УДК 681.3.06

АВТОМАТИЧЕСКАЯ ГЕНЕРАЦИЯ АВТОМАТНОГО КОДА

С. Ю. Канжелев,
магистрант
А. А. Шалыто,
доктор техн. наук, профессор
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Рассматривается подход к автоматической генерации кода автоматных программ на любом априори заданном языке программирования по графам переходов автоматов.

The approach to the automatic automata's program code generation is considered in the paper. The code is generated in any programming language by the automaton's transition graph.

Если ты ленив и упорен,
то ты непременно чего-нибудь добьешься.
Жорж Фейдо

Введение

В последнее время весьма актуален вопрос о визуальном конструировании программ [1], в частности, проблема автоматической генерации кода по графическим моделям. Модели можно разделить на два класса — статические и динамические. Среди статических моделей объектно-ориентированных программ основной является диаграмма классов. Многие инструментальные средства генерируют «скелет» кода по диаграммам классов.

Динамические свойства рассматриваемого класса программ наиболее эффективно описывают диаграммы состояний (графы переходов). Известны различные инструментальные средства, генерирующие код по графам переходов [2–5]. Указанные средства являются специализированными — строят код для одного-двух языков программирования.

Цель настоящей работы состоит в описании подхода к генерации кода автоматных программ (программ с явно выделенными состояниями) на любом априори заданном языке программирования.

На основе этого подхода было создано инструментальное средство MetaAuto [6], которое позволяет преобразовывать изображения графов переходов, представленных с помощью редактора MS Visio, в исходные коды программ на любых языках программирования, для которых предварительно созданы соответствующие шаблоны. В ходе работы над этим инструментальным средством в качестве примера были созданы шаблоны для трех языков программирования (C++, C# и Turbo Assembler).

Этапы генерации кода

Декомпозиция задачи. Как отмечалось выше, описываемый подход предполагает построение исходного кода на различных языках программирования. Для реализации этого требования разумно разделить этап преобразования графической модели в исходный код на два: преобразование графической модели в некий общий формат представления графа переходов и преобразование из этого формата в различные языки программирования. Такое разделение приводит к большей платформенной независимости. При этом легко перейти от одного редактора графов переходов на другой, реализовав для нового редактора процедуру преобразования в общий формат. Также общий формат представления графа переходов может быть использован в валидаторах — программах проверки корректности графов переходов.

Для облегчения работы с данными, содержащимися в графе переходов, имеет смысл использовать программное представление общего формата. Под программным представлением понимается библиотека классов, изоморфная по своей структуре данным общего формата, которая предназначена для упрощения процесса чтения и сохране-

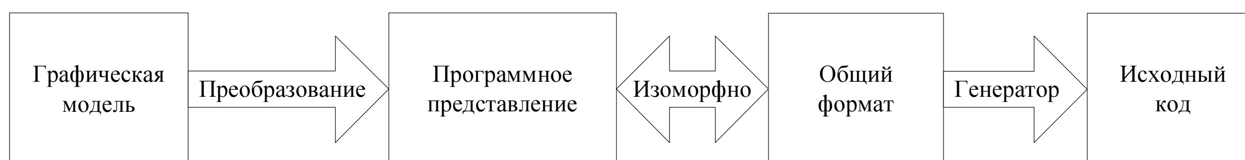


Рис. 1. *Декомпозиция процесса генерации исходного кода*

ния данных различными программами: валидаторами, генераторами кода, другими программами пользователя.

Таким образом, этап генерации кода может быть разбит на три: преобразование графа переходов в программное представление, затем изоморфное преобразование программного представления в общий формат и, наконец, преобразование из общего формата в исходный код программы. На рис. 1 изображены этапы генерации исходного кода программы. По этой схеме построено разработанное инструментальное средство MetaAuto [6].

Генерация кода. В рамках подхода к генерации исходного кода программы необходимо осуществлять преобразование графа переходов автомата, представленного в некотором общем формате, в исходный код программы. В общем случае граф переходов можно рассматривать как набор метаданных определенной структуры из предметной области решаемой задачи. Вопрос об автоматической генерации кода с заданными свойствами при этом является одним из вопросов, решаемых в рамках порождающего программирования (Generative Programming) [7].

В настоящее время в рамках порождающего программирования разработан ряд способов генерации кода. Среди них можно выделить: подстановки; подстановки с исполнением кода; обработчики данных регулярной структуры.

Генерация кода, основанная на подстановках, предполагает, что разработчик создает шаблон кода и набор данных в специальном формате, а затем, с помощью вспомогательной программы, выполняет подстановку этих данных в шаблон. Набор используемых в шаблоне подстановок может определяться как статически, так и динамически. Такой подход нагляден и прост в использовании, однако имеет весьма ограниченную область применения и требует предварительной подготовки передаваемых для подстановки данных. Классический пример подстановок (с некоторыми оговорками) — шаблоны (templates) языка C++.

Гораздо шире возможности при *генерации кода с использованием подстановки с исполнением кода*. Этот вид генерации отличается от предыдущего возможностью применять в шаблоне не только подстановки, но также и вставки исполняемого кода, оперирующего переданными в шаблон данными. Исполняемый код чаще всего использует язык, который специально создан для конкретного типа шаблонов и включает основные алгоритмические конструкции, такие как, например, простое ветвление (IF), выполнение итераций по переданным в качестве параметров спискам (WHILE), циклы с заданным количеством итераций (FOR).

Отметим, что в процессе усложнения конструкций, применяемых в шаблоне, и увеличения возможностей языка уменьшается наглядность самих шаблонов.

Пример использования такого способа генерации, где применяется технология ASP, первоначально разработанная для генерации HTML-страниц, приведен в работе [8]. В качестве языка исполняемого кода в указанной технологии используется язык Visual Basic.

Недостатком подходов к генерации кода, основанных на подстановках и подстановках с исполнением кода, является необходимость специальной подготовки данных для передачи в шаблон.

Третий способ генерации кода основан на *обработчиках данных регулярной структуры*; он предполагает полное разделение данных и их представления. В этом случае шаблон играет роль обработчика данных и пишется на специальном метаязыке. Примером может служить XSLT-обработка данных, представленных в XML-формате.

Вопрос генерации кода на основе XSLT-преобразований рассматривается в работах [9–11].

Основным достоинством этого способа генерации кода является возможность обработки данных сложной структуры без предварительной подготовки этих данных.

Существование перечисленных видов генерации кода обусловлено неоднородностью решаемых в рамках порождающего программирования задач и структуры метаданных. Вопрос об использовании того или иного способа генерации зависит от условий конкретной задачи и данных, используемых для такой генерации.

Особенности генерации исходного кода для графов переходов. Одним из вопросов, решаемых при преобразовании графов переходов автоматов в исходные коды программ, является вопрос о реализации групповых переходов [12]. Чаще всего автоматная процедура (процедура, реализующая поведение автомата), состоит из одного оператора switch. Групповые переходы в этом случае реализуются с помощью дублирования их для всех состояний, содержащихся в группе, к которой данный групповой переход относится.

Также непростой задачей является генерация логических выражений. Например, код програм-

мы, реализующей вычисление значения логических выражений для языка Turbo Assembler, сильно отличается от кода программы, реализующей вычисление логических выражений для языков высокого уровня, в которых логические выражения часто записываются одной формулой.

Большое количество рекурсивных структур и другие особенности графов переходов определяют выбор способа генерации исходного кода.

Инструментальное средство MetaAuto использует XSLT-технологию для генерации исходного кода. В терминах рис. 1 в качестве общего формата в MetaAuto для представления графов переходов выбран XML-формат. Генератор для преобразования общего формата в исходный код программы базируется на XSLT-шаблонах.

XSLT-преобразование. XSLT (*e*X*tensible* S*tylesheet* L*anguage* T*ransformations* — расширяемый язык стилей для преобразований) в последнее время стал популярной XML-технологией. Спецификация определяет XSLT как язык для преобразований одних XML-документов в другие XML-документы. Однако за время своего существования XSLT стал использоваться значительно шире, например для автоматической генерации кода.

В работах [9–11] рассматривается вопрос применения языка XSLT в качестве инструмента для генерации кода программ. В работе [9] отмечены основные достоинства и недостатки генерации кода с использованием этого языка.

Достоинства:

— широкие возможности по изменению шаблонов без изменения любой другой функциональности;

— широкие возможности по манипулированию данными; возможность извлекать данные из дополнительных источников;

— возможность изменять язык программирования с помощью небольшого изменения шаблона;

— наглядность и широкое распространение XML-формата; нет необходимости каждый раз придумывать новый формат хранения данных.

Недостатки:

— трудно контролировать отступы и пробелы при генерации текста; небольшое количество функций работы со строками (спецификация XSLT 2.0 призвана исправить эти недостатки, однако на данный момент трудности существуют);

— язык XSLT не предоставляет прямого доступа к операционной системе, это несколько ограничивает возможности по генерации платформенно-специфичного кода;

— генерация нескольких исходящих документов в рамках спецификации XSLT 1.0 невозможна.

В работах, посвященных генерации кода с помощью XSLT-преобразования, код генерируется, как правило, по данным, имеющим линейную структуру, например по структуре таблиц базы данных [10] или по списку классов UML-подобных диаграмм [11]. При применении метаданных линейной структуры сложно оценить все преимущества использования XSLT-преобразования. Разработанное инструментальное средство MetaAuto демонстрирует применение XSLT-преобразований для метаданных с нелинейной структурой.

Конфигурирование. Инструментальное средство MetaAuto позиционируется как универсальное. Поэтому оно должно уметь «понимать» различные обозначения, применяемые в графах переходов. Необходимо иметь возможность настройки:

— синтаксиса, используемого для задания списков действий в состояниях и на переходах;

— применяемых в этих списках символов-разделителей;

— формата входных и выходных переменных;

— использующихся в логических выражениях обозначений.

Возможность конфигурирования такого рода основана на применении регулярных выражений — мощного инструмента для задания синтаксиса. Преимуществами применения регулярных выражений в качестве инструмента конфигурирования графов переходов являются:

— гибкость задания синтаксиса;

— широкое распространение регулярных выражений и наличие большого количества вспомогательных инструментов для их создания;

— большие дополнительные возможности, такие как, например, задание любого количества именованных областей.

Разработанное инструментальное средство включает в себя стандартный конфигурационный файл, описывающий обозначения графов переходов из работы [13]. Однако имеется возможность задания собственного файла конфигурации.

Рассмотрим пример создания такого файла. Создадим конфигурационный файл, поддерживающий названия входных переменных в стиле инструментального средства UniMod [5].

В графах переходов, создаваемых с помощью этого инструментального средства, входные переменные, как и выходные, имеют источник — класс, наследуемый от класса `ControlledObject` (объект управления), со следующим синтаксисом:

```
<название объекта управления>.<название входной/
                                  выходной переменной>
```

Например, если название объекта управления `o1`, а название входной переменной `x10`, то на графе перехода эта переменная изображается как `o1.x10`.

Такой способ наименования входных переменных не был предусмотрен в нотации графов переходов из работы [13].

Зададим формат таких входных переменных в конфигурационном файле. Предположим, что, как и в языке Java (использующемся в UniMod), названия объекта управления и входных переменных состоят из букв, цифр, знака подчеркивания и некоторых других дополнительных символов. Тогда синтаксис для каждой такой входной пере-

менной можно описать с помощью регулярных выражений вида

```
(\w+)\.(\w+)
```

Здесь `\w` означает допустимый в названии символ. Если применять в регулярном выражении именованные области, то синтаксис можно описать следующим образом:

```
(?<controlledObjectName>\w+)\.(?<inputVariableName>\w+)
```

При этом первая именованная область имеет имя `controlledObjectName` и соответствует множеству допустимых символов до точки, а вторая (с именем `inputVariableName`) — множеству допустимых символов после точки.

В процессе проверки на соответствие входной строки данному регулярному выражению, при положительном ее исходе, можно извлечь название объекта управления (именованная область `controlledObjectName`) и название входной переменной (именованная область `inputVariableName`).

Фрагмент конфигурационного файла, представленный в листинге 1, задает синтаксис входной переменной в стиле инструментального средства UniMod.

Листинг 1. *Фрагмент конфигурационного файла*

```
<nodeTemplate  template="INPUT_VARIABLE"
   regexp="(?<controlledObjectName>\w+)\.(?<inputVariableName>\w+)"
   name="o${controlledObjectName}x${inputVariableName}"
   type="INPUT_VARIABLE"  >
   <parameter  name="object"
              value="${controlledObjectName}"  />
   <parameter  name="variable"
              value="${inputVariableName}"  />
</nodeTemplate>
```

Этот фрагмент задает шаблон для разбора входных переменных вида `o1.x10`. Шаблон называется `INPUT_VARIABLE`. Он использует именованные области для генерации уникального имени, а также создания двух параметров `object` и `variable`.

При применении данного фрагмента в конфигурационном файле для входной переменной `o1.x10` получим объект программного представления модели (библиотеки MetaAuto) со значениями свойств, приведенными в листинге 2, а также изоморфное ему XML-представление, записанное в том же листинге.

Листинг 2. *Полученный объект библиотеки MetaAuto и его XML-представление*

```
Type  =  "INPUT_VARIABLE"
Name  =  "o1x10"
Parameters[0].Name  =  "object"
Parameters[0].Value  =  "o1"
Parameters[1].Name  =  "variable"
Parameters[1].Value=  "x10"

<actionNode  name="o1x10"  type="INPUT_VARIABLE">
   <parameter  name="object"  value="o1"  />
   <parameter  name="variable"  value="x10"  />
</actionNode>
```

Методика генерации исходного кода

Опишем методику генерации исходного кода с применением разработанного инструментального средства. Будем преобразовывать граф переходов из работ [4, 13] в исходный код программы на языке С++.

Шаг 1. Изображение графов переходов. Изобразим этот граф переходов с помощью редактора MS Visio (рис. 2).

Для изображения графа переходов следует использовать специально созданный шаблон, содержащий все используемые на графе переходов типы элементов. Этот шаблон предоставляется вместе с инструментальным средством MetaAuto. Он включает в себя такие элементы, как «состояние», «переход», «описание автомата» и т. д.

Шаг 2. Преобразование в XML-формат. Изображенный на первом шаге граф переходов преобразуется в XML-формат. Этот шаг включает в себя сразу два этапа (см. рис. 1): преобразование графического представления графа переходов в программное представление и изоморфное преобразование программного представления в XML-формат.

Для такого преобразования воспользуемся компонентой Visio2Xml.exe, являющейся частью инструментального средства. В качестве параметров передадим компоненте путь до файла, содержащего графическую модель — *analizers.vsd* и название итогового XML-файла. Для запуска из командной строки используем следующую командную строку:

```
Visio2Xml.exe  analizers.vsd  automatas.xml
```

Полученный XML-файл полностью описывает граф переходов.

Например, состояния и групповые состояния описываются с помощью узлов `state` XML-файла, вложенных друг в друга. Каждое состояние может включать список запускаемых в нем автоматов и выполняемых действий. На листинге 3 представлен фрагмент XML-файла.

Листинг 3. *Фрагмент полученного XML-файла. Описывает состояние «2. Предпусковые операции», вложенное в групповое состояние*

```
<state  name="group_0"  description="">
```

Состояние 2 вложено в групповое состояние group_0

```
 <state  name="2"  description="Предпусковые  операции">
```

Список вложенных автоматов (A: 4,3,1)

```
  <stateMachineRef>

     <actionNode  name="A4e0"
               type="SIMPLE_AUTOMATA_CALL_IN_STATE">
```

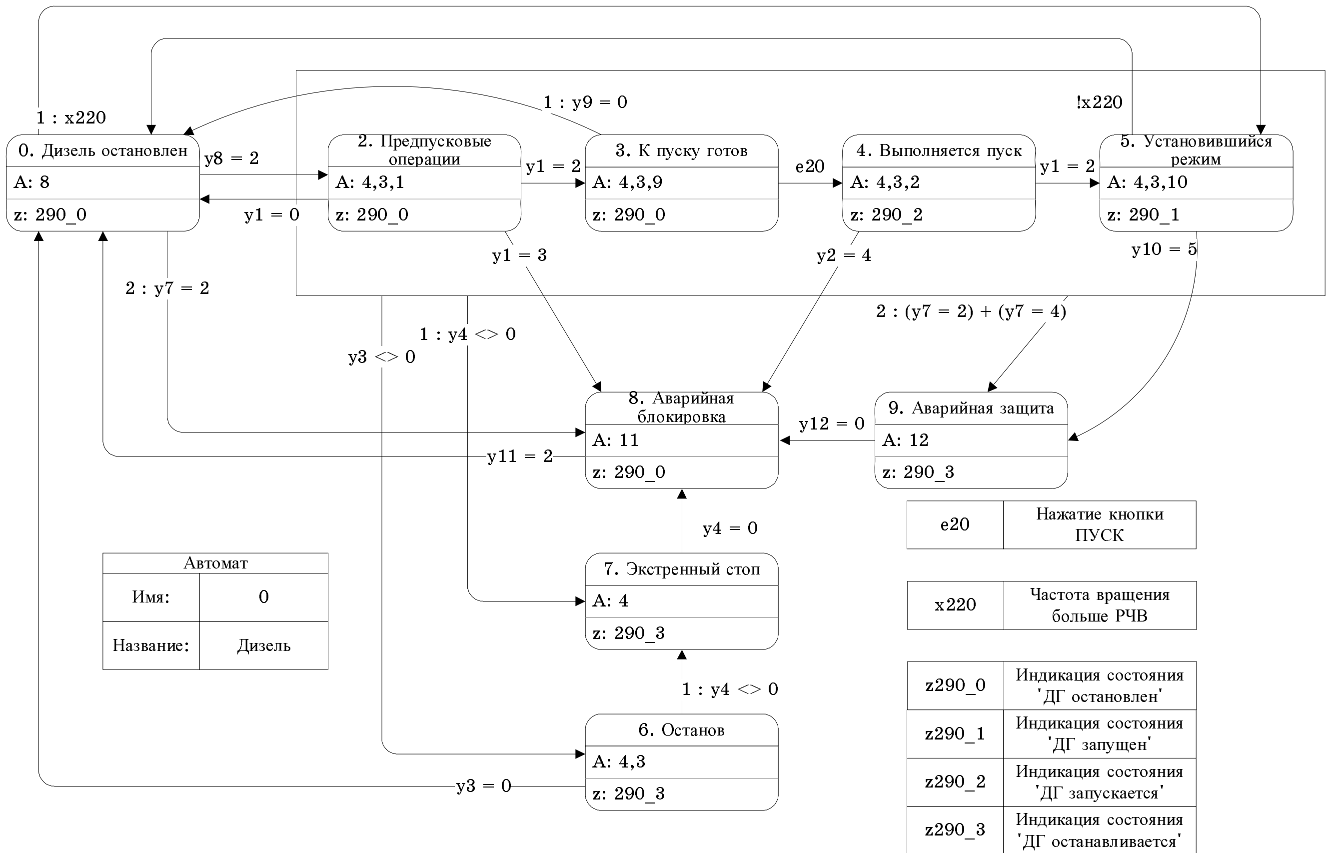
вызов автомата "4" …

```
      <parameter  name="automata"  value="4"  />
```

…с событием "0"

```
      <parameter  name="event"  value="0"  />
     </actionNode>

     <actionNode  name="A3e0"
             type="SIMPLE_AUTOMATA_CALL_IN_STATE">
```



■ ***Рис. 2.*** *Граф переходов автомата (страница A0 файла analizers.vsd)*

```
      <parameter name="automata" value="3" />
      <parameter name="event" value="0" />
    </actionNode>

    <actionNode name="A1e0"
              type="SIMPLE_AUTOMATA_CALL_IN_STATE">
      <parameter name="automata" value="1" />
      <parameter name="event" value="0" />
    </actionNode>

  </stateMachineRef>
```

Список выполняемых в состоянии действий (z: 290_0)

```
  <outputAction>
    <actionNode name="290_0"
                type="SIMPLE_OUTPUT_ACTION_IN_STATE">
      <parameter name="name" value="290_0" />
    </actionNode>
  </outputAction>
</state>
```

...

Описание остальных состояний, вложенных в групповое, пропущено

```
</state>
```

Переходы в полученном файле описываются с помощью узлов `transition` XML-файла. Вложенные в них узлы описывают условие перехода, а также описывают список действий на переходе. На листинге 4 приведен фрагмент XML-файла.

Листинг 4. *Фрагмент полученного XML-файла. Описывает групповой переход в состояние 9*

Переход из группового состояния group_0 в состояние 9 с приоритетом 2 2: (y7=2)+(y7=4)

```
<transition sourceRef="group_0" targetRef="9"
            priority="2">
```

Условие перехода (y7=2)+(y7=4)

```
  <condition>
```

Бинарная операция OR (обозначаемая +). Использует два вложенных узла conditionNode в качестве операндов

```
    <binaryOperation type="OR">
```

Проверка состояния автомата (y7=2)

```
      <conditionNode name="y7e2"
                     type="OTHER_AUTOMATA_EVENT_EQUAL">
        <parameter name="automata" value="7" />
        <parameter name="event" value="2" />
      </conditionNode>
```

Проверка состояния автомата (y7=4)

```
      <conditionNode name="y7e4"
                     type="OTHER_AUTOMATA_EVENT_EQUAL">
        <parameter name="automata" value="7" />
        <parameter name="event" value="4" />
      </conditionNode>
    </binaryOperation>
  </condition>
```

Далее следует список действий, выполняемых на переходе. Рассматриваемый групповой переход не со-

держит никаких действий. Формат списка действий идентичен формату списка действий в состоянии

```
</transition>
```

XML-файл содержит также и другие узлы. Эти узлы соответствуют описанию действий и входных переменных автомата, названию автомата и т. д. Весь XML-файл, а также подробное описание его формата приведены в работе [6].

Шаг 3. Создание XSLT-шаблона. Для получения кода на языке C++ необходимо создать соответствующий шаблон. Как правило, по одному графу переходов требуется построить сразу несколько файлов. Например, в языке C++ требуется создать файл заголовков *common.h* и файл реализации *common.cpp*. В таком случае необходимо создать шаблон для каждого такого файла.

Рассмотрим шаблон для создания файла реализации *common.cpp*, как более информативный. Назовем этот шаблон *common.cpp.xslt*. В листинге 5 приведена часть этого шаблона для генерации файла, содержащего автоматную процедуру.

Другая часть шаблона, отвечающая за вывод логического выражения и различных типов входных и выходных воздействий, приведена в работе [6]. Выполняемые в ней действия весьма стандартны. Примеры создания таких шаблонов можно найти практически в любом пособии по языку XSLT.

Листинг 5. *Шаблон common.cpp.xslt*

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method='text' indent="no"/>

<xsl:template match="/model">
```

«шапка» файла

```
//--- this file is machine generated ---
#include "StdAfx.h"
#include "common.h"
#include "log.h"

common_t cm;
```

Обрабатываем все автоматы в модели. Для каждого из них создаем автоматную процедуру с названием A<название автомата>

```
<xsl:for-each select="stateMachine">
void A<xsl:value-of select="@name"/>( ubyte e )
{
 ubyte y_old = cm.y<xsl:value-of select="@name"/>;
 switch( cm.y<xsl:value-of select="@name"/> )
 {
```

Рассматриваем все конечные — не групповые — состояния. Условие `count(state) = 0` показывает, что выборка будет производиться именно по таким состояниям

```
  <xsl:for-each select="//state[count(state) = 0]">
   <xsl:sort select="@name"/>
```

Вызываем вложенные автоматы

```
   case <xsl:value-of select="@name"/>:
   <xsl:for-each select="stateMachineRef/actionNode">
     A<xsl:value-of select="parameter
                        [@name='automata']/@value"/>(e);
   </xsl:for-each>
```

Выбираем все переходы из данного состояния и переходы из групповых состояний, содержащих данное состояние. Из всех состояний текущего конечного автомата `ancestor::stateMachine/transition` выбираем те, у которых атрибут `sourceRef` совпадает с именем текущего состояния или состояния, содержащего текущее `current()/ancestor-or-self::state/@name`

```
   <xsl:apply-templates
        select="ancestor::stateMachine/transition
              [current()/ancestor-or-self::state/
               @name = sourceRef]">
     <xsl:sort select="count(condition)"
              data-type="number" order="descending"/>
     <xsl:sort select="string-length(@priority) = 0"
             data-type="text" order="ascending"/>
     <xsl:sort select="@priority" data-type="number"/>
     <xsl:sort select="@targetRef" data-type="text"/>
   </xsl:apply-templates>

   break;
  </xsl:for-each>

   default:
     #ifdef A<xsl:value-of select="@name"/>_ERRORS_LOGGING
       log_write(LOG_GRAPH_ERROR, "Unknown state!");
     #else
       ;
     #endif
  }

 if( y_old == cm.y<xsl:value-of select="@name"/> )
   goto A<xsl:value-of select="@name"/>_end;
```

В случае, если состояние автомата изменилось, вызываем вложенные автоматы и выполняем действия в состоянии

```
 switch( cm.y<xsl:value-of select="@name"/> )
 {
  <xsl:for-each select="//state[count(state) = 0]" >
   <xsl:sort select="@name" />
    case <xsl:value-of select="@name"/>:
     <xsl:apply-templates
            select="stateMachineRef/actionNode" />
     <xsl:apply-templates
            select="outputAction/actionNode" />
    break;
   </xsl:for-each>
  }
 A<xsl:value-of select="@name"/>_end: ;
}
</xsl:for-each>
```

Далее следуют шаблоны для вывода логических выражений, действий в состоянии и на переходе. Полностью эти шаблоны приведены в работе [6]

Шаг 4. Получение исходного кода. Преобразуем полученный на втором шаге XML-файл в исходный код с помощью XSLT-шаблона, созданного на предыдущем шаге. Воспользуемся компонентой *XslTransform.exe* инструментального средства MetaAuto. Данной компоненте необходимо передать в качестве параметров путь до XML-файла *(automatas.xml)*, путь до шаблона *(common.cpp.xslt)* и название итогового файла с исходным кодом *(common.cpp)*.

Для запуска компоненты воспользуемся следующей командной строкой:

XslTransform.exe automatas.xml common.cpp.xslt common.cpp

В результате получим файл *common.cpp*, содержащий исходный код на языке С++ , который реализует граф переходов (см. рис. 2) и представлен в листинге 6 в два столбца.

Листинг 6. *Исходный код автоматной процедуры на языке С++ . Фрагмент файла common.cpp, содержащего код программы*

«шапка» файла

```
//— this file is machine generated
#include "StdAfx.h"
#include "common.h"
#include "log.h"
common_t cm;
```

Автоматная процедура автомата 0

```
void A0( ubyte e )
{
 ubyte y_old = cm.y0;

 switch( cm.y0 )
 {
```

Переходы из состояния 0 в состояния 0 по условию `x220()`, в состояние 8 по условию `cm.y7 == 2` и в состояние 2 по условию `cm.y8 == 2`

```
  case 0:
   A8(e);
   if ( x220() )
   {
    cm.y0 = 5;
   }
   else if ( cm.y7 == 2 )
   {
    cm.y0 = 8;
   }
   else if ( cm.y8 == 2 )
   {
    cm.y0 = 2;
   }
   break;
```

Переходы из состояния 2 в состояния 7, 9, 0, 3, 6 и 8

```
  case 2:
   A4(e); A3(e); A1(e);
   if ( cm.y4 != 0 )
   {
    cm.y0 = 7;
   }
   else
   if ((cm.y7==2)||(cm.y7==4))
   {
    cm.y0 = 9;
   }
   else if ( cm.y1 == 0 )
   {
    cm.y0 = 0;
   }
   else if ( cm.y1 == 2 )
   {
    cm.y0 = 3;
   }
   else if ( cm.y3 != 0 )
   {
    cm.y0 = 6;
   }
   else if ( cm.y1 == 3 )
   {
    cm.y0 = 8;
   }
   break;
```

Часть switch-блока для других состояний пропущена

```
 default:
  #ifdef A0_ERRORS_LOGGING
   log_write(LOG_GRAPH_ERROR, "Unknown state!");
  #else
   ;
  #endif
}
```

Выполнение действий в стостоянии и запуск вложенных автоматов

```
 if( y_old == cm.y0 )
  goto A0_end;

 switch( cm.y0 )
 {
   case 0:
    A8(0);
   break;
   case 2:
    A4(0);
    A3(0);
    A1(0);
   break;
```

Часть switch-блока для других состояний пропущена

```
 }
```

Далее следует остальной код

Шаг 5. Преобразование XML-файла в документ MS Visio. В состав инструментального средства также включена компонента обратного преобразования — файла формата XML в документ MS Visio. Для выполнения такого преобразования необходимо воспользоваться компонентой *Xml2Visio.exe*. Компоненте в качестве параметров передаются путь до файла с XML-представлением графа переходов и название итогового файла:

```
Xml2Visio.exe automatas.xml analyzers.new.vsd
```

Данная компонета по XML-файлу (*automatas.xml*) строит графическое представление графа переходов. Получаемый в результате файл отличается от исходного только расположением состояний.

Интеграция инструментального средства со средой разработки MS Visual Studio. Инструментальное средство MetaAuto предполагает возможность интеграции со средами разработки. Интеграцию предлагаемого инструментального средства со средой разработки MS Visual Studio рассмотрим на изложенном выше примере.

Для обеспечения интеграции можно создать *makefile* — файл, использующийся утилитой *nmake*. Этот файл предназначен для автоматизации процесса преобразования графа переходов в файл с исходным кодом. В рассматриваемом примере необходимо автоматизировать шаги 2 и 4. Для задания зависимостей между файлами, генерируемыми на этих шагах, используются командные строки, приведенные в описании этих шагов. Вариант *makefile* для рассматриваемого выше примера приведен в листинге 7.

Листинг 7. *makefile для автоматизации процесса генерации исходного кода*

```
    Необходимый результат
result : common.h common.cpp

    Шаг 4 для файла common.h
common.h : automatas.xml common.h.xslt
   XSLTransform.exe automatas.xml common.h.xslt common.h

    Шаг 4 для файла common.cpp
common.cpp : automatas.xml common.cpp.xslt
   XSLTransform.exe automatas.xml common.cpp.xslt
                                              common.cpp
    Шаг 2
automatas.xml : automatas.vsd
        Visio2Xml.exe automatas.vsd automatas.xml
```

Интеграция инструментального средства предполагает внедрение его в процесс компиляции проекта. Для этого в процесс компиляции требуется:

— добавить в проект необходимые для преобразования файлы;

— задать командную строку `"$(ProjectDir)/nmake.exe"`, выполняющую преобразование в качестве события, выполняющегося до компиляции, с помощью свойства `Pre-Build Event Command Line`.

Программа *nmake.exe* использует *makefile* и создает файлы *common.h common.cpp*.

Для тестирования рассмотренного способа интеграции инструментальное средство было внедрено в процесс компиляции самого себя. С помощью инструментального средства генерируются исходные коды синтаксического и лексического анализаторов, используемых для разбора логических выражений. Исходные коды разработанного инструментального средства, в которых используется интеграция со средой разработки MS Visual Studio, размещены на сайте http://is.ifmo.ru/projects/metaauto/.

Заключение

В статье описан подход к автоматической генерации исходного кода автоматных программ. Рассмотрены различные технологии и методики, их преимущества и недостатки. Также описана реализация предложенного подхода в инструментальном средстве MetaAuto.

Это инструментальное средство разрабатывалось как генератор кода автоматных программ на любом априори заданном языке программирования. Оно было использовано для генерации кода синтаксического и лексического анализаторов, использующихся самим инструментальным средством. Таким образом, первым успешным внедрением инструментального средства стало оно само.

Однако после написания прототипа инструментального средства оказалось, что его можно использовать также и для верификации программ. Такая верификация была выполнена в работе [14]. С помощью инструментального средства процесс создания верификатора был достаточно прост, так как граф переходов легко доступен при использовании программного представления модели.

Авторы надеются, что настраиваемость инструментального средства и легкость его использования позволят более эффективно внедрять автоматное программирование [15].

Литература

1. **Новиков Ф. А.** Визуальное конструирование программ //Информационно-управляющие системы. 2005. № 6. С. 9–22. http://is.ifmo.ru/works/visualcons/
2. **Finite** state machine. Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Finite_automaton.
3. **Сайт** проекта «Finite State Machine». http://fsme.sourceforge.net.
4. **Головешин А.** Использование конвертора Visio2 SWITCH. http://is.ifmo.ru/?i0=progeny&i1=visio2switch.
5. **Гуров В. С., Мазин М. А., Нарвский А. С, Шалыто А. А.** UML. SWITCH-технология. Eclipse //Информационно-управляющие системы. 2004. № 6. С. 12–17. http://is.ifmo.ru/works/uml-switch-eclipse/.
6. **Канжелев С. Ю., Шалыто А. А.** Преобразование графов переходов, представленных в формате MS Visio, в исходные коды программ для различных языков программирования (инструментальное средство MetaAuto). 102 с. http://is.ifmo.ru/projects/metaauto/.
7. **Чарнецки К., Айзенекер У.** Порождающее программирование: методы, инструменты, применение. СПб.: Питер, 2005. 736 с.
8. **Селлз К.** Современные способы автоматизации повторяющихся задач программирования //MSDN Magazine. 2002. N 6. P. 8–13.
9. **Dodds L.** Code generation using XSLT, ibm.com/developerWorks, http://www-106.ibm.com/developerworks/edu/x-dw-codexslt-i.html.
10. **Ashley P.** Simplify Development and Maintenance of Microsoft .NET Projects with Code Generation Techniques, http://msdn.microsoft.com/msdnmag/issues/03/08/CodeGeneration/default.aspx.
11. **Herrington J.** Extensible Code Generation with Java, http://today.java.net/pub/a/today/2004/05/12/generation1.html.
12. **Заякин Е. А., Шалыто А. А.** Метод устранения повторных фрагментов кода при реализации конечных автоматов. 21 с. http://is.ifmo.ru/projects/life_app.
13. **Шалыто А. А., Туккель Н. И.** SWITCH-технология — автоматный подход к созданию программного обеспечения «реактивных» систем //Программирование. 2001. № 5. С. 45–62. http://is.ifmo.ru/works/switch/.
14. **Канжелев С. Ю., Шалыто А. А.** Моделирование кнопочного телефона с использованием SWITCH-технологии. Вариант 2. 104 с. http://is.ifmo.ru/projects/phone/.
15. **Шалыто А. А.** Технология автоматного программирования //Мир ПК. 2003. № 10. С. 74–78. http://is.ifmo.ru/works/tech_aut_prog/