхождению символа " в которую предшествует символ \;
5) целое число, например, -2;
6) вещественное число, например, 3.1415 или 10.12E-15;
7) {A_1 ... A_n} или '{A_1 ... A_n}, где A_i — атомы видов (1)-(6);
8) [A_1 ... A_n] или '[A_1 ... A_n], где A_i — атомы видов (1)-(6);
9) 'A, где A — атом вида (1)-(4).

Атомы имеют предопределенные свойства, которые могут использоваться в образцах поиска и модификации и в операциях ., ::= и :=. Набор предопределенных свойств зависит от вида атомов.

Атомы всех видов имеют логическое свойство [atom].

Пусть E — атом вида (1), задаваемый последовательностью длины n. Тогда E имеет логическое свойство [pure atom], меточное свойство length со значением n и порядковые свойства 1, ..., n, значениями которых являются соответствующие символы этой последовательности.

Пример. Атом if сопоставляется с образцом ([pure atom] 1:i 2:f). Действие modify:if match:(1:o) заменяет if на of.

Пусть E — атом вида (2), задаваемый последовательностью длины n. Тогда E имеет логическое свойство [special atom], меточное свойство length со значением n и порядковые свойства 1, ..., n, значениями которых являются соответствующие символы этой последовательности.

Пусть E — атом вида (3). Тогда E имеет логическое свойство [text atom], меточные свойства type, delimiter и text со значениями A, B′ и C′, где B′ и C′ — атомы вида (1), полученные из B и C заменой каждого

вхождения скобки, пробельного или специального символа S на \S.

Пример. Действие `new:([text atom] delimiter:\' text:A)` порождает атом `"'"A'"`.

Пример. Действие

```
new:([text atom] type:MultilineComment delimiter:**
text:Это\ многострочный\ комментарий\,
не\ содержащий последовательности\ символов\ **.)
```

порождает атом

```
MultilineComment"**"Это многострочный комментарий,
не содержащий последовательности символов **".
```

Пусть E — атом вида (4). Тогда E имеет логическое свойство `[string atom]`, меточные свойства `type` и `text` со значениями A и C′, где C′ — атом вида (1), полученные из B и C заменой каждого вхождения скобки, пробельного или специального символа S на \S.

Пример. Действие `new:([string atom] text:green)` порождает атом `""green"`.

Пример. Действие `new:([string atom] type:color text:green)` порождает атом `color""green"`.

Пусть E — атом вида (5), задаваемый последовательностью цифр длины n, возможно, со знаком. Тогда E имеет обязательное логическое свойство `[integer]`, опциональное логическое свойство `[-]`, которое означает, что E имеет знак, меточное свойство `length` со значением n и порядковые свойства 1, ..., n, значениями которых являются соответствующие цифры в представлении E.

Пусть E — атом вида (6). Тогда E имеет обязательное логическое свойство `[real]`, опциональное логическое свойство `[-]`, которое означает, что E имеет знак и меточные свойства `int`, `frac` и `order`, значениями которых являются атомы вида (5), представляющие целую, дробную и порядковую части вещественного числа.

Пример. Атом `10.12E-15` сопоставляется с образцом `({~[-] int:10 frac:12 order:-15})`.

Пусть E — атом вида (7), задаваемый последовательностью длины n. Тогда E имеет логическое свойство `[labelled property]`, меточное свойство `length` со значением n и порядковые свойства 1, ..., n, значениями которых являются соответствующие атомы видов (1)-(6).

Пусть E — атом вида (8), задаваемый последовательностью длины n. Тогда E имеет логическое свойство `[logic property]`, меточное свой-

ство `length` со значением `n` и порядковые свойства `1`, ..., `n`, значениями которых являются соответствующие атомы видов (1)-(6).

Атом вида `'A`, имеет те же свойства, что и атом `A`, и дополнительно логическое свойство `[hided]`.

Атомы видов (1)-(4) и (6) имеют дополнительно логическое свойство `[labelled property]`. Атомы вида (5) имеют дополнительно логическое свойство `[ordinal property]`.

Таким образом, свойства являются атомами специального вида. Свойства имеют такой же вид как атомы. Скрытые свойства имеют один из видов: (9), `'{`A_1 ... A_n`}` или `'[`A_1 ... A_n`]`. Таким образом, скрытыми могут быть только меточные и логические свойства.

Пример. Рассмотрим, как различные виды атомов используются для определения лексических конструкций языка Си. Атомы вида `c\"B"` или `cL\"B"`, где `B` — последовательность специальных символов и представимых символов языка Си за исключением символов `'`, `\` и символа новой строки, представляют символьные константы языка Си.

Атомы вида `s""B"` или `sL"B"`, где `B` — последовательность специальных символов и представимых символов языка Си за исключением символов `"`, `\` и символа новой строки, представляют строковые константы языка Си.

Атомы вида `com"d"Bd"`, где `B` — последовательность символов Unicode, которая не содержит подпоследовательности `d"`, представляет комментарии языка Си.

11. ВЫРАЖЕНИЯ

Выражения представляют атоменты. Для каждого вида выражений имеется алгоритм трансляции их в атоменты. По способу трансляции выражения делятся на выражения порядка `0`, выражения порядка `1` и т. д. Выражения порядка `0` непосредственно транслируются в атоменты. Выражения порядка `k`, где `k>0`, транслируются в выражения порядков, меньших чем `k`.

Замечание. Алгоритмы трансляции могут быть недетерминированными.

Выражение порядка `0` имеет один из видов:

1. атом или `()`;
2. логическое свойство;
3. `N:B`, где `N` — порядковое или меточное свойство, `B` — выражение

вида 1 или 4;

4. (A$_1$ ... A$_n$), где A$_i$ — выражения вида 2 или 3.

Пусть A — выражение порядка 0. Действие транслировать A имеет семантику:

```
A — атом⟶ возвратить A;
A = ()⟶ возвратить new:(context:ct);
A — логическое свойство⟶ возвратить new:(context:ct A);
A = N:B, (транслировать B) возвращает D⟶
 возвратить new:(context:ct N:D);
A = (A1 ... An)⟶
 (E := new:(context:ct);
  для i от 1 до n выполнить:
  (Ai = N⟶ E.N := true;
   Ai = N:B, (транслировать B) возвращает D⟶ E.N := D);
  возвратить E)
```

Пример. Выражение (1:red 2:yellow 3:green) определяет элемент с порядковыми свойствами 1, 2 и 3 со значениями red, yellow и green.

Пример. Выражение ([светофор] цвет:зеленый) определяет элемент с логическим свойством [светофор] и именованным свойством цвет со значением зеленый. Логическое свойство используется здесь, чтобы задать категорию (тип) элемента.

Выражения более высоких порядков обеспечивают более гибкий и удобный синтаксис по сравнению с выражениями порядка 0. Рассмотрим основные виды таких выражений.

Списочные выражения имеют один из видов:

1) атом или ();
2) логическое свойство;
3) N:B, где N порядковое или меточное свойство, B — выражение вида 1 или 4;
4) (A$_1$ ... A$_n$), где A$_i$ — списочные выражения.

Они позволяют использовать списочную нотацию и являются выражениями порядка 1. Пусть A — списковое выражение. Действие транслировать A имеет семантику:

```
A — атом, () или логическое свойство⟶ возвратить A;
A = N:B, (транслировать B) возвращает D⟶ возвратить N:D;
A = (A1 ... An)⟶
 (для i от 1 до n выполнить:
```

```
(Ai имеет вид 2 или 3, (транслировать Ai) возвращает Di⟶
  Ai′ := Di;
 иначе⟶
 (j — ближайшее к i, j < i, Aj′ имеет вид m:B,
  m — порядковое свойство,
  (транслировать Ai) возвращает Di⟶
   Ai′ := (m+1):Di;
  (транслировать Ai) возвращает Di⟶ Ai′ := 1:Di));
возвратить (A1′ ... An′))
```

Пример. Списковое выражение `(red yellow green)` сводится к выражению `(1:red 2:yellow 3:green)`.

Префиксные выражения `F(X1, ..., Xn)` транслируются в выражения `(applyfun:F 1:X1 ... n:Xn)`. Они позволяют записывать выражения в префиксной форме.

Пример. Префиксное выражение `sin(x)` сводится к выражению `(applyfun:sin 1:x)`.

Неоднозначность между одноместными префиксными и списковыми выражениями разрешается в зависимости от наличия пробельного символа перед скобкой:

- `(a (b))` транслируется в `(1:a (1:b))`;
- `(a(b))` транслируется в `(1:(applyfun:a 1:b))`.

Инфиксные выражения `(A1 op ... op An)` транслируются в `(applyfun:x 1:A1 ... n:An)`. Инфиксные выражения позволяют записывать бинарные операции в инфиксной форме и использовать ассоциативность операций. Бинарные операции делятся на предопределенные (в стандартной библиотеке языка) и определяемые (схемами выполнения). Схема выполнения для бинарной операции `op` имеет образец `A op B` с параметрами `A` и `B`. Ассоциативность операции определяется свойством `[assoc]` этой схемы.

Другие виды выражений определяются в стандартной библиотеке языка Atoment.

Для бинарных операций задан приоритет, определяющий порядок их применения. Последовательность операций в порядке убывания приоритета имеет следующий вид (операции в круглых скобках имеют одинаковый приоритет):

```
(.) (* / %) (+ -) (< > <= >= = !=) (:=) (and) (or) (:) (;)
```

Пример. Выражение `(x+y*z)` транслируется в выражение `(applyfun:+ 1:x 2:(applyfun:* 1:y 2:z))`, так как приоритет опера-

ции * больше приоритета операции +.

Выполнение выражения состоит в трансляции выражения в атомент и выполнении этого атомента. Действие `выполнить E` имеет семантику:

```
ct′ := породить контекст; ct′.parent := ct;
ct′.[inheritvar] := true; ct′.[inherit] := true;
R := транслировать E в ct′;
V := выполнить R в ct; удалить ct′; возвратить V
```

Вспомогательный контекст $\texttt{ct}'$ используется для хранения результатов трансляции. После завершения выполнения выражения, этот контекст удаляется.

11.1. Выражения поиска

Выражения поиска используются чтобы представлять образцы поиска.

Пусть X — последовательность. Выражение называется выражением поиска относительно X, если оно имеет один из видов:

- атом;
- $(\texttt{A}_1\ \ldots\ \texttt{A}_n)$, где $\texttt{A}_i$ — выражения-проекции поиска.

Выражение называется выражением-проекцией поиска относительно X, если оно имеет один из видов:

- [B];
- (~[B]);
- B:x, где B — меточное или порядковое свойство, x — выражение поиска;
- ~B, где B — меточное или порядковое свойство;
- B#C:D, где B, C — целые числа, D принадлежит X;
- ~B#C, где B, C — целые числа.

Пусть A — выражение поиска. Действие `транслировать A относительно X` имеет семантику:

```
A = (A1 ... An), где Ai — выражения-проекции поиска⟶
 (для i от 1 до n выполнить: Ei := транслировать проекцию Ai;
  возвратить new:(compoundpattern:new:(1:E1 ... n:En)));
иначе⟶ возвратить A
```

Действие `транслировать проекцию A` имеет семантику:

```
A — атом⟶ возвратить A;
A = [B]⟶ возвратить [B];
```

```
A = ~[B]⟶ возвратить new:(not:[B]);
A = B:x⟶
 (D := транслировать x; возвратить new:(prop:B value:D));
A = ~B⟶ возвратить new:(notprop:B);
A = B#C:D⟶ возвратить new:(from:B to:C value:D);
A = ~B#C⟶ возвратить new:(notfrom:B to:C).
```

Пример. Образец `([фигура])` определяет атоменты, которые имеют свойство `[фигура]`.

Пример. Образец `([фигура] ~[треугольник])` определяет атоменты, которые имеют свойство `[фигура]` и не имеют свойства `[треугольник]`.

Пример. Образец `([фигура] длина:5)` определяет атоменты, которые имеют логическое свойство `[фигура]` и меточное свойство `длина` со значением `5`.

Пример. Образец `(1:`a_1 `... n:`a_n`)` определяет атоменты, имеющие свойства `1`, ..., `n` со значениями a_1, ..., a_n.

Образец `(1:`x_1 `... n:`x_n`)` с параметрами x_1, ..., x_n определяет атоменты, имеющие свойства `1`, ..., `n`. Значения этих свойств сохраняются в параметрах x_1, ..., x_n.

Пример. Образец `([фигура] длина:x ~ширина)` с параметром `x` определяет атоменты, которые имеют логическое свойство `[фигура]`, меточное свойство `длина` и не имеют меточного свойства `ширина`. Значение свойства `длина` сохраняется в параметре `x`.

Условные выражения поиска используются, чтобы представлять условные образцы поиска.

Выражение `A` называется условным выражением поиска, если `A` имеет вид `B where:C var:D [show]`, где `B` — образец поиска, `C` — выражение, `D` — выражение-спецификатор параметров поиска. Выражение `C` называется условием `A`. Выражение `D` определяет параметры `A`.

Выражение называется выражением-спецификатором параметров поиска, если оно имеет один из видов:

- атом;
- `(`D_1 `...` D_n`)`, где
  - либо D_i — атом;
  - либо D_i `= $x`, где `x` — атом.

Действие `транслировать D` имеет семантику:

```
D — атом⟶ возвратить new:(1:D);
D = (D1 ... Dn)⟶
```

```
((Di — атом⟶ Di′ := Di;
  Di = $x, где x — атом⟶ Di′ := new:(exec:x));
 возвратить new:(1:D1′ ... n:Dn′))
```

Пусть A — условное выражение поиска. Действие транслировать A имеет семантику:

```
B′ := транслировать B; C′ := транслировать C;
D′ := транслировать D;
возвратить new:(modificationpattern:B′ where:C′ var:D′ [show])
```

12. ВЫРАЖЕНИЯ МОДИФИКАЦИИ

Выражения модификации используются, чтобы представлять образцы модификации.

Выражение называется выражением модификации, если оно имеет один из видов:

- атом;
- (A1 ... An), где Ai — выражения-проекции модификации A, n ≥ 0.

Выражение называется выражением-проекцией модификации, если оно имеет один из видов:

- M:B, где B — выражение модификации и M ∈ {inslab, inslog, inslabn, inslogn, insleft, insright};
- del:M, где M ∈ {lab, log, ord};
- [B];
- (~[B]);
- B:x, где B — меточное или порядковое свойство, x — выражение модификации;
- ~B, где B — меточное или порядковое свойство.

Пусть A — выражение модификации. Действие транслировать A имеет семантику:

```
A — атом⟶ возвратить A;
A = (A1 ... An), где Ai — выражения-проекции образца
 модификации⟶
 ((транслировать проекцию Ai) возвращает Ei для каждого i⟶
   возвратить new:(compoundpattern:new:(1:E1 ... n:En)))
```

Действие транслировать проекцию A имеет семантику:

```
A = M:B⟶ (B′ := транслировать B; возвратить new:(M:B′));
```

```
A = del:M⟶ возвратить new:(del:M);
A = [B]⟶ возвратить [B];
A = (~[B])⟶ возвратить new:(not:[B]);
A = B:x⟶
 (D := транслировать x; возвратить new:(prop:B value:D));
A = ~B⟶ возвратить new:(notprop:B)
```

Пример. Действие `modify:E match:([фигура] длина:5 ~ширина)` эквивалентно действию `E.[фигура] := true; E.длина := 5; E.ширина := void`.

Пример. Действие `modify:E match:(B:(C:3))` эквивалентно действию `E.B.C := 3`.

Пример. Действие `modify:E match:(B:(C:3) [new])` эквивалентно действию `x := new:(C:3); E.B := x`.

Условные выражения модификации используются, чтобы представлять условные образцы модификации.

Выражение `A` называется условным выражением модификации, если `A` имеет вид `B where:C var:D`, где `B` — выражение модификации, `C` — выражение, `D` — выражение-спецификатор параметров модификации. Выражение `C` называется условием `A`. Выражение `D` определяет параметры `A`.

Выражение называется выражением-спецификатором параметров модификации, если оно имеет один из видов:

- атом;
- `(`D_1 `...` D_n`)`, где D_i — атомы.

Действие `транслировать D` имеет семантику:

```
D — атом⟶ возвратить new:(1:D);
D = (D1 ... Dn)⟶  возвратить new:(1:D1 ... n:Dn)
```

Пусть `A` — условное выражение модификации. Действие `транслировать A` имеет вид:

```
B′ := транслировать B; C′ := транслировать C;
D′ := транслировать D;
возвратить new:(modificationpattern:B′ where:C′ var:D′)
```

13. ЗАКЛЮЧЕНИЕ

В работе дана спецификация языка Atoment. Рассмотрены синтаксис и семантика языка. Сформулированы требования к предметно-ори-

ентированному языку разработки методов верификации программ, и показано, что язык Atoment удовлетворяет всем этим требованиям. 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. Непомнящий В.А., Анureев 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).
2. Шилов Н.В., Анureев И.С., Бодин Е.В. О генерации условий корректности для императивных программ // Программирование. — 2008. — № 6. — С. 307–321.
3. Непомнящий В.А., Анureев 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.
4. Непомнящий В.А., Анureев И.С., Промский А.В. На пути к верификации С программ. Аксиоматическая семантика языка C-kernel // Программирование. — 2003. — № 6. — С. 338–350.
5. Непомнящий В.А., Анureев И.С., Промский А.В., Дубрановский И.В. На пути к верификации C# программ: трехуровневый подход // Программирование. — 2006. — № 4. — С. 190–202.
6. Anureev I.S. A three-stage method of C program verification // Joint NCC&IIS Bulletin. Ser. Comput. Sci. — 2008. — Vol. 28 — P. 1–30.
7. Анureев 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.
8. Анureев И.С. Метод элиминации структур данных, основанный на системах переписывания формул // Программирование. — 1999. — № 4. — С. 5–15.
9. Анureев И.С. Язык Atoment: стандартная библиотека. — Новосибирск, 2010. — 47 с. — (Препр./РАН. Сиб. отд-ние. ИСИ; № 158).
10. Анureев И.С. Операционно-онтологический подход к формальной спецификации языков программирования // Программирование. — 2009. — № 1. — С. 35–42.

И.С. Ануреев

ЯЗЫК ATOMENT: СИНТАКСИС И СЕМАНТИКА

Препринт
157

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 2,23 уч.-изд.л., 2,38 п.л.
Тираж 60 экз.

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