ЯЗЫК ОПИСАНИЯ ВИЗУАЛИЗАТОРОВ АЛГОРИТМОВ

Г.А. Корнеев, А.А. Шалыто

В статье предлагается язык описания логики визуализаторов алгоритмов, основанный на XML. Этот язык позволяет автоматизировать процесс построения логики визуализаторов алгоритмов дискретной математики.

Введение

Визуализаторы алгоритмов широко применяются для обучения дискретной математике и доказали свою эффективность [1, 2]. При этом часто используются визуализаторы, создаваемые студентами [3].

Основной проблемой при разработке визуализаторов алгоритмов является обеспечение возможности просматривать алгоритм не только в прямом, но и в обратном направлениях [4]. При этом много ошибок допускается именно при реализации «обратного хода». Таким образом, автоматизация создания логики визуализаторов алгоритмов, обеспечивающей трассировку алгоритма как в прямом, так и в обратном направлении, весьма полезна.

В настоящей работе предлагается язык описания логики визуализаторов алгоритмов на основе расширяемого языка разметки *XML* (*Extensible Markup Language* [5]). Применение этого языка позволяет автоматизировать разработку логики визуализаторов алгоритмов, что реализовано в проекте *Vizi* [6].

Концепции языка

В рамках предлагаемого языка алгоритм рассматривается как набор процедур и глобальных переменных. Среди процедур выделяется главная процедура, запускаемая при выполнении алгоритма.

Процедура рассматривается как последовательность операторов следующих типов:

- оператор присваивания;
- оператор ветвления;
- оператор цикла с предусловием;
- оператор вызова процедуры;
- блочный оператор.

Каждому типу оператора, кроме блочного, соответствует XML-элемент. Блочные операторы кодируются неявно.

Отметим, что операторы вызова процедуры позволяют задавать рекурсивные алгоритмы как с явной, так и с косвенной рекурсией. Оба этих случая обрабатываются корректно.

Операторы могут использовать как глобальные переменные, определенные на уровне алгоритма, так и локальные переменные, определенные в рамках процедуры.

Для шага может быть указан его уровень, шаблон комментария и связь с визуальным представлением.

Уровень шага определяет, в каком случае визуализатор останавливается при исполнении данного шага. Уровень задается целым числом. При этом значение -1 обозначает пропуск шага, 0 – остановку при просмотре алгоритма маленькими шагами, 1 – остановку при просмотре алгоритма большими шагами.

Шаблон комментария определяет вид и параметры комментария, отображаемого на данном шаге.

Связь с визуальным представлением позволяет обновлять визуальное представление при отображении шага.

На рис. 1 приведена диаграмма XML-элементов, применяемых при описании логики визуализаторов. Диаграмма основана на диаграмме классов *UML* [7]. При этом классы соответствуют XML-элементам, а их атрибуты – атрибутам элементов. Атрибуты, имеющие стереотип «text», соответствует вложенному тексту элемента.

Абстрактные элементы введены для отображения общности элементов. При этом конкретные XML-элементы им не соответствуют.

Описание алгоритма

Как следует из рисунка, описание алгоритма задается элементом `algorithm`, содержащим описания процедур (элемент `auto`), глобальных переменных (элемент `variable`), а также вспомогательных конструкций (элементы `import`, `toString` и `method`).

Описание процедур будет рассмотрено ниже.

Описание глобальных переменных. Для каждой глобальной переменной указывается ее описание (атрибут `description`), имя (`name`), тип (`type`) и значение по умолчанию (`value`).

Описание переменной применяется при отладке алгоритма и не используется при генерации логики визуализатора [6].

Имена переменных должны быть корректными *Java*-идентификаторами и не содержать символов подчеркивания.

Значение по умолчанию применяется при автоматизированной проверке корректности построения логики визуализатора [6].

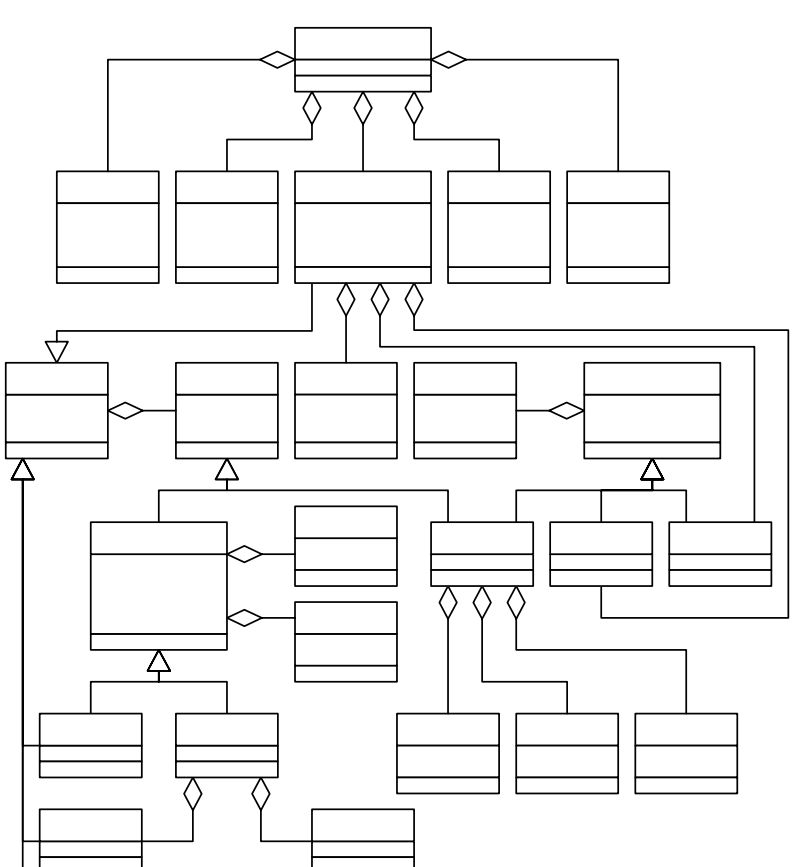

Рис. Диаграмма элементов

1 «element»
algorithm

Описание вспомогательных конструкций. Элемент `import` позволяет описывать связь визуализатора с другими типами аналогично конструкции `import` языка *Java* [8]. Имя импортируемого класса (пакета) задается обычным образом и записывается внутри элемента `import`.

Элемент `toString` задает метод, представляющий значения основных переменных визуализатора в виде строки. Этот метод используется при автоматизированной проверке корректности построения логики визуализатора [6]. При этом значения, выдаваемые этим методом при прямом и обратном проходах, сравниваются, и при их неравенстве выдается сообщение об ошибке.

Элемент `method` позволяет задавать невизуализируемые методы, которые могут, например, применяться для построения визуального представления.

Для метода указывается сигнатура метода (атрибут `header`), комментарий (`comment`) и тело метода (текст элемента). Сигнатура и тело метода записываются также как в языке *Java*.

Описание процедур

Описание процедуры задается элементом `auto`. При этом указываются имя (атрибут `name`) и описание (`description`).

В рамках процедуры описываются локальные переменные, начальное и конечное состояния, а также шаги алгоритма.

Описание локальных переменных. Локальные переменные описываются элементом `auto`. Для каждой локальной переменной указываются ее описание (атрибут `description`), имя (`name`) и тип (`type`).

Отметим, что для локальных переменных значения по умолчанию не указываются. Таким образом, исходное значение локальной переменной не определено.

Описание начального и конечного состояний. Описание начального (элемент `start`) и конечного (элемент `finish`) состояний позволяет задать комментарий и визуальное преставление, отображаемые пользователю при входе и выходе из процедуры. Они обычно применяются в главной процедуре алгоритма.

Комментарий задается шаблоном комментария (атрибут `comment`) и параметрами комментария (`comment-args`).

Шаблон комментария содержит ссылки на параметры, записываемые в виде

`{`*`номер аргумента`*`}`

Параметры комментария перечисляются через запятую. Каждый параметр представляет собой выражение, вычисляемое в момент отображения комментария. Таким образом, комментарии могут включать в себя значения переменных алгоритма.

Связь с визуальным представлением указывается во вложенном элементе `draw`, содержащем код, исполняемый при отображении визуального представления пользователю. Таким образом, визуальное представление может зависеть не только от текущего состояния, но и от значений переменных.

Описание операторов

С точки зрения удобства визуализации операторы (абстрактный элемент `statement` на рисунке) разделены следующим образом:

- неотображаемые операторы;
    - оператор вызова процедуры;
    - блочный оператор (абстрактный элемент `block` на рисунке).
- отображаемые операторы:
    - оператор присваивания;

- условные операторы (абстрактный элемент `condition` на рисунке):
- оператор ветвления;
- оператор цикла с предусловием.

Оператор вызова процедуры. Оператор вызова процедуры описывается элементом `call-auto`. При этом указывается имя вызываемой процедуры (атрибут `id`).

Блочный оператор. Предлагаемый язык не содержит выделенной конструкции для описания блочных операторов. Вместо этого тело процедуры, цикла с предусловием и ветвей оператора ветвления являются блочными операторами.

Отображаемые операторы. Для отображаемых операторов указываются идентификаторы (атрибут `id`), описание (`description`) и уровень оператора (`level`), указывающий остановится ли визуализатор на данном операторе.

Значение уровня по умолчанию равно нулю.

Оператор присваивания. В операторе присваивания могут быть изменены значения одной или нескольких переменных.

Для трассировки в обратном направлении требуется осуществить обращение операторов – построить код, исполняемый при обратном проходе [4].

Оператор присваивания может обращаться либо автоматически, либо вручную. Ручное обращение иногда позволяет сэкономить память и/или время по сравнению с автоматическим выполнение этой операции [4].

При автоматическом обращении выполняемые действия записываются в элементе `action`. При этом для изменения переменных применяются операторы обратимого присваивания, имеющие вид `@=`. При обратном проходе все изменения, выполненные операторами обратимого присваивания, автоматически «откатываются» [4].

В случае ручного обращения отдельно указываются действия, выполняемые при прямом (элемент `direct`) и обратном (элемент `reverse`) проходах. В этом случае проверка верности обращения может быть автоматизирована [6].

В случае отображения пользователю для оператора присваивания указываются комментарий и связь с визуальным представлением, как это было указано выше для начального и конечного состояний.

Условные операторы. Для условного оператора указывается условие (атрибут `test`), позволяющее выбрать следующий оператор при прямом проходе.

Выбор следующего оператора при обратном проходе может осуществляться как автоматически, так и вручную. Во втором случае соответствующее условие указывается в атрибуте `rtest`.

Для условных операторов также указываются комментарии и связь с визуальным представлением. При этом шаблоны комментариев и действия по обновлению визуального представления указываются отдельно для истинного (атрибут `comment-true` и элемент `draw-true`) и ложного (`comment-false`, `draw-false`) значений условия.

Оператор ветвления задается элементом `if`, в который вложены описания ветвей, исполняемых при истинности (элемент `then`) и ложности (`else`) условия. Для каждой ветви указывается набор операторов, составляющих эту ветвь.

Отметим, что в случае укороченного оператора ветвления ветвь `else` опускается.

Оператор цикла с предусловием задается элементом `while`, в который вложены описания операторов, составляющих тело цикла.

Переменные

При записи описания логики визуализатора переменные делятся на глобальные и локальные. Глобальные переменные доступны во всех процедурах, а локальные – только в той процедуре, в которой они объявлены.

Для доступа к переменным используется @-нотация. При этом настоящее имя переменной не используется в явном виде, а подставляется автоматически.

Доступ к переменной осуществляется при помощи выражения вида

```
@<имя переменной>
```

Например, оператор

```
@max = @a[@i]
```

выполняет присваивание переменной `max` значения `i`-го элемента массива `a`. Заметим, что при наличии в области видимости глобальной и локальной переменных с одним и тем же именем используется локальная переменная.

Для создания строкового представления автомата в процедуре `toString` введен синтаксис, позволяющий производить доступ к локальным переменным других процедур. Доступ к переменной, объявленной в другой процедуре, осуществляется следующим образом:

```
@<имя процедуры>@<имя переменной>
```

Например, оператор

```
buffer.append(@Main@i);
```

осуществляет добавление к буферу значения локальной переменной `i` процедуры `Main`.

Для доступа к переменным модели из кода визуализатора применяются для глобальных переменных выражения вида

```
<переменная модели>.<имя переменной>
```

а для локальных переменных выражение вида

```
<переменная модели>.<имя процедуры>_<имя переменной>
```

Например:

```
data.max = 0;
System.out.println(data.Main_i);
```

Пример

Рассмотрим описания логики визуализатора алгоритма поиска максимума в массиве натуральных чисел.

Данная задача может быть решена следующей программой:

```
void main() {
  int max = 0;
  for (int i = 0; i < a.length; i++) {
    if (max < a[i]) {
      max = a[i];
    }
  }
}
```

Здесь `a` – массив, в котором производится поиск максимума, а `max` – значение текущего максимума (после `i`-ой итерации – среди первых `i` элементов).

Отметим, что инициализация максимума нулем не приводит к ошибке, так как по условию задачи в массиве содержатся только натуральные числа.

На предложенном языке данный алгоритм может быть записан следующим образом:

```
1:  <algorithm>
2:    <variable name="a" type="int[]" value="new int[]{1, 2, 3, 1,
              6}" description="Массив для поиска"/>
3:    <variable name="max" type="int" value="0"
              description="Текущий максимум"/>
4:    <auto id="Main" description="Ищет максимум в массиве">
5:      <variable description="Переменная цикла" name="i"
              type="int"/>
6:      <start comment-ru="На экране изображен массив, в котором
              будет осуществляться поиск максимума">
7:        <draw>@visualizer.updateArray(0, 0);</draw>
```

```
 8:      </start>
 9:      <step id="Initialization" description="Инициализация"
                comment-ru="Инициализируем максимум нулем (так как в
                массиве только натуральные числа).">
10:       <draw>@visualizer.updateArray(0, 0);</draw>
11:       <action>@max @= 0;</action>
12:     </step>
13:     <step id="LoopInit" description=":" level="-1">:</step>
14:     <while id="Loop" description="Цикл" test="@i &lt; @a.length"
                rtest="@i &gt;= 0" level="-1">
15:       <if id="Cond" description="Условие" test="@max &lt; @a[@i]"
                true-comment-ru="{0} больше текущего максимума ({1})"
                false-comment-ru="{0} не больше текущего максимума
                ({1})"
                comment-args="new Integer(@a[@i]), new Integer(@max)">
16:         <draw>@visualizer.updateArray(@i, 1);</draw>
17:         <then>
18:           <step id="newMax" description="Обновление максимума"
                comment-ru="Обновляем текущий максимум">
19:             <draw>@visualizer.updateArray(@i, 2);</draw>
20:             <action>@max @= @a[@i];</action>
21:           </step>
22:         </then>
23:       </if>
24:       <step id="inc" description=":" level="-1">:</step>
25:         <forward>@i = @i + 1;</forward>
26:         <reverse>@i = @i - 1;</reverse>
27:       </step>
28:     </while>
29:     <finish comment-ru="Максимум найден ({0})"
                comment-args="new Integer(@max)">
30:       <draw>@visualizer.updateArray(0, 0);</draw>
31:     </finish>
32:   </auto>
33: </algorithm>
```

Рассмотрим, некоторые части этого описания подробнее.

Данный алгоритм содержит две глобальные переменные: `a` – массив, в котором осуществляется поиск (объявлена во второй строке) и `max` – текущее значение максимума (объявлена в третьей строке).

В четвертой строке начинается описание основной процедуры `main`, которое заканчивается в строке 31. В процедуре объявлена локальная переменная – индекс текущего элемента массива (строка 4).

За описанием локальной переменной следует описание начального состояния (строки 6–8), включающее комментарий (строка 6) и связь с визуальным представлением (строка 7).

Далее приводится описание шагов алгоритма, начинающееся операторами присваивания начальных значений переменным `max` (строки 9–12) и `i` (строка 13). Отметим, что первое из этих присваиваний визуализируется, а второе – нет, так как для него указан уровень `-1`.

В строках с 14 по 28 описан оператор цикла с предусловием. Отметим, что для него применяется обращение вручную. Соответствующее условие указанно в атрибуте `rtest`.

Тело оператора цикла включает два оператора: ветвления (строки 15–23) и присваивания (строки 24–27). При этом оператор присваивания использует косвенное обращение: в строке 25 указано действие, осуществляемое при прямом проходе, а в строке 26 – действие, выполняемое при обратном проходе.

Заключение

Предложенный язык описания логики визуализаторов позволяет записывать логику визуализатора в привычной форме, дополняя ее описанием комментариев и связью с визуальным представлением. По такому описанию логика визуализатора может быть сгенерирована автоматически [6].

Автоматизация процесса построения логики визуализатора позволяет реализовывать визуализаторы таких сложных алгоритмов, как, например, алгоритм Укконена построения суффиксного дерева [9] или алгоритм построения кратчайшего дерева путей в графе [10], что практически невозможно при ручном подходе.

Работа выполнена в совместной лаборатории СПбГУ ИТМО и центра разработки Borland «Технологии программирования».

Литература

1. Byrne M., Catrambone R., Stasko J. Evaluating Animations as Student Aids in Learning Computer Algorithms // Computers & Education. 1999. Vol. 33. № 4. P. 253–278.
2. Казаков М.А., Столяр С.Е. Визуализаторы алгоритмов как элемент технологии преподавания дискретной математики и программирования / Международная научно-методическая конференция «Телематика–2000». СПб: СПбИТМО (ТУ), 2000. С. 189–191.
3. Stasko J. Using Student-Built Algorithm Animations as Learning Aids / Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE '97), CA, 1997, pp. 25–29.
4. Корнеев Г.А., Казаков М.А., Шалыто А.А. Метод построения логики работы визуализатора алгоритмов на основе конечных автоматов. // Телекоммуникации и информатизация образования. 2003. № 6. С. 27–58. http://is.ifmo.ru/works/vis/
5. Extensible Markup Language (XML) 1.1. http://www.w3.org/TR/xml11/.
6. Сайт проекта Vizi // http://ctddev.ifmo.ru/vizi
7. Буч Г., Якобсон А., Рамбо Дж. UML. 2-е издание. СПб: Питер, 2006. 736 с.
8. Joy B., Steele G., Gosling J., Bracha G. Java Language Specification, Second Edition. NJ.: Addison-Wesley. 2000. http://java.sun.com/docs/books/jls/
9. Ахметов И.Р. Разработка визуализатора алгоритма Укконена построения суффиксных деревьев на основе технологии Vizi. http://is.ifmo.ru/vis/ukkonen/
10. Пименов С.Ю., Корнеев Г.А., Шалыто А.А. Алгоритм Чу Йонджина и Лю Цзенхонга построения кратчайшего корневого дерева в ориентированном графе. http://is.ifmo.ru/vis/ctree/