Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

С.Ю. Канжелев, А.А. Шалыто

Преобразование графов переходов, представленных в формате *MS Visio*, в исходные коды программ для различных языков программирования (инструментальное средство *MetaAuto*)

Проектная документация

Проект выполнен в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2005

Оглавление

Введение 5

Глава 1. Постановка задачи 8

1.1. Требования, предъявляемые к инструментальным средствам для работы с графами переходов автоматов 8

1.2. Инструментальные средства для создания и работы с графами переходов автоматов 9

1.2.1. *Visio2Switch* 10

1.2.2. *UniMod* 11

1.2.3. Редактор *FSME* (*Finite State Machine Editor*) 13

1.2.4. Необходимость разработки нового инструментального средства 13

1.3. Постановка задачи 14

1.3.1. Требования, предъявляемые к инструментальному средству 14

1.3.2. Нотация графов переходов автоматов 15

Глава 2. Общие принципы реализации 18

2.1. Преобразование графа переходов в исходный код программы 18

2.1.1. Известные способы генерации кода программ 18

2.1.2. Особенности генерации исходного кода для графа переходов в SWITCH-технологии 21

2.1.3. `XSLT`-преобразование 22

2.2. Работа с графическим редактором *MS Visio* 24

2.2.1. Создание библиотеки графических элементов 24

2.2.2. Работа с файлами редактора *MS Visio* 27

2.3. Конфигурирование 27

Глава 3. Проектирование 29

3.1. Декомпозиция задачи 29

3.2. Компонента *Visio2Xml* 31

3.3. Компонента *Xml2Visio* 32

3.4. Формат и способ использования конфигурационного файла 32
3.4.1. Конфигурируемые элементы графа переходов 32
3.4.2. Формат конфигурационного файла.. 33
3.4.3. Способ использования конфигурационного файла 35
3.5. Библиотека *MetaAuto* 35
3.5.1. Диаграмма классов библиотеки *MetaAuto*.. 36
3.6. Парсеры.. 40
3.6.1. Синтаксические анализаторы (общие принципы) 41
3.6.2. Синтаксический анализатор для разбора линейного списка 43
3.6.3. Синтаксический анализатор для разбора логического выражения43
3.6.4. Лексический анализатор 45
3.6.5. Формат выходного XML-файла 46
3.7. XSLT-процессор 51
3.8. Программа *nmake* 51
Глава 4. Примеры использования программы 52
4.1. Методика использования инструментального средства 52
4.1.1. Шаг 1. Изображение графов переходов.. 52
4.1.2. Шаг 2. Преобразования в XML-формат 54
4.1.3. Шаг 3. Создание XSLT-шаблона... 57
4.1.4. Шаг 4. Получение исходного кода 57
4.1.5. Шаг 5. Преобразование XML-файла в документ *MS Visio* 60
4.2. Интеграция инструментального средства со средой разработки *MS Visual Studio*.. 60
4.3. Создание XSLT-шаблонов... 61
4.4. Создание конфигурационного файла 72
Заключение 75
Список литературы 76
Приложение 1. Стандартный конфигурационный файл 78

Приложение 2. XSLT-шаблон, генерирующий исходный код на языке *C#* 79
Приложение 3. XSLT-шаблон, генерирующий файл auto.asm 83
Приложение 4. XSLT-шаблоны, генерирующие код на языке *C*, идентичный коду, генерируемому конвертером *Visio2Switch* 90

Введение

В работе [1] был предложен подход к созданию программ для систем логического управления на основе использования конечных автоматов. Этот подход был назван «SWITCH-технология» или «автоматное программирование» и получил развитие в работах [2, 3] в виде «процедурного программирования с явным выделением состояний» и «объектно-ориентированного программирования с явным выделением состояний».

Изложим основные особенности этого подхода. Если традиционное процедурное программирование базируется на терминах «данные» и «процедуры», объектно-ориентированное программирование – на «атрибутах» и «методах», а событийное программирование – на «событиях» и «обработчиках событий», то в автоматном программировании, как и в теории управления, используются три понятия: «входное воздействие», «состояние» и «выходное воздействие». При этом «состояния» рассматриваются как абстракции и выделяются на этапе проектирования явно. Связь этих понятий осуществляется при построении графов переходов автоматов. Автоматы обычно рассматриваются, как конечные и детерминированные и предназначены для описания поведения (логики) программ.

Таким образом, автоматное программирование основано на априорном задании графов переходов автоматов и их визуализации. Как отмечено в работе [2], процесс создания программного продукта в случае, когда требуется построить исходный код, разбивается на два этапа: проектирование с описанием поведения в виде графов переходов автоматов и собственно программирование, которое в этой части сводится к формальному преобразованию построенных графов переходов в исходные коды программ.

В этом и заключается основное достоинство автоматного подхода – в возможности разделения этапов проектирования и создания программы таким образом, чтобы эти этапы могли выполняться различными специалистами. При

этом переход от первого этапа ко второму может осуществляться формально и изоморфно.

В ходе педагогического эксперимента [4] выполнено более шестидесяти проектов с применением автоматного подхода (http://is.ifmo.ru, раздел «Проекты»). Использовались различные языки программирования и подходы к реализации автоматов. В работе [5] сделана попытка систематизации предлагаемых подходов. Следует отметить, что, несмотря на то, что семантика графов переходов часто отличалась от предложенной в работе [3], нотация, применяемая для построения графов переходов и описанная в данной работе, оставалась неизменной.

При создании многих проектов возникали проблемы с формальным преобразованием графов переходов в исходные коды программ, так как для различных языков программирования и различных методов реализации не разработаны соответствующая семантика и шаблоны реализации. При этом изменения, вносимые в граф переходов, приходилось переносить в исходные коды программ вручную. Это снижало эффективность применения автоматного подхода.

Известны инструментальные средства для поддержки автоматного программирования. Довольно большой список этих средств представлен в статье из интернет–энциклопедии *WikipediA*, посвященной конечным автоматам [6]. В настоящей работе выполнен анализ трех средств, которые поддерживают SWITCH-технологию: *Visio2Switch* [7], *UniMod* [8, 9], *FSME* [10]. Главный недостаток перечисленных средств состоит в том, что в них не предусмотрена настройка получаемого кода под требования конкретного проекта. Исходный код в них генерируется на одном или двух заранее заданных языках.

Настоящая работа посвящена разработке инструмента, позволяющего устранить указанный недостаток. Учитывая тот факт, что в работах на сайте http://is.ifmo.ru большинство графов переходов проектировалось с помощью редактора *MS Visio*, в настоящей работе создается инструментальное средство,

позволяющая преобразовывать изображение графов переходов, представленное с помощью этого редактора, в исходные коды программ на любых языках программирования, для которых предварительно создаются специальные шаблоны. Также инструментальное средство позволяет преобразовывать `XML`-описание автомата в документ *MS Visio*. В ходе работы над инструментальным средством в качестве примера созданы шаблоны для трех языков программирования (*C, C#, Turbo Assembler*).

В работе обосновывается целесообразность разработки данного инструментального средства и проводится его сравнение с существующими аналогами. Также приводятся примеры его использования. В частности, рассматривается способ интеграции инструментального средства со средой разработки *MS Visual Studio 2003* для языков *C#* и *C*.

В первой главе выполнена постановка задачи. Перечислены требования, предъявляемые к разрабатываемому инструментальному средству, названному *MetaAuto*, и выполнен анализ аналогичных инструментов. Формализуется нотация, применяемая при построении графов переходов.

Во второй главе описываются технологии, использованные при создании инструментального средства. Обосновывается выбор данных технологий.

В третьей главе приводится проектная документация на разработанную программу. Описываются форматы получаемых данных (в частности, формат `XML`-описания автоматов) и интерфейсы взаимодействия частей созданной программы.

В четвертой главе приводится пример разработки `XSLT`-шаблона, и приводятся примеры применения разрабатываемого средства.

Глава 1. Постановка задачи

1.1. Требования, предъявляемые к инструментальным средствам для работы с графами переходов автоматов

Графы переходов автоматов предназначены для:

- документирования логики работы программы;
- уменьшения времени, затрачиваемого на ручное и автоматическое кодирование программ;
- уменьшения времени, затрачиваемого на отладку и тестирование программ.

В связи с этим возникает ряд требований к программам, работающим с графами переходов автоматов:

- должна быть обеспечена возможность добавления комментариев и рисунков, облегчающих понимание графов переходов;
- должна быть обеспечена возможность экспорта графов переходов в графический формат или иной формат, воспринимаемый текстовыми редакторами;
- для программ, предусматривающих автоматическую генерацию кода, необходимы средства настройки получаемых файлов с исходным кодом. Такой настройкой может являться добавление контекстных переменных, специфичных для проекта, использование особых потоков вывода для логирования, добавление кода обработки ошибок, или добавление информации об авторских правах.

Выполним анализ существующих инструментов для создания и работы с графами переходов автоматов на основе перечисленных выше требований.

1.2. Инструментальные средства для создания и работы с графами переходов автоматов

Для анализа поведения одиночного объекта в *UML* используются диаграммы состояний (*Statecharts Diagram*) [11]. В работе [3] произведено сравнение этих диаграмм с графами переходов, применяемыми в SWITCH-технологии. В ходе сравнения было отмечено, что рассматриваемые графы переходов реализуются по шаблону, в то время как вопрос о реализации диаграмм состояний в *UML* не затрагивается.

Так, в частности, в книге [11] отмечается, что *UML*-диаграммы состояний строятся только для описания сути происходящих процессов, а вопрос о формальном и изоморфном преобразовании в исходный код даже не обсуждается.

Поэтому будем рассматривать только те инструменты, которые работают с графами переходов, используемыми в SWITCH-технологии. «Пограничным» инструментом является *UniMod*, использующий идеологию SWITCH-технологии и нотацию *UML*.

Среди этих инструментов рассмотрим три: конвертор *Visio2 Switch* [7], плагин *UniMod* к платформе *Eclipse* [8, 9] и продукт *FSME (Finite State Machine Editor)* [10]. Эти продукты созданы для решения различных задач.

Конвертер *Visio2Switch* предназначен для преобразования графов переходов, разработанных с помощью редактора *MS Visio*, в исходный код программы на языке *C*.

Редактор *Finite State Machine Editor* позволяет изображать графы переходов и преобразовывать их в исходный код на языках *C++* и *Python*.

Плагин *UniMod* к платформе *Eclipse* позволяет строить графы переходов автоматов, используя в качестве входных и выходных воздействий методы объектов управления, реализованных на языке *Java*. Этот плагин позволяет

интерпретировать описание, а при необходимости его компилировать. Также он позволяет производить визуальную отладку этих графов.

Рассмотрим подробнее особенности перечисленных инструментов.

1.2.1. *Visio2Switch*

Конвертор *Visio2Switch* предназначен для автоматической генерации кода на языке *C* по автоматным графам, разработанным в редакторе *MS Visio* в соответствии с требованиями SWITCH-технологии [2]. В комплект поставки программы входит файл `SWITCH.vss`, являющийся шаблоном *MS Visio*, который содержит элементы для изображения графов переходов автоматов. Конвертор представляет собой встраиваемый модуль (*add-in*) к графическому редактору *MS Visio*. Это исполняемый файл, имеющий самостоятельный пользовательский интерфейс, но требующий для работы наличие запущенной программы *MS Visio* и открытого документа, который предстоит преобразовывать.

Достоинствами данного инструмента являются:

- широкие возможности для оформления графов переходов (добавление комментариев и рисунков), предоставляемые редактором *MS Visio*;
- возможность экспорта графов переходов в различные графические форматы или непосредственная их вставка в документ *MS Word*. Это важно при оформлении документации и создании отчетов;
- удобство создаваемого кода. В частности, выделены файлы для модификации пользователем и предусмотрено автоматическое логирование.

Недостатками инструмента являются:

- исходный код инструмента закрыт;
- фиксированный язык программирования, на котором генерируется код;
- невозможность настройки создаваемого кода для нужд конкретного проекта;

- ограничения в задании графов переходов. В частности, не предусмотрена возможность вызова автоматов на ребре, не предусмотрен вызов автомата с конкретным событием, не поддерживается логическая операция `XOR`, названия входных и выходных переменных должны быть уникальны для всех автоматов в проекте;
- недостаточно полный шаблон. Не предусмотрены графические элементы, обозначающие переходы, условия и действия на которых разделены не знаком «слеш» («/»), а горизонтальной чертой;
- невозможность запуска из командной строки и требование наличия запущенной программы *MS Visio*.

1.2.2. *UniMod*

Инструментальное средство *UniMod* используется для создания «исполняемых» графов переходов для платформы *Eclipse*. В качестве основного применяется так называемый интерпретируемый подход к реализации графов переходов. Этот подход характеризуется наличием ядра (*UniMod Runtime Engine*), которое, используя внутреннее представление графов переходов, реализует логику работы автомата в процессе выполнения программы (при применении интерпретации).

Данное инструментальное средство использует идеологию SWITCH-технологии и нотацию *UML*.

Достоинствами *UniMod* являются:

- привязка к классам уже существующего проекта;
- экспорт графов переходов в графический файл;
- визуальная отладка графов переходов;
- интерпретационный и компилятивный подходы к реализации графов переходов.

Недостатки *UniMod*:

- ограниченность в оформлении графов переходов – не предусмотрено возможности добавления комментариев и картинок;
- возможность применения только в проектах, создаваемых на языке *Java* и только на платформе *Eclipse*;
- невозможность генерации исходного кода программы.

Как отмечено выше, *UniMod* также позволяет автоматически генерировать код. В частности, на встрече *Java User Group* (http://jug.ru), прошедшей 26.02.2005, был продемонстрирован «компилятивный подход» к использованию графов переходов, создаваемых с помощью *UniMod* [12]. Данный подход, в отличие от интерпретационного, реализует логику работы автомата не в процессе исполнения программы, а сгенерировав предварительно исходный код автоматного класса. На рис. 1 приведена схема, по которой выполняется такая генерация.

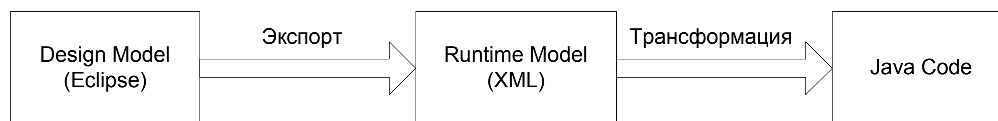


Рис. 1. Схема генерации кода при использовании компиляции графов переходов в *UniMod*

Трансформация `XML`-модели в *Java*-код осуществляется с использованием *Velocity*-шаблонов. Подробнее *Velocity*-шаблоны и другие способы генерации кода будут описаны в разд. 2.1.

«Компилируемый подход» был использован при создании приложений для мобильных телефонов. Необходимость его применения была связана с тем, что *UniMod Runtime Engine* (ядро *UniMod*, реализующее логику работы автомата при «интерпретируемом подходе») требует наличие библиотеки *JDK 1.4* и, поэтому его невозможно применять при разработке приложений для мобильных телефонов (в частности, для *J2ME MIDP* приложений).

1.2.3. Редактор *FSME* (*Finite State Machine Editor*)

Продукт состоит из трех компонент:

- *FSME* (*Finite State Machine Editor*). Данная компонента используется для создания и редактирования графов переходов автоматов. Полученный с его помощью граф переходов может быть сохранен в формате `XML`;
- *FSMC* (*Finite State Machine Compiler*). Конвертер графов переходов, сохраненных в `XML`-формате в языки *C++* и *Python*;
- *FSMD* (*Finite State Machine Debugger/Tracer*). Компонента для визуальной отладки графов переходов.

Достоинствами программы являются:

- совмещение функций редактирования и преобразования графов переходов в одной программе;
- возможность отладки полученных программ;

Недостатками программы являются:

- несоответствие дизайна получаемых графов переходов общепринятому;
- не предусмотрена возможность дополнительного оформления графов переходов;
- возможность генерации кода лишь на двух языках (*C++* и *Python*) и отсутствие возможности настройки получаемого кода.

1.2.4. Необходимость разработки нового инструментального средства

Проведенный анализ инструментальных средств для работы с графами переходов автоматов показал, что рассмотренные средства могут использоваться исключительно для конкретных языков программирования. Отсутствует возможность настройки на требуемый язык программирования. Часто, как, например, в случае с программным обеспечением для мобильных телефонов,

возникает необходимость модифицировать инструментальное средство или писать дополнительный модуль к нему.

Поэтому, даже несмотря на теоретическую возможность автоматической генерации кода по графу переходов автомата, фактически, как показывает анализ проектов на http://is.ifmo.ru, кодирование чаще всего происходит вручную. Это снижает надежность построенных программ.

Необходимо разработать новое инструментальное средство, позволяющее устранить эти недостатки.

1.3. Постановка задачи

1.3.1. Требования, предъявляемые к инструментальному средству

Учитывая требования, предъявляемые к инструментальным средствам для работы с графами переходов, сформулированные в разд. 1.1, и особенности рассмотренных в разд. 1.2 средств, были сформулированы требования к инструментальному средству, разрабатываемому в рамках данной работы. Перечислим эти требования:

- инструментальное средство должно работать с графами переходов, разработанными с помощью редактора *MS Visio*. В комплекте с программой должен предоставляться файл с шаблонами для изображения графов переходов;
- необходима обратная совместимость с конвертером *Visio2Switch*;
- инструментальное средство должно позволять преобразовывать `XML`-описания графов переходов в файлы *MS Visio*;
- необходимо иметь возможность настраивать используемые на графе переходов обозначения. В частности, требуется предусмотреть возможность использования названий состояний, входных и выходных переменных, автоматных вызовов, не ограничиваясь форматом, предлагаемым в работе [7]. Нотация графов переходов, которую

должно поддерживать инструментальное средство, приведена в разд. 1.3.2;

- необходимо предусмотреть возможность генерации исходного кода на различных языках программирования. Также необходима возможность «настройки» кода для нужд конкретного проекта;
- инструментальное средство должно запускаться из командной строки с целью интеграции его с различными средами разработки.

В следующем разделе приведена нотация графов переходов, использующихся в SWITCH-технологии и в разрабатываемом инструментальном средстве.

1.3.2. Нотация графов переходов автоматов

В работе [2] была предложена нотация графов переходов, представленная на рис. 2. Данная нотация успешно использовалась в нескольких проекта, однако, впоследствии она была расширена. В частности, была добавлена возможность вызова автомата на переходе (дуге) и вызова автомата с конкретным событием. Также не всегда использовались предложенные в работе [2] обозначения для входных и выходных переменных. Однако основные составляющие графов переходов всегда оставались. Так, состояния всегда имели название и включали в себя список выполняемых в состоянии действий (выходных переменных) и автоматов. Переходы – соединяли состояния или групповые состояния и имели приоритет, условие перехода и действия, выполняемые на переходе. Формализуем нотацию графа переходов автомата, с которой работает разрабатываемое инструментальное средство.

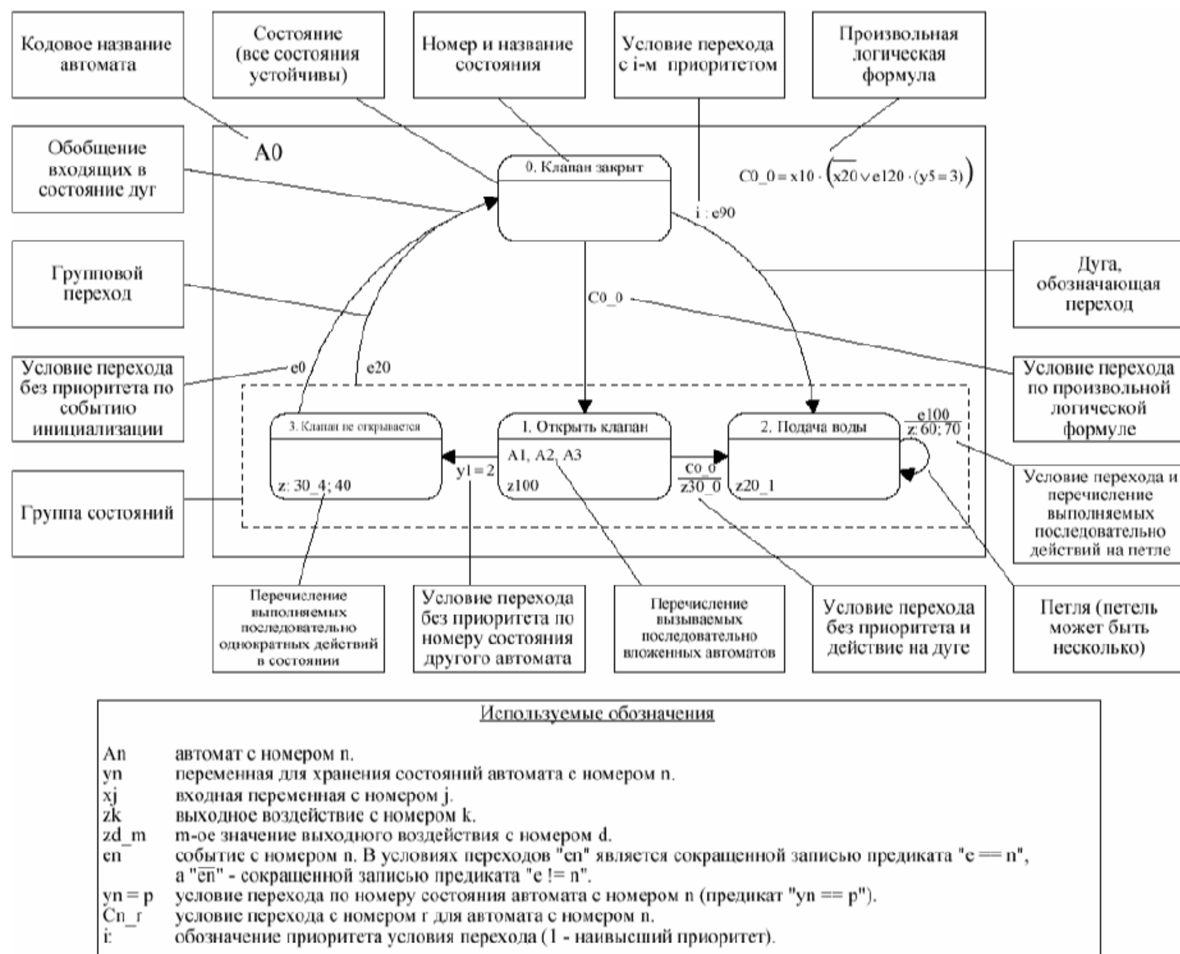


Рис. 2. Нотация графов переходов

Граф переходов автоматов состоит из состояний, групповых состояний, переходов, а также групповых переходов. Опишем каждую составляющую графа переходов отдельно.

Состояние

Состояние имеет идентификатор (в нотации работы [2] – номер) и название. Состояние может содержать набор вложенных автоматов (список автоматов для запуска) и набор выходных переменных (список действий для исполнения).

Групповое состояние

Служит для задания групповых переходов. Содержит в себе множество состояний и других групповых состояний. На графах переходов, используемых в

рамках SWITCH-технологии, не принято давать название групповым состояниям.

Переход и групповой переход

Переход и групповой переход соединяют одно состояние или групповое состояние с другим состоянием. Переход и групповой переход содержат условие перехода (произвольное логическое выражение) и набор действий на переходе (список действий для исполнения). Переход может иметь приоритет – произвольное число, задающее порядок проверки условий переходов, выходящих из одного состояния.

Элементы всех списков, а также логического выражение, отвечающее за условие перехода по дуге, должны настраиваться.

Глава 2. Общие принципы реализации

2.1. Преобразование графа переходов в исходный код программы

Разрабатываемое инструментальное средство должно осуществлять преобразование графа переходов автомата в исходный код программы. В общем случае, можно рассматривать граф переходов как набор данных определенной структуры из предметной области решаемой задачи. Вопрос об автоматической генерации кода с заданными свойствами при этом является одним из вопросов решаемых в рамках *Generative Programming* (в русском языке нет устоявшегося названия для этого термина: в разных источниках его называют «генеративное программирование» или «порождающее программирование»).

Рассмотрим основные способы генерации кода, используемые в «порождающем программировании».

2.1.1. Известные способы генерации кода программ

На данный момент разработано множество способов генерации кода. Среди этих способов можно выделить следующие виды:

- 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, осуществляет подстановку этих данных в шаблон. Набор используемых в шаблоне подстановок может определяться как статически, так и динамически. Такой подход весьма 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`) *C++*.

Гораздо шире возможности при генерации кода на основе второго из перечисленных способов. Этот вид генерации отличается от предыдущего возможностью использовать в шаблоне не только подстановки, но также вставки исполняемого кода, оперирующего переданными в шаблон данными. Исполняемый код чаще всего использует язык, специально созданный для конкретного типа шаблонов и включающий следующие основные конструкции:

- простое ветвление;
- выполнение итераций по переданным в качестве параметров спискам;
- циклы с заданным количеством итераций;
- обращение к данным из внешних источников (обращение к специально подготовленным и переданным в качестве параметров объектам);
- включение других файлов шаблонов или любого текстового файла (статическое включение);
- синтаксический разбор файлов шаблона (динамическое включение);
- локальные переменные.

Примером такого подхода к генерации является технология *Jakarta Velocity*, в которой используется специальный язык (называемый ее создателями «язык шаблонов») с синтаксисом, похожим на используемый в языке *Perl*.

Однако есть варианты, когда применяются языки с более широкими возможностями. Например, технологии *ASP* и *JSP* используют языки *Basic* и *Java* соответственно. Следует заметить, что в процессе усложнения конструкций, применяемых в шаблоне, и увеличения возможностей языка, уменьшается наглядность самих шаблонов.

Способ генерации кода с применением технологии *ASP* приведен в статье [13]. Ниже в листинге 1 приведен пример шаблона, описываемого в этой статье.

Листинг 1. Способ генерации кода с использованием технологии *ASP*

```
<%@ language=jscript %><% // test.cpp.asp %>
<%
    var greeting = Request.QueryString("greeting");
    if( greeting.length == 0 ) greeting = "Hello, World.";
%>
// test.cpp

<% if( Request.QueryString("iostream").length != 0 ) { %>
#include <iostream>
using namespace std;
<% } else { %>
#include <stdio.h>
<% } %>

int main()
{
<% if( Request.QueryString("iostream").length != 0 ) { %>
    cout << "<%= greeting %>" << endl;
<% } else { %>
    printf("<%= greeting %>\n");
<% } %>
    return 0;
}
```

Данный шаблон в зависимости от переданных ему данных (в частности, в зависимости от переменной `iostream`, содержащейся в строке запроса), создает один из приведенных в листинге 2 кодов программы.

Листинг 2. Коды программ, получаемых с помощью шаблона, приведенного в листинге 1, в случае разных значений переменной `iostream`

```
// test.cpp

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

```
// test.cpp

#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

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, основанный на обработчиках данных регулярной структуры, 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 [14–18].

Основным достоинствами генерации кода программ, основанной на обработчиках данных регулярной структуры, является возможность обработки данных со сложной структурой. Недостатком является меньшая, по сравнению с генерацией кода, основанной на подстановках, наглядность получаемых шаблонов.

Из изложенного следует, что вопрос о применении того или иного способа генерации кода программ зависит от условий конкретной задачи и данных, используемых для такой генерации. Рассмотрим особенности генерации исходного кода для графов переходов автоматов в SWITCH-технологии.

2.1.2. Особенности генерации исходного кода для графа переходов в SWITCH-технологии

Одним из вопросов, решаемых при преобразовании графов переходов автоматов в исходные коды программ, является вопрос о реализации групповых переходов [19]. Как отмечалось в работе [3], 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.

В случае использования способа генерации кода, основанного на подстановках, предложенный метод реализации группового перехода требует предварительной подготовки данных. Необходимо продублировать все

групповые переходы для всех внутренних состояний групп. При такой подготовке теряется информация о групповых переходах, которые были вначале.

Другой трудностью использования генерации, основанной на подстановках, является наличие логических выражений, которые чаще всего представляются бинарным деревом. В случае отсутствия в языке, используемом в шаблоне, поддержки вызова рекурсивных функций, разбор такого дерева представляется непростой задачей.

Перечисленные особенности генерации исходного кода для графов переходов в SWITCH-технологии определяют выбор способа генерации исходного кода – использование XML-представления графа перехода автомата и XSLT-преобразования для генерации исходного кода представляются наиболее удобными.

2.1.3. XSLT-преобразование

XSLT (eXtensible Stylesheet Language Transformations – на русский язык переводится как «расширяемый язык стилей для преобразований») в последнее время стал очень популярной XML-технологией. Спецификация определяет XSLT как язык для преобразований одних XML-документов в другие XML-документы. Однако за время своего существования XSLT стал применяться значительно шире. Теперь уместнее согласиться с редактором новой версии языка Майклом Кеем (Michael Kay) в том, что XSLT – это язык для преобразования структуры документов.

В преобразовании участвуют два документа: входящий документ, который подвергается преобразованию, и документ, который описывает само преобразование. Результатом преобразования является исходящий документ. Отметим, что в процессе преобразования можно использовать данные из других XML-документов, что позволяет создавать еще более настраиваемые решения.

На данный момент еще не вышла спецификация языка XSLT 2.0, значительно расширяющая действующую – XSLT 1.0. Однако уже сейчас спецификация XSLT находится на уровне, позволяющем ее эффективное применение для самых разных задач. Например, для автоматической генерации кода.

В работах [14–18] рассматривается вопрос использования языка XSLT в качестве инструмента для генерации кода программ. В работе [14] отмечены основные достоинства и недостатки генерации кода с применением XSLT. Перечислим их.

XSLT-генерация кода имеет следующие преимущества:

- 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 распространения XML-формата для данных. Нет необходимости каждый раз придумывать новый формат хранения данных;

Недостатки языка XSLT при использовании его для генерации кода:

- 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 [16, 18] или по списку классов *UML*-подобных диаграмм [17]). В этом случае сложно оценить все преимущества использования `XSLT`-преобразования. Разрабатываемое инструментальное средство демонстрирует применение `XSLT`-преобразований, использующих данные нелинейной структуры.

2.2. Работа с графическим редактором *MS Visio*

Графический редактор *MS Visio* является приложением для создания и редактирования графических изображений. Редактор предоставляет широкие возможности для создания библиотек графических элементов и манипулирования этими элементами. Разрабатываемое инструментальное средство, согласно требованиям, сформулированным в разд. 1.3.1, должно иметь библиотеку графических элементов. Также, рассматриваемое инструментальное средство должно уметь извлекать информацию из файлов, создаваемых *MS Visio*. Рассмотрим, как реализуются создание библиотеки и извлечение информации из файлов, создаваемых *MS Visio*.

2.2.1. Создание библиотеки графических элементов

В целях обеспечения обратной совместимости с конвертером *Visio2Switch*, все графические элементы шаблона `switch.vss`, поставляемого вместе с конвертером, были скопированы в новый шаблон. Помимо старых графических элементов, были добавлены новые – переходы, условия и действия у которых разделены горизонтальной чертой. Разработка данных графических элементов осуществлялась с помощью задания `ShapeSheet`-информации для графических элементов.

ShapeSheet-информация графического элемента определяет размеры, форматирование и поведение элемента и представляет собой набор пар «свойство-значение», где в качестве значений могут выступать как определенные величины, так и формулы, зависящие от значений других свойств того же или другого графического элемента.

Рассмотрим, например, процесс создания элемента, представляющего собой переход, у которого условие и действие разделены горизонтальной чертой. Такой элемент изображен на рис. 3.

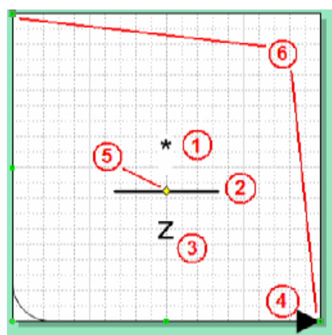


Рис. 3. Элемент, представляющий переход, условие и действие которого разделены горизонтальной чертой

Элемент состоит из четырех составных частей:

- условие перехода (обозначено цифрой 1 на рис. 3);
- горизонтальная разделительная черта (2);
- действия на переходе (3);
- стрелка перехода (4);

Также в элементе предусмотрены:

- управляющая точка (Control Point) для управления положением текста (5);
- две точки соединения (Connection Point) – точки для соединения с состояниями и группами состояний (6).

Перечисленные составные части должны некоторым образом взаимодействовать. Например, ширина разделительной черты должна быть не

меньше ширины условия и действия (рис. 4). Также эта черта не может быть наклонена, а всегда должна быть горизонтальной.

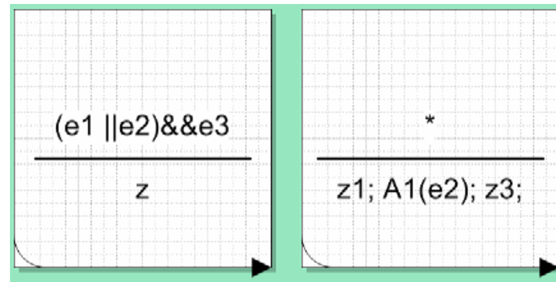


Рис. 4. Горизонтальная черта должна быть не меньше ширины условия и действия

Эти взаимодействия могут быть заданы с помощью `ShapeSheet`-информации. Например, для горизонтальной черты необходимо задать следующие пары свойство-значение:

```
BeginX    =GUARD(PNTX(PAR(PNT( DynConnector!Controls.TextPosition-
          MAX(DynConnector!TxtWidth/2,ActionText!Width/2),
          DynConnector!Controls.TextPosition.Y ))))
EndX      =GUARD(PNTX(PAR(PNT( DynConnector!Controls.TextPosition
          +MAX(DynConnector!TxtWidth/2,ActionText!Width/2),
          DynConnector!Controls.TextPosition.Y ))))
BeginY    =GUARD(PNTY(PAR(PNT( DynConnector!Controls.TextPosition,
          DynConnector!Controls.TextPosition.Y ))))
EndY      =GUARD(BeginY)
```

В приведенных формулах применяются следующие названия:

- `DynConnector` – название стрелки перехода (на рис. 3 обозначена цифрой 4);
- `DynConnector!Controls.TextPosition` и `DynConnector!Controls.TextPosition.Y` – координаты управляющей точки по осям X и Y соответственно, отвечающей за положения текста графического элемента `DynConnector`;
- `ActionText` – текст, действия на переходе (на рис. 3 обозначена цифрой 3);

и функции

- `GUARD` – предохранение формулы от затирания конкретным значением;
- `PNTX` – взять *x* координату точки;
- `PAR` – преобразовать координаты переданной точки из системы координат родительского элемента в абсолютную систему координат;
- `PNT` – преобразование координат в точку.

Аналогично образом задается поведение остальных элементов. Справку по работе со `ShapeSheet`-информацией, можно найти в *MSDN* [20].

2.2.2. Работа с файлами редактора *MS Visio*

Извлечение информации из файлов *MS Visio* и генерация таких файлов в инструментальном средстве производится с использованием *COM*-объекта, устанавливаемого вместе с установкой редактора *MS Visio*. Программа, извлекающая информацию, написана на языке *C#*. Интерфейсы *COM*-объекта позволяют открыть любой файл, созданный с помощью редактора *MS Visio*, а также получить информацию о каждом графическом элементе, содержащемся в данном файле. Также эти интерфейсы позволяют создавать файлы и записывать в них необходимые графические элементы.

Отметим, что исходя из требований, сформулированных в разд. 1.3.1, анализируя графические элементы, необходимо пропускать неизвестные элементы. Это могут быть различные пояснительные рисунки и надписи.

2.3. Конфигурирование

Согласно требованиям, сформулированным в разд. 1.3.1, инструментальное средство должно иметь возможность настройки применяемых на графе переходов обозначений. Необходимо иметь возможность настройки синтаксиса, используемого для задания списков действий в состоянии и на ребрах, используемые в этих списках символы-разделители, а также формат входных и выходных переменных.

Возможность конфигурирования такого рода основана на применении регулярных выражений. Регулярные выражения – мощный инструмент 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;
- возможность задавать и использовать любое количество именованных областей. Эта возможность применяется в разработанном инструментальном средстве (разд. 3.6.5).

Глава 3. Проектирование

3.1. Декомпозиция задачи

Преобразование графа переходов из формата *MS Visio* в исходные коды программ и из `XML`-формата в файлы *MS Visio* было разделено на несколько этапов:

- преобразование документа *MS Visio* в файл формата `XML`;
- преобразование файла формата `XML` в документ *MS Visio*;
- преобразование файла из формата `XML` в исходные коды программы.

Первые два из этих этапов, были разделены на два подэтапа.

Преобразование документа *MS Visio* в файл формата `XML` состоит из:

- заполнения информацией объектов библиотеки *MetaAuto*, выполняемого компонентой *Visio2Xml*;
- сериализации (полное сохранение состояния) объекта класса `Model` библиотеки *MetaAuto*, содержащего всю извлеченную из файла *MS Visio* информацию, в `XML`-формат.

Преобразование файла формата `XML` в документ *MS Visio* состоит из:

- десериализации (восстановления состояния) объекта класса `Model` библиотеки *MetaAuto* из `XML`-файла;
- создания с помощью компоненты *Xml2Visio* файла *MS Visio* по информации, содержащейся в объектах библиотеки *MetaAuto*.

Третий этап преобразования – преобразование XML-файла в файл с исходным кодом программы осуществляется XSLT-процессором с применением заданного пользователем шаблона. На рис. 5 представлены все перечисленные этапы.

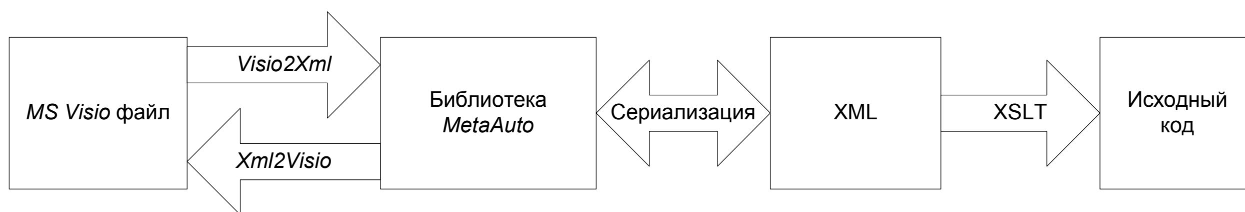


Рис. 5 Декомпозиция задачи

Из рассмотрения этого рисунка следует, что единственный этап, не имеющий себе обратного – преобразование XML-файла в исходный код. Вопрос об эффективной реализации данного преобразовании является открытым. Он в рамках данной работы не обсуждается.

Следует отметить, что в большинстве `XSLT`-шаблонов, создаваемых для генерации исходного кода, необходимо сгенерировать не один файл, а несколько. Например, конвертер *Visio2Switch* создает 10 файлов (`types.h`, `common.h`, `common.cpp`, `log.h`, `log.cpp`, `log_user.cpp`, `x.cpp`, `x_user.cpp`, `z.cpp`, `z_user.cpp`). Поэтому требуется создать не один файл шаблона, а столько, сколько файлов необходимо сгенерировать. Встает вопрос об автоматизации сборки всех файлов проекта.

В работе [14] для реализации автоматической сборки всех необходимых файлов с исходным кодом из одного или нескольких файлов, содержащих метаданные (в случае рассматриваемого инструментального средства – из файлов редактора *MS Visio*), предлагается использовать программу *Ant*. Аналогом этой программы является программа *nmake*, которая служит для задания связей типа «главный файл – зависимый файл» между различными файлами с указанием, способа, с помощью которого можно получить зависимый файл из главного. При каждой сборке программа автоматически проверяет, есть ли необходимость в повторной генерации зависимого файла, основываясь на информации о дате последней модификации файлов. В случае если главный файл был изменен позже, чем зависимый то зависимый файл создается заново.

В данном инструментальном средстве для автоматической сборки предполагается использовать программу *nmake*. Данная программа использует наиболее широко распространенный синтаксис и будет рассмотрена в разд. 3.8.

Таким образом, инструментальное средство состоит из:

- компоненты *Visio2Xml*, осуществляющей чтение *MS Visio* файла и создание объектов библиотеки *MetaAuto* (разд. 3.2);
- компоненты *Xml2Visio*, осуществляющей создание *MS Visio* файла по информации, содержащейся в объектах библиотеки *MetaAuto* (разд. 3.3);
- библиотеки *MetaAuto*, полностью описывающей автоматы и обеспечивающей сериализацию их в формат `XML` (разд. 3.5);
- `XSLT`-процессора, обеспечивающего преобразование полученного `XML`-файла, в соответствии с `XSLT`-шаблоном, в исходный код программы. В настоящем проекте предполагается использование внешнего (запускаемого отдельно) `XSLT`-процессора (разд. 3.7);
- программы *nmake* (разд. 3.8).

3.2. Компонента *Visio2Xml*

Компонента *Visio2Xml* является консольным приложением, реализованным на языке *C#*. Данная компонента принимает в качестве входных параметров пути к:

- файлу *MS Visio*, содержащему граф переходов автомата;
- файлу, который требуется записать;
- конфигурационному файлу, как необязательный параметр. В случае, когда данный параметр не указан, используется стандартный конфигурационный файл (разд. 3.4.3).

Отметим, что пути могут быть как абсолютными, так и относительными.

Исходные коды компоненты могут быть скачены с сайта http://is.ifmo.ru.

3.3. Компонента *Xml2Visio*

Компонента *Xml2Visio* является консольным приложением, реализованным на языке *C#*. Данная компонента принимает в качестве входных параметров пути к:

- `XML`-файлу, содержащему исходные данные;
- файлу *MS Visio*, который требуется создать;
- конфигурационному файлу, как необязательный параметр. В случае, когда данный параметр не указан, используется стандартный конфигурационный файл (разд. 3.4.3).

Отметим, что пути могут быть как абсолютными, так и относительными.

Исходные коды компоненты могут быть скачены с сайта http://is.ifmo.ru.

3.4. Формат и способ использования конфигурационного файла

3.4.1. Конфигурируемые элементы графа переходов

После анализа элементов графа переходов автомата, нотация которого представлена в разд. 1.3.2, выделены объекты, требующие возможности конфигурирования:

- список автоматов, вызываемых в состоянии. Необходимо иметь возможность задать синтаксис автоматного вызова, символы-разделители, пробельные символы;
- список действий, выполняющихся в состоянии. Необходимо иметь возможность задать синтаксис вызова действия, символы-разделители для разделения вызовов, пробельные символы;
- список действий, выполняющихся на ребре. Должен задаваться аналогично списку действий, выполняющихся в состоянии;

- условия перехода на ребре. Необходимо иметь возможность задать синтаксис условий на ребре, унарных и бинарных операций, а также открывающих и закрывающих скобок.

Отметим, что возможность конфигурирования синтаксиса автоматного вызова, вызова действия (выходной переменной) на ребре и вызова действия в состоянии стирает семантику, вкладываемую в эти элементы. Все они становятся «узлами» в списке или бинарном дереве логического выражения. Руководствуясь этим наблюдением, и создавался формат конфигурационного файла.

3.4.2. Формат конфигурационного файла

В данном инструментальном средстве используется `XML`-файл конфигурации. Этот файл имеет следующую структуру:

- корневой элемент – **`configuration`**. Он содержит элемент **`parsers`**;
- элемент **`parsers`** содержит, в свою очередь, список шаблонов «узлов», описание которых приведено ниже (элементы **`nodes`**/**`nodeTemplates`**), а также элементы **`descriptions`**, **`state`** и **`transition`**;
- элемент **`descriptions`** содержит список элементов **`node`**, определяющих возможные для задания описаний «узлов» шаблоны. Задание описаний «узлов» происходит с помощью элементов, представляющих из себя прямоугольник, разделенный на две части. На рис. 6 представлен такой элемент. В первой части прямоугольника содержится название описываемого «узла» (на рисунке – `z200`), а во втором – само описание.

Рис. 6. Элемент, задающий описание узла

По названию описываемого «узла» среди шаблонов, перечисленных в элементе **descriptions**, выбирается наиболее подходящий. Выбранный шаблон и определяет тип узла;

- элемент **state** содержит два дочерних элемента: **stateMachineRef** и **outputAction**, описывающих список автоматов, вызываемых в состоянии, и список действий, совершаемых в состоянии, соответственно;
- элемент **transition** содержит два дочерних элемента: **outputAction** и **condition**;
- элементы, описывающие списки выполняемых действий или списки автоматов содержат произвольное количество элементов **node**, **delimeter** и **space**;
- элемент **condition** содержит произвольное количество элементов **node**, **binaryOperation**, **unaryOperation**, **bracketStart**, **bracketEnd**.

Элемент **nodeTemplate** файла конфигурации задает объект класса `NodeElement` библиотеки *MetaAuto* и содержит следующие атрибуты:

template – название шаблона. Данное название будет установлено в качестве типа (`Type`) соответствующего объекта класса `NodeElement`;

regexp – регулярное выражение, служащее для проверки строк на соответствие данному шаблону;

format – строка, позволяющая преобразовывать данный элемент в строку. Может содержать подстановки вида «`#параметр#`», вместо которых будут подставлены реальные значения параметров (описание параметров шаблона приведено дальше);

name – строка, значение которой будет установлено в качестве свойства `Name` объекта класса `NodeElement`. Эта строка может использовать значения именованных областей, содержащихся в регулярном выражении (для

использования значения именованной области с именем **name**, необходимо написать **${name}**);

Также, элемент **nodeTemplate** файла конфигурации может содержать дочерние узлы **parameter**. Эти узлы содержат следующие атрибуты:

name – название параметра;

value – значение параметра. В данной строке можно использовать именованные области, заданные в атрибуте **regexp** родительского узла.

Операции (бинарные и унарные) также задаются с помощью регулярных выражений. За конфигурирование операций отвечают элементы **binaryOperation** и **unaryOperation** файла конфигурации. Эти элементы имеют следующие атрибуты:

- **regexp** – регулярное выражение;
- **type** – тип операции.

Пример создания фала конфигурации приведен в разд. 4.4.

3.4.3. Способ использования конфигурационного файла

Конфигурационный файл может передаваться как компоненте *Visio2Xml.exe*, так и компоненте *Xml2Visio.exe*. Заметим, что сами эти компоненты не используют конфигурационный файл, а лишь передают его в библиотеку *MetaAuto*.

В случае, когда конфигурационный файл не передан, используется стандартный конфигурационный файл, приведенный в приложении 1.

3.5. Библиотека *MetaAuto*

Библиотека *MetaAuto* содержит классы, обеспечивающие сохранение графов переходов автоматов, сериализацию этих графов, а также разбор различных строк, с помощью лексического и синтаксического анализаторов с использованием конфигурационного файла.

3.5.1. Диаграмма классов библиотеки *MetaAuto*

На рис. 7 приведена диаграмма классов для библиотеки *MetaAuto*.

Рис. 7. Диаграмма классов библиотеки *MetaAuto*

На рис. 7, как и в книге [11], используются следующие обозначения:

- класс представляется прямоугольником. Каждый прямоугольник разделен на три части. Эти части означают название, атрибуты и операции класса. Названия абстрактных классов выделены курсивом;
- стрелка с большим треугольником на конце означает обобщение. Класс у треугольника является обобщением классов на другом конце стрелки;
- стрелка с ромбом на конце означает агрегацию классов. Класс у ромба включает в себя класс с противоположной стороны стрелки. При этом закрашенный ромб означает композицию – строгую агрегацию, характеризующуюся тем, что объект включаемого класса может принадлежать только одному объекту включающего класса.
- стрелка с пунктирной линией означает зависимость. Класс, на который указывает стрелка, является зависимым от класса, из которого стрелка исходит;
- обычная стрелка означает ассоциацию с навигацией. Класс, находящийся с противоположной от стрелки стороны, должен иметь ссылку на класс, к которому направлена стрелка.

Цифры и символ * на стрелках агрегации и ассоциации показывают кратность включения или ассоциации соответственно. Символ * при этом означает любое число от нуля до бесконечности. Названия на упомянутых ссылках соответствуют названию переменной, через которую происходит доступ к объектам соответствующего класса.

На диаграмме представлены только основные отношения между классами. Более подробную информацию о составе и отношениях между классами можно получить из исходного кода программы.

Диаграмма составлена с помощью редактора *MS Visio*.

Приведем краткую характеристику классов, применяемых в этой диаграмме.

Класс `Model`

Основной класс библиотеки. Содержит в себе информацию обо всех автоматах проекта.

Класс `Automata`

Класс содержит информацию о конкретном автомате.

Класс `AutomataCollection`

Класс является коллекцией автоматов. Автоматы идентифицируются по названиям.

Класс `NodeDescriptions`

Содержит описание входных, выходных переменных, других элементов списков и логических условий, присутствующих в автомате. Каждое описание специфицируется типом элемента, к которому данное описание относится и названием этого элемента.

Класс `State`

Класс содержит информацию о состоянии или групповом состоянии автомата. Групповое состояние отличается от обычного тем, что число детей, определяемое свойством `Children.Count`, у него не нулевое.

Класс `States`

Класс является коллекцией состояний.

Класс `Connections`

Класс является коллекцией переходов (объектов класса `Connector`).

Класс NodeElement

Как следует из диаграммы классов (рис. 7), в библиотеке не выделено отдельных классов для входных и выходных переменных, вызовов автоматов и конкретных логических операций (И, ИЛИ, НЕ). Вместо этого, введены классы NodeElement, UnaryOperation и BinaryOperation.

Первый из этих классов (класс NodeElement) представляет входные и выходные переменные, вызовы автоматов, проверки состояний других автоматов, а также любые другие элементы списков действий, выполняющихся в состоянии и на ребре, и переменных, участвующих в логических выражениях. Тип элемента определяется с помощью свойства Type.

Класс ActionNode

Класс описывает элемент списка. Используется при задании списка выполняемых в состоянии автоматов и выполняемых действий, а также при задании действий, выполняемых при переходе.

Класс Connector

Класс содержит информацию о переходе – приоритет, условие и действие на переходе.

Класс Connectors

Класс является коллекцией переходов. Коллекция принадлежит одному из состояний, а переходы в ней идентифицируются другим состоянием. Таким образом, для того, чтобы узнать, есть ли переход из состояния A в состояние B необходимо проверить, содержит ли коллекция, принадлежащая состоянию A, переход по ключу B.

Класс `Condition`

Класс описывает элемент условного выражения. Является базовым для классов `ConditionNode`, `BinaryOperation` и `UnaryOperation`.

Класс `ConditionNode`

Класс описывает листовой элемент бинарного дерева, соответствующего условному выражению.

Классы `UnaryOperation` и `BinaryOperation`

Классы `UnaryOperation` и `BinaryOperation` представляют собой унарную и бинарную операции бинарного дерева, соответствующего условному выражению, соответственно. Тип операции определяется свойством `Type`.

Класс `RegexpParser`

Парсер регулярных выражений.

Классы `ConditionParser`, `ActionParser`, `SyntacticalAnalyzer` и `LexicalAnalyzer`, `A0`, `A1` и `BaseAutomata` будут рассмотрены в разд. 3.6.

3.6. Парсеры

В программе применяются различные парсеры и анализаторы строк. Среди них:

- два синтаксических анализатора (синтаксический анализатор для разбора линейного списка и синтаксический анализатор для разбора логического выражения);
- лексический анализатор;
- анализатор регулярных выражений.

3.6.1. Синтаксические анализаторы (общие принципы)

В программе используются два синтаксических анализатора:

- синтаксический анализатор для разбора линейного списка (класс `ActionParser`);
- синтаксический анализатор для разбора логического выражения (класс `ConditionParser`).

Оба анализатора основаны на совместном применении автоматного программирования и LL(1)-грамматики. Подробно совместное использование LL(1)-грамматики и SWITCH-технологии описано в работах [21, 22].

Классы, реализующие синтаксические анализаторы обоих типов, наследуются от общего автоматного класса, описанного в работе [21]. На рис. 8 приводится схема связей автомата лексического анализатора, а на рис. 9 – граф его переходов.

Код автоматных классов, реализующих синтаксические анализаторы, был получен в процессе тестирования библиотеки *MetaAuto*. Тестовое приложение создавало объекты библиотеки и заполняло свойства всех ее объектов. Далее было произведено преобразование объектов библиотеки в `XML`-формат, по которому был сгенерирован код автоматных классов.

Схема связей (рис. 8) и граф переходов (рис. 9) были взяты без изменений из работы [21].

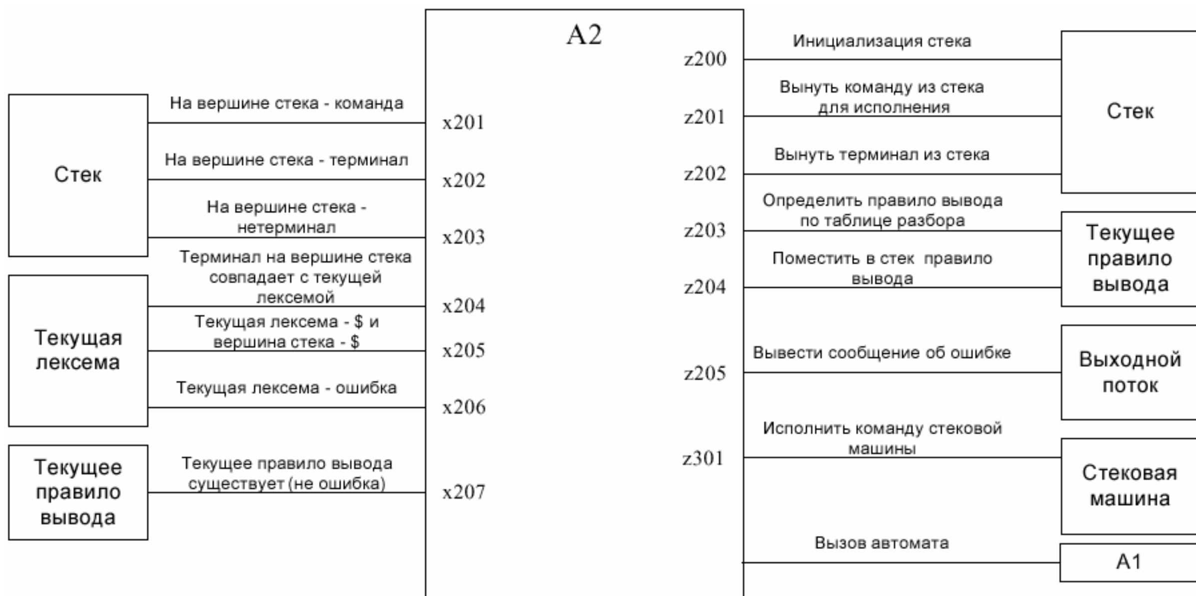
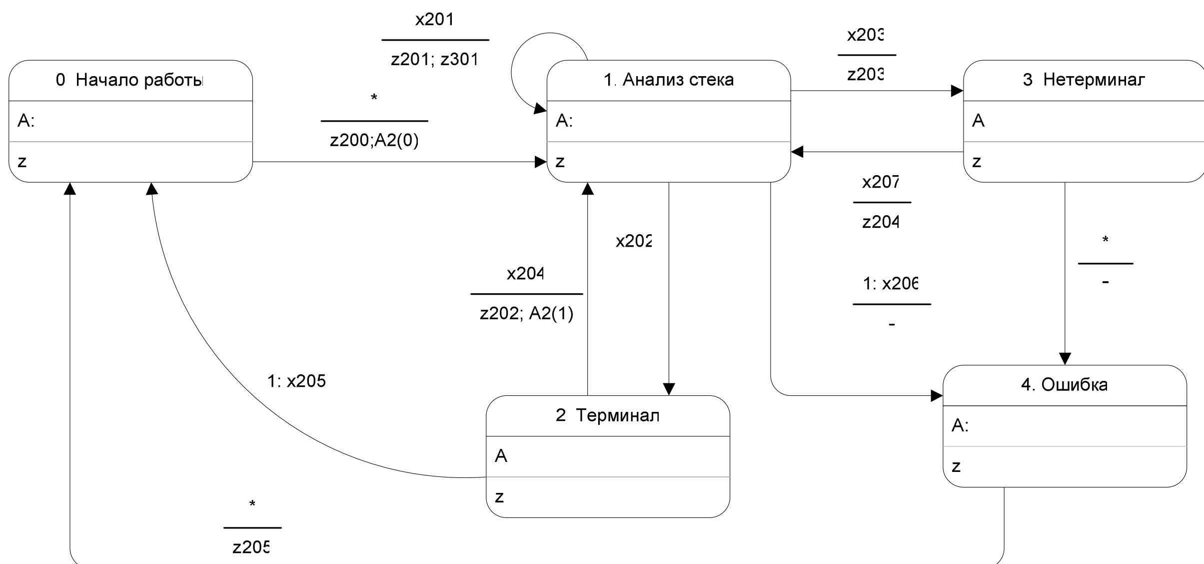


Рис. 8. Схема связей автомата синтаксического анализатора



z200	Инициализация стека
z201	Вынуть команду из стека
z202	Определить терминал стека
z203	Определить правило вывода по таблице разбора
z204	Поместить в стек правило вывода
z205	Вывести сообщение об ошибке
z301	Исполнить команду стековой машины

x201	На вершине стека - команда
x202	На вершине стека - терминал
x203	На вершине стека - нетерминал
x204	Терминал на вершине стека совпадает с текущей лексемой
x205	Текущая лексема - $ и вершина стека $
x206	Текущая лексема - ошибка
x207	Текущее правило вывода существует (не ошибка)

Автомат	
Имя	A1
Название	Синтаксический анализатор

Рис. 9. Граф переходов автомата синтаксического анализатора

3.6.2. Синтаксический анализатор для разбора линейного списка

В автомате синтаксического анализатора линейного списка применяется следующая LL(1)-грамматика:

G = {N;T;O;S;R},

где N={S,E} – набор нетерминальных символов,

T={delimiter,V,$} – множество терминальных символов,

O={@c} – набор операционных символов,

R – правила.

Единственный операционный символ @c имеет следующий смысл – создать новый элемент линейного списка. Поместить его в конец списка.

Грамматика содержит следующие правила:

```
R = {     1)    S → @c V E,
          2)    S → $
          3)    E → delimiter
          4)    E → $               }
```

Описанной грамматике соответствует управляющая таблица (табл. 1).

Таблица 1. Управляющая таблица грамматики для разбора линейного списка

	V	Delimiter	$
S	1	0	2
E	0	1	4

3.6.3. Синтаксический анализатор для разбора логического выражения

Грамматика, используемая в автомате синтаксического анализатора логического выражения, похожа на грамматику, используемую в работе [22] для разбора произвольных логических выражений. Отличие состоит в том, что не разделяются операции & и |. Терминалы, использующиеся для обозначения этих операций, заменены терминалом binaryOp. Для сохранения типа текущей

операции введен новый операционный символ @s. Таким образом, грамматика принимает следующий вид:

G = {N;T;O;S;R}

Грамматика G состоит из набора нетерминальных символов N={S,E,F,G}, множества терминальных символов T={unaryOp,binaryOp, (,),V,$}, набора операционных символов O={@a,@o,@n,@c,@s}, имеющих следующий смысл:

@a – извлечь дерево из стека и установить его как результат разбора;

@o – создать дерево. Вытолкнуть из стека операций название операции и пометить этим названием корневую вершину созданного дерева. Считать из стека адрес правого потомка для корневой вершины созданного дерева и вытолкнуть его. Считать из стека адрес левого потомка для корневой вершины созданного дерева и вытолкнуть его. Поместить в стек адрес этого дерева;

@n – создать дерево. Вытолкнуть из стека операций название операции и пометить этим названием корневую вершину созданного дерева. Считать из стека адрес правого потомка для корневой вершины созданного дерева и вытолкнуть его. Поместить в стек адрес этого дерева;

@c – создать дерево и поместить его адрес в стек стековой машины; корневую вершину пометить текущей лексемой из лексического анализатора;

@s – записать в стек операций текущую операцию из лексического анализатора.

Грамматика содержит следующие правила:

```
R = {    1)   S → E @a S,
         2)   S → $,
         3)   E → G F,
         4)   F → @s binaryOp G @o F,
         5)   F → ε,
         6)   G → @s unaryOp G @n,
         7)   G → ( E ),
         8)   G → @c V                       }
```

Описанной грамматике соответствует управляющая таблица (табл. 2).

Таблица 2. Управляющая таблица грамматики для разбора линейного списка

	unaryOp	binaryOp	(	)	V	$
S	1	0	1	0	1	2
E	3	0	3	0	1	0
F	0	4	0	5	0	5
G	6	0	7	0	8	0

3.6.4. Лексический анализатор

В программе используется один лексический анализатор. На рис. 10 приведена схема связей автомата, а на рис. 11 – граф его переходов. Для использования его в разных случаях (для списка вложенных автоматов, списка выполняемых в состоянии действий и списка действий на переходе) анализатор параметризуется из конфигурационного файла.

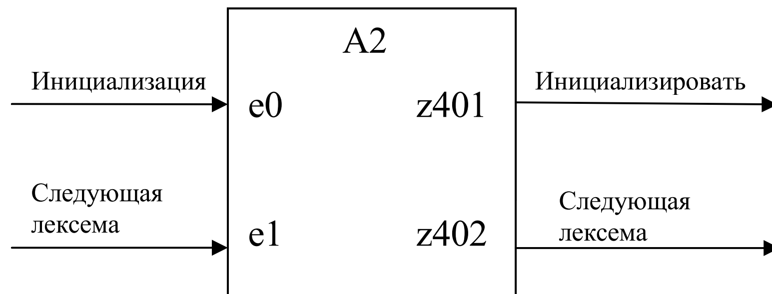


Рис. 10. Схема связей автомата лексического анализатора

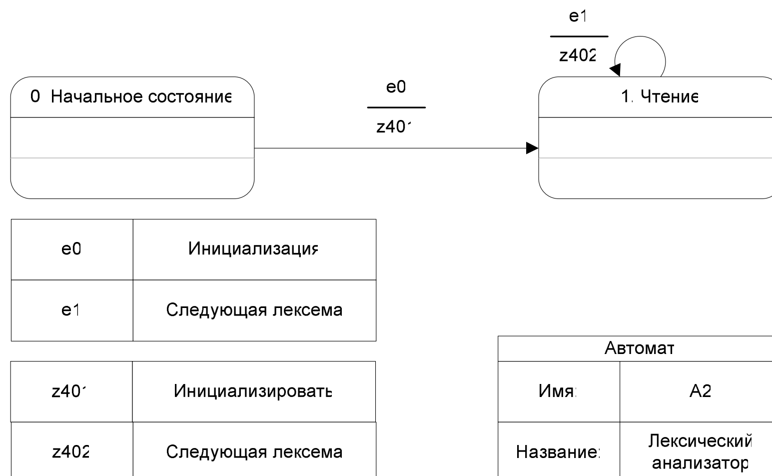


Рис. 11. Граф переходов автомата лексического анализатора

3.6.5. Формат выходного `XML`-файла

Формат `XML`-файла, использующегося для описания автоматов в системе, разработан на основе нотации, описанной в разд. 1.3.2. За основу взят формат `XML`, применяемый инструментальным средством *UniMod* для экспорта автоматов. Данный `XML`-файл получается в результате сериализации объекта класса `Model` библиотеки *MetaAuto*.

Элемент верхнего уровня

`model`

Название элемента верхнего уровня совпадает с названием класса объекта, из которого он получается.

Список используемых элементов

- ***model***
- ***stateMachine***
- ***nodeDescriptions***
- ***node***
- ***state***
- ***stateMachineRef***
- ***transition***
- ***condition***
- ***outputAction***
- ***conditionNode***
- ***operation***
- ***actionNode***
- ***parameter***

Описание используемых элементов

- ***model***

Элемент верхнего уровня `XML`-документа. Содержит все автоматы рассматриваемого проекта. В качестве атрибутов выступают имя модели и ее описание.

Атрибуты:

name	CDATA	атрибут не обязателен
description	CDATA	атрибут не обязателен

Дочерние элементы:

stateMachine	любое количество

- ***stateMachine***

Элемент описывает автомат. В качестве атрибутов выступают название автомата и его описание. Содержит в себе описания используемых в автомате входных и выходных переменных (`nodeDescriptions`), список переходов (`transition`) и основную группу состояний. Основная группа состояний – это виртуальная группа, которая включает в себя все состояния автомата.

Атрибуты:

name	CDATA	обязательный атрибут
description	CDATA	атрибут не обязателен

Дочерние элементы:

nodeDescriptions	ровно один
state	ровно один
transition	любое количество

- ***nodeDescriptions***

Включает в себя список описаний для входных и выходных переменных.

Атрибуты:

нет атрибутов

Дочерние элементы:

node	любое количество

- ***node***

Содержит описание для входной или выходной переменной. Атрибуты указывают тип переменной, название и описание. Имя и тип должны совпадать с именем и типом в элементах `actionNode` и `conditionNode`.

Атрибуты:

type	CDATA	обязательный атрибут
name	CDATA	обязательный атрибут
description	CDATA	обязательный атрибут

Дочерние элементы:

нет дочерних элементов

- ***state***

Состояние автомата или группа состояний. В качестве атрибутов выступают название и описание состояния. Может включать в себя другие состояния. Также может содержать список вложенных автоматов и список действий.

Атрибуты:

name	CDATA	обязательный атрибут
description	CDATA	атрибут не обязателен

Дочерние элементы:

state	любое количество
stateMachineRef	не больше одного раза
outputAction	не больше одного раза

- ***stateMachineRef***

Список вложенных в состояние автоматов.

Атрибуты:

нет атрибутов

Дочерние элементы:

actionNode любое количество

- ***transition***

Элемент описывает переход или групповой переход в автомате. С помощью атрибутов задаются приоритет перехода, начальное и конечное состояния, а также описание перехода. Может содержать в себе условие перехода и действия на ребре.

Атрибуты:

priority	CDATA	атрибут не обязателен
sourceRef	CDATA	обязательный атрибут
targetRef	CDATA	обязательный атрибут
description	CDATA	атрибут не обязателен

Дочерние элементы:

condition не больше одного раза

outputAction не больше одного раза

- ***condition***

Условие перехода. Условие перехода представляет собой логическое выражение, представленное в виде бинарного дерева. Листом данного дерева является узел `conditionNode`. Нелистовой элемент – логическая операция – представлена элементом `operation`.

Атрибуты:

нет атрибутов

Дочерние элементы:

conditionNode|operation один из двух

- ***outputAction***

Список действий на ребре.

Атрибуты:

нет атрибутов

Дочерние элементы:

actionNode	любое количество

conditionNode

Лист дерева, представляющего логическое выражение. Может содержать параметры, специфицирующие переменную. Подробности описаны в разд. 2.3.

Атрибуты:

name	ID	обязательный атрибут
type	CDATA	обязательный атрибут
description	CDATA	атрибут не обязателен

Дочерние элементы:

parameter	любое количество

- ***operation***

Бинарная операция. Нелистовой элемент дерева, представляющего логическую операцию. Может иметь в качестве дочерних элементов узел `conditionNode` или другой узел `operation`.

Атрибуты:

type	CDATA	обязательный атрибут

Дочерние элементы:

conditionNode	любое количество
operation	любое количество

- ***actionNode***

Выходная переменная, автоматный вызов или другое действие на переходе.

Атрибуты:

name	ID	обязательный атрибут
type	CDATA	обязательный атрибут
description	CDATA	атрибут не обязателен

Дочерние элементы:

`parameter`	любое количество

- ***`parameter`***

Параметр входных и выходных переменных. Подробности описаны в разд. 3.6.5.

Атрибуты:

`name`	ID	обязательный атрибут
`value`	CDATA	обязательный атрибут

Дочерние элементы:

`operation`	любое количество
`parameter`	любое количество

3.7. `XSLT`-процессор

В состав инструментального средства включено консольное приложение *XSLTransformer*, которое реализованно на языке *C#*. Оно осуществляет `XSLT`-преобразование. Данное консольное приложение использует процессор *MSXML* и принимает в качестве входных параметров пути к:

- `XML`-файлу, содержащему исходные данные;
- `XSLT`-шаблону;
- генерируемому файлу.

Отметим, что пути могут быть как абсолютными, так и относительными.

Исходные коды компоненты размещены на сайте http://is.ifmo.ru.

3.8. Программа *nmake*

В состав инструментального средства включена программа *nmake* версии 7.10.3077.0, взятая из дистрибутива *Microsoft Visual Studio 2003*. Данная программа для своей работы требует наличия в текущем каталоге файла *makefile*, имеющего определенный формат [23].

Глава 4. Примеры использования программы

В этой главе будут рассмотрены следующие вопросы:

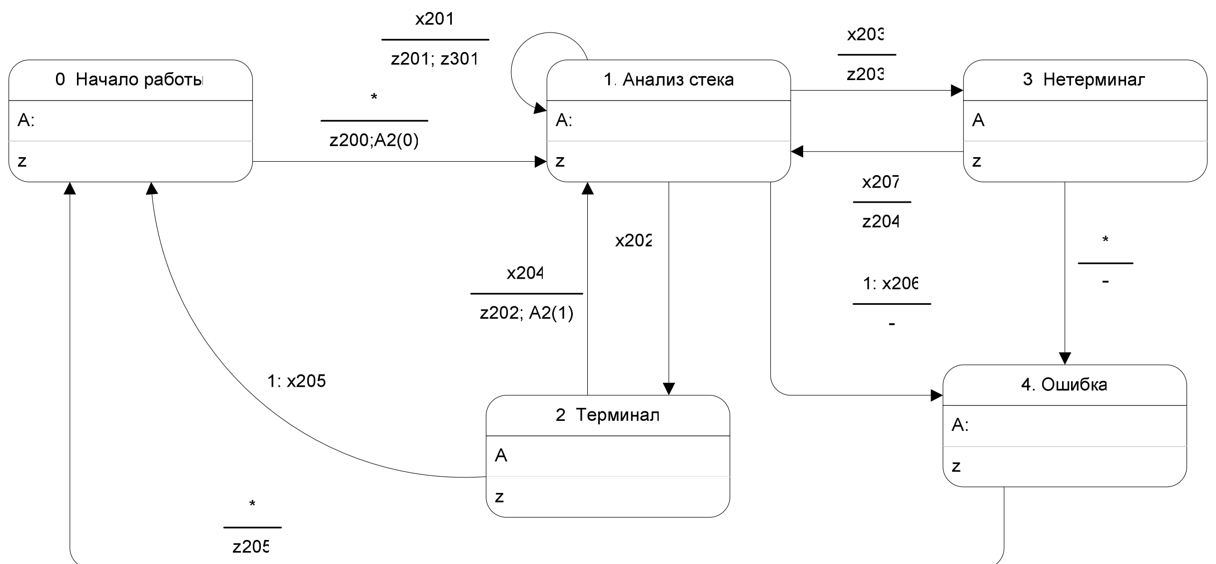
- методика использования инструментального средства;
- пример интеграции инструментального средства со средой разработки *MS Visual Studio*;
- процесс создания XSLT-шаблонов;
- процесс создания конфигурационного файла.

4.1. Методика использования инструментального средства

Опишем методику генерации кода с использованием разработанного инструментального средства. Для примера преобразуем графы переходов, отвечающие за синтаксический и лексический анализаторы (разд. 3.6.1 и 3.6.4) в исходный код программы.

4.1.1. Шаг 1. Изображение графов переходов

Изобразим графы переходов требуемых автоматов в редакторе *MS Visio*. Графы переходов автоматов `A1` (рис. 12) и `A2` (рис. 13) расположим на страницах с названиями A1, и A2 соответственно. Сохраним графы переходов в файл *analizers.vsd*.

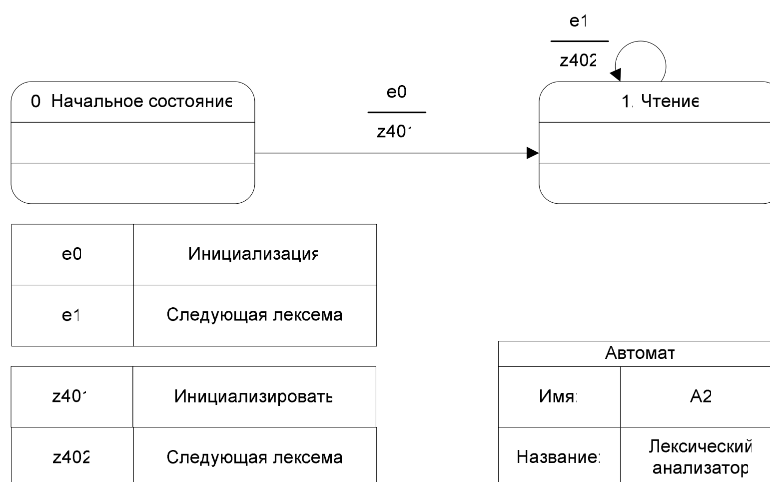


z200	Инициализация стека
z201	Вынуть команду из стека
z202	Определить терминал стека
z203	Определить правило вывода по таблице разбора
z204	Поместить в стек правило вывода
z205	Вывести сообщение об ошибке
z301	Исполнить команду стековой машины

x201	На вершине стека - команда
x202	На вершине стека - терминал
x203	На вершине стека - нетерминал
x204	Терминал на вершине стека совпадает с текущей лексемой
x205	Текущая лексема - $ и вершина стека $
x206	Текущая лексема - ошибка
x207	Текущее правило вывода существует (не ошибка)

Автомат	
Имя:	A1
Название:	Синтаксический анализатор

Рис. 12. Граф переходов автомата A1 (страница A1 файла *analizers.vsd*)



e0	Инициализация
e1	Следующая лексема

z401	Инициализировать
z402	Следующая лексема

Автомат	
Имя:	A2
Название:	Лексический анализатор

Рис. 13. Граф переходов автомата A2 (страница A2 файла *analizers.vsd*)

4.1.2. Шаг 2. Преобразования в XML-формат

Для преобразования полученного на первом шаге файла *analizers.vsd* в XML-формат воспользуемся компонентой *Visio2Xml.exe*, являющейся частью инструментального средства. Для запуска из командной сроки воспользуемся следующей командой:

Visio2Xml.exe analizers.vsd automatas.xml

Получим XML-файл, представленный ниже:

```xml
<?xml version="1.0"?>
<model name="Visio project">
  <stateMachine name="A2" description="Лексический анализатор">
    <nodeDescriptions>
      <node type="EVENT" name="0" description="Инициализация" />
      <node type="EVENT" name="1" description="Следующая лексема" />
      <node type="OUTPUT_ACTION" name="402" description="Следующая лексема" />
      <node type="OUTPUT_ACTION" name="401" description="Инициализировать" />
    </nodeDescriptions>
    <state name="Top" description="">
      <state name="0" description="Начальное состояние" />
      <state name="1" description="Чтение" />
    </state>
    <transition sourceRef="0" targetRef="1">
      <condition>
        <conditionNode name="0" type="EVENT">
          <parameter name="name" value="0" />
        </conditionNode>
      </condition>
      <outputAction>
        <actionNode name="401" type="OUTPUT_ACTION">
          <parameter name="name" value="401" />
        </actionNode>
      </outputAction>
    </transition>
    <transition sourceRef="1" targetRef="1">
      <condition>
        <conditionNode name="1" type="EVENT">
          <parameter name="name" value="1" />
        </conditionNode>
      </condition>
      <outputAction>
        <actionNode name="402" type="OUTPUT_ACTION">
          <parameter name="name" value="402" />
        </actionNode>
      </outputAction>
    </transition>
  </stateMachine>
  <stateMachine name="A1" description="Синтаксический анализатор">
    <nodeDescriptions>
      <node type="INPUT_VARIABLE" name="207"
            description="Текущее правило вывода существует (не ошибка)" />
      <node type="INPUT_VARIABLE" name="201"
            description="На вершине стека - команда" />
      <node type="INPUT_VARIABLE" name="202"
        description="На вершине стека - терминал" />
```

```
    <node type="INPUT_VARIABLE" name="203"
      description="На вершине стека - нетерминал" />
    <node type="INPUT_VARIABLE" name="204"
      description="Терминал на вершине стека совпадает с текущей лексемой" />
    <node type="INPUT_VARIABLE" name="205"
      description="Текущая лексема - $ и вершина стека $" />
    <node type="INPUT_VARIABLE" name="206"
      description="Текущая лексема - ошибка" />
    <node type="OUTPUT_ACTION" name="301"
      description="Исполнить команду стековой машины" />
    <node type="OUTPUT_ACTION" name="200" description="Инициализация стека" />
    <node type="OUTPUT_ACTION" name="201"
      description="Вынуть команду из стека" />
    <node type="OUTPUT_ACTION" name="202"
      description="Определить терминал стека" />
    <node type="OUTPUT_ACTION" name="203"
      description="Определить правило вывода по таблице разбора" />
    <node type="OUTPUT_ACTION" name="204"
      description="Поместить в стек правило вывода" />
    <node type="OUTPUT_ACTION" name="205"
      description="Вывести сообщение об ошибке" />
  </nodeDescriptions>
  <state name="Top" description="">
    <state name="2" description="Терминал" />
    <state name="3" description="Нетерминал" />
    <state name="0" description="Начало работы" />
    <state name="1" description="Анализ стека" />
    <state name="4" description="Ошибка" />
  </state>
  <transition sourceRef="2" targetRef="0" priority="1">
    <condition>
      <conditionNode name="205" type="INPUT_VARIABLE">
        <parameter name="name" value="205" />
      </conditionNode>
    </condition>
  </transition>
  <transition sourceRef="2" targetRef="1">
    <condition>
      <conditionNode name="204" type="INPUT_VARIABLE">
        <parameter name="name" value="204" />
      </conditionNode>
    </condition>
    <outputAction>
      <actionNode name="202" type="OUTPUT_ACTION">
        <parameter name="name" value="202" />
      </actionNode>
      <actionNode name="A2e1" type="AUTOMATA_CALL">
        <parameter name="automata" value="2" />
        <parameter name="event" value="1" />
      </actionNode>
    </outputAction>
  </transition>
  <transition sourceRef="3" targetRef="4" />
  <transition sourceRef="3" targetRef="1">
    <condition>
      <conditionNode name="207" type="INPUT_VARIABLE">
        <parameter name="name" value="207" />
      </conditionNode>
    </condition>
    <outputAction>
      <actionNode name="204" type="OUTPUT_ACTION">
```

```xml
      <parameter name="name" value="204" />
    </actionNode>
  </outputAction>
</transition>
<transition sourceRef="0" targetRef="1">
  <outputAction>
    <actionNode name="200" type="OUTPUT_ACTION">
      <parameter name="name" value="200" />
    </actionNode>
    <actionNode name="A2e0" type="AUTOMATA_CALL">
      <parameter name="automata" value="2" />
      <parameter name="event" value="0" />
    </actionNode>
  </outputAction>
</transition>
<transition sourceRef="1" targetRef="3">
  <condition>
    <conditionNode name="203" type="INPUT_VARIABLE">
      <parameter name="name" value="203" />
    </conditionNode>
  </condition>
  <outputAction>
    <actionNode name="203" type="OUTPUT_ACTION">
      <parameter name="name" value="203" />
    </actionNode>
  </outputAction>
</transition>
<transition sourceRef="1" targetRef="2">
  <condition>
    <conditionNode name="202" type="INPUT_VARIABLE">
      <parameter name="name" value="202" />
    </conditionNode>
  </condition>
</transition>
<transition sourceRef="1" targetRef="4" priority="1">
  <condition>
    <conditionNode name="206" type="INPUT_VARIABLE">
      <parameter name="name" value="206" />
    </conditionNode>
  </condition>
</transition>
<transition sourceRef="1" targetRef="1">
  <condition>
    <conditionNode name="201" type="INPUT_VARIABLE">
      <parameter name="name" value="201" />
    </conditionNode>
  </condition>
  <outputAction>
    <actionNode name="201" type="OUTPUT_ACTION">
      <parameter name="name" value="201" />
    </actionNode>
    <actionNode name="301" type="OUTPUT_ACTION">
      <parameter name="name" value="301" />
    </actionNode>
  </outputAction>
</transition>
<transition sourceRef="4" targetRef="0">
  <outputAction>
    <actionNode name="205" type="OUTPUT_ACTION">
      <parameter name="name" value="205" />
    </actionNode>
```

```
      </outputAction>
    </transition>
  </stateMachine>
</model>
```

4.1.3. Шаг 3. Создание XSLT-шаблона

Для получения кода на языке *C#* необходимо создать соответствующий шаблон. Способ создания шаблона приведен в разд. 4.3, используемый шаблон – в приложении 2.

4.1.4. Шаг 4. Получение исходного кода

Преобразуем полученный XML файл в исходный код с помощью XSLT-шаблона. Воспользуемся компонентой *XslTransform.exe*. Для ее запуска воспользуемся следующей командной строкой:

XslTransform.exe automatas.xml automatas.cs.xslt automatas.cs

В результате получаем файл *automatas.cs*, содержащий исходный код на языке *C#*. Листинг файла представлен ниже:

```
//--- this file is machine generated ---
//Model: Visio project

namespace Automatas
{
    public class BaseAutomata
    {
    }

    /// <summary>
    /// Лексический анализатор
    /// </summary>
    public abstract class A2 : BaseAutomata
    {
        protected string y = "0";

        public void A(int e)
        {
            switch (y)
            {
                case "0":
                    if      (e == 0)            {z401();  y = "1";}
                break;

                case "1":
                    if      (e == 1)            {z402();  y = "1";}
                break;
            }
        }
```

```
        /// <summary>
        /// Инициализировать
        /// </summary>
        protected abstract void z401();

        /// <summary>
        /// Следующая лексема
        /// </summary>
        protected abstract void z402();
}

/// <summary>
/// Синтаксический анализатор
/// </summary>
public abstract class A1 : BaseAutomata
{
    protected string y = "0";

    public void A(int e)
    {
        switch (y)
        {
            case "0":
                if      (true)          {z200(); Call_2(0);  y = "1";}
            break;

            case "1":
                if      (x206())              { y = "4";}
                else if (x201())              {z201(); z301();  y = "1";}
                else if (x202())              { y = "2";}
                else if (x203())              {z203();  y = "3";}
            break;

            case "2":
                if      (x205())              { y = "0";}
                else if (x204())              {z202(); Call_2(1);  y = "1";}
            break;

            case "3":
                if      (x207())              {z204();  y = "1";}
                else if (true)          { y = "4";}
            break;

            case "4":
                if      (true)          {z205();  y = "0";}
            break;
        }
    }

        /// <summary>
        /// Инициализация стека
        /// </summary>
        protected abstract void z200();

        /// <summary>
        /// Вынуть команду из стека
        /// </summary>
        protected abstract void z201();
```

```
/// <summary>
/// Определить терминал стека
/// </summary>
protected abstract void z202();

/// <summary>
/// Определить правило вывода по таблице разбора
/// </summary>
protected abstract void z203();

/// <summary>
/// Поместить в стек правило вывода
/// </summary>
protected abstract void z204();

/// <summary>
/// Вывести сообщение об ошибке
/// </summary>
protected abstract void z205();

/// <summary>
/// Исполнить команду стековой машины
/// </summary>
protected abstract void z301();

/// <summary>
/// Вызов реализации автомата 2.
/// </summary>
protected abstract void Call_2(int e);

/// <summary>
/// На вершине стека - команда
/// </summary>
/// <returns>True если условие выполнено, false - в противном
                                                случае</returns>
protected abstract bool x201();

/// <summary>
/// На вершине стека - терминал
/// </summary>
/// <returns>True если условие выполнено, false - в противном
                                                случае</returns>
protected abstract bool x202();

/// <summary>
/// На вершине стека - нетерминал
/// </summary>
/// <returns>True если условие выполнено, false - в противном
                                                случае</returns>
protected abstract bool x203();

/// <summary>
/// Терминал на вершине стека совпадает с текущей лексемой
/// </summary>
/// <returns>True если условие выполнено, false - в противном
                                                случае</returns>
protected abstract bool x204();

/// <summary>
/// Текущая лексема - $ и вершина стека $
/// </summary>
```

```
        /// <returns>True если условие выполнено, false - в противном
                                                            случае</returns>
        protected abstract bool x205();

        /// <summary>
        /// Текущая лексема - ошибка
        /// </summary>
        /// <returns>True если условие выполнено, false - в противном
                                                            случае</returns>
        protected abstract bool x206();

        /// <summary>
        /// Текущее правило вывода существует (не ошибка)
        /// </summary>
        /// <returns>True если условие выполнено, false - в противном
                                                            случае</returns>
        protected abstract bool x207();

    }

}
```

4.1.5. Шаг 5. Преобразование XML-файла в документ *MS Visio*

Для преобразования XML-файла в документ *MS Visio* необходимо воспользоваться компонентой *Xml2Visio.exe*. Для ее запуска воспользуемся следующей командной строкой:

Xml2Visio.exe automatas.xml analyzers.new.vsd

В результате получится файл, отличающийся от исходного только расположением состояний.

4.2. Интеграция инструментального средства со средой разработки *MS Visual Studio*

Интеграцию разработанного инструментального средства со средой разработки *MS Visual Studio* рассмотрим на примере из предыдущего раздела.

Интеграция инструментального средства предполагает внедрение его в процесс компиляции проекта. Для обеспечения такой интеграции необходимо создать файл *makefile*, служащий для автоматизации процесса создания файла с исходным кодом. В рассматриваемом примере необходимо автоматизировать шаги 2 и 4. Для рассматриваемого примера *makefile* может выглядеть следующис образом:

```
..\Parsers\automatas.cs : automatas.xml automatas.cs.xslt
    "XSLTransform\XSLTransform.exe" automatas.xml automatas.cs.xslt "..\Parsers\automatas.cs"
automatas.xml : ..\Parsers\Analyzers.vsd Visio2Xml.exe.xml
    "Visio2Xml\Visio2Xml.exe" "..\Parsers\Analyzers.vsd" automatas.xml Visio2Xml.exe.xml
```

Отметим, что пути ко всем файлам в приведенном *makefile* несколько изменены. Это связано со спецификой расположения файлов в проекте. Также добавился файл конфигурации, передаваемый компоненте *Visio2Xml.exe*.

Для интеграции в процесс компиляции требуется:

- добавить в проект необходимые для преобразования файлы;
- задать командную строку, выполняющую преобразование в качестве события, выполняющегося до компиляции. Для этого необходимо открыть свойства проекта и задать `Pre-Build Event Command Line`:

```
cd "$(ProjectDir)MetaAuto"
"$(ProjectDir)MetaAuto/nmake.exe"
```

Для тестирования данного способа интеграции, инструментальное средство было внедрено в процесс компиляции самого себя. Исходные коды с реализованной интеграцией размещены на сайте http://is.ifmo.ru.

4.3. Создание XSLT-шаблонов

В качестве примера работы с инструментальным средством, разработаем шаблон для генерации кода на языке *C#*. Этот шаблон был применен для генерации исходного кода синтаксического и лексического анализаторов.

Ниже приведен код, который необходимо получить (листинг 3). Согласно диаграмме классов, представленной на рис. 7, автоматы реализуются как абстрактные классы. Входные и выходные переменные создаются, как чисто виртуальные функции и должны быть реализованы в классах-наследниках. В приведенном коде пропущена часть генерируемых входных и выходных переменных.

Листинг 3. Пример получаемого кода

```
//--- this file is machine generated ---
//Model: ModelName
namespace Automatas
{
    public class BaseAutomata
    {
    }
    /// <summary>
    /// Lexical analyzer
    /// </summary>
    public abstract class A2 : BaseAutomata
    {
        protected string y = "0";
        public void A(int e)
        {
            switch (y)
            {
                case "0":
                    if      (e == 0)            {z401();  y = "1";}
                break;
                case "1":
                    if      (e == 1)            {z200();  y = "1";}
                    else if (e == 0)            {z401();  y = "1";}
                break;
            }
        }
        /// <summary>
        ///
        /// </summary>
        protected abstract void z200();

        /// <summary>
        /// Initialize and return the first match
        /// </summary>
        protected abstract void z401();

    }

    /// <summary>
    /// Syntactical analyzer
    /// </summary>
    public abstract class A1 : BaseAutomata
    {
        protected string y = "0";
        public void A(int e)
        {
            switch (y)
            {
                case "0":
                    if      (true)         {z200(); Call_A2(0);  y = "1";}
                break;
                case "1":
                    if      (x206())            { y = "4";}
                    else if (x201())            {z201(); z301();  y = "1";}
                    else if (x202())            { y = "2";}
                    else if (x203())            {z203();  y = "3";}
                break;
```

```
            case "2":
                if      (x205())            { y = "0";}
                else if (x204())            {z202(); Call_A2(1);  y = "1";}
                else if (true)          { y = "4";}
            break;
            case "3":
                if      (x207())            {z204();  y = "1";}
                else if (true)          { y = "4";}
            break;
            case "4":
                if      (true)          {z205();  y = "0";}
            break;
        }
    }

        /// <summary>
        /// Command in the top of the stack
        /// </summary>
        /// <returns>Is condition correct</returns>
        protected abstract bool x201();

        /// <summary>
        /// Initialize and return the first match
        /// </summary>
        protected abstract void z401();

        /// <summary>
        /// Calls the automata A2 realization
        /// </summary>
        protected abstract void Call_A2(int e);

        /*Часть входных переменных и действий пропущено*/

    }

}
```

Анализ кода показывает, что в шаблоне необходимо производить следующие действия:

- получать множество использующихся в автомате входных и выходных переменных;
- получать список переходов из состояния;

Согласно формату XML-данных, приведенному в разд. 3.6.5. Это обеспечивают следующие шаблоны:

```
<xsl:key name="distinctActions" match="//actionNode" use="@name"/>
<xsl:key name="distinctActions" match="//actionNode" use="@name"/>

<xsl:template match="stateMachine">

    <xsl:variable name="stateMachineName" select="@name"/>

    <xsl:apply-templates
        select="//actionNode
```

```
            [generate-id(.) = generate-id(key('distinctActions', @name)
            [ancestor::stateMachine/@name=$stateMachineName])]"
        mode="FUNCTION_DEFINITIONS">
         <xsl:sort select="@type"/>
         <xsl:sort select="@name"/>
    </xsl:apply-templates>

    <xsl:apply-templates
        select="//conditionNode
            [generate-id(.) = generate-id(key('distinctConditions', @name)
            [ancestor::stateMachine/@name=$stateMachineName])]"
        mode="FUNCTION_DEFINITIONS">
         <xsl:sort select="@type"/>
         <xsl:sort select="@name"/>
    </xsl:apply-templates>

</xsl:template>
```

```
<xsl:template match="state//state" mode="SWITCH_BLOCK">
    <xsl:apply-templates
        select="ancestor::stateMachine/transition
            [current()/ancestor-or-self:state/@name = @sourceRef]"
        mode="SWITCH_BLOCK">
          <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>
</xsl:template>
```

В первом шаблоне используются ключи и свойство функции `generate-id`, которая генерирует ключ по первому из узлов множества. Во втором шаблоне производится поиск элементов **`transition`**, имеющих атрибут **`sourceRef`**, равный одному из родителей текущего состояния.

Целиком код преобразования приведен в приложении 2.

Другим, более изощренным примером является шаблон для генерации кода на языке *Turbo Assembler*. Получаемый код должен содержать автоматные процедуры для каждого автомата и обеспечивать логирование действий автомата. Также необходимо выделить отдельно файл для изменения пользователем.

Прежде всего необходимо проверить, с каким конфигурационным файлом работает данное инструментальное средство. Это необходимо для того, чтобы обрабатывать узлы **actionNode** и **conditionNode** правильных типов (разд. 3.6.5). Будем строить шаблоны в предположении, что используется с конфигурационный файл, приведенный в предыдущем разделе.

Выделим файлы, необходимые для реализации автоматов, и опишем их назначение:

- `log.asm` – файл, обеспечивающий логирование;
- `proc.asm` – файл, содержащий заголовки процедур, которые должен реализовать пользователь;
- `auto.asm` – файл, содержащий автоматные процедуры всех автоматов в проекте.

Рассмотрим каждый файл более подробно.

log.asm

Данный файл содержит процедуры, обеспечивающие логирование входных и выходных переменных, начала и конца работы автомата. Данный файл не содержит никакой специфичной для проекта информации. Поэтому он генерируется не по шаблону, а может быть просто скопирован в каждый новый проект. Не будем приводить здесь полный код данного файла. Отметим только что в нем реализуются методы `logZ`, `logX`, `logA` и `logAend`, обеспечивающие логирование выходной и входной переменной, начала и окончания работы автомата соответственно.

proc.asm

Данный файл содержит реализацию входных и выходных переменных. Файл предназначен для модификации пользователем. Поэтому он должен быть сгенерирован один раз и перезаписываться только по желанию пользователя. Файл может быть разделен на следующие логические части:

- объявление публичных процедур;
- объявление внешних процедур, обеспечивающих логирование;
- заготовки процедур.

auto.asm

Данный файл содержит автоматные процедуры всех автоматов в проекте. Файл целиком генерируется автоматически и не должен модифицироваться вручную.

Файл может быть разделен на следующие логические части:

- объявление публичных процедур;
- объявление внешних процедур (процедур, реализованных в файле `proc.asm` и процедур, обеспечивающих логирование);
- реализация автоматных процедур.

Каждая автоматная процедура состоит из `switch`-блока, обеспечивающего логику работы автомата.

Для того, чтобы собрать наиболее часто употребляющиеся функции извлечения различных параметров из XML-документа, был написан XSLT-шаблон, приведенный в листинге 4.

Листинг 4. Наиболее часто употребляющиеся функции извлечения параметров из XML-файла

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

<!-- GETTING AUTOMATA NAME FROM ANY CHILD -->
<xsl:template name="automataName">
<xsl:value-of select="ancestor-or-self::stateMachine/@name" />
</xsl:template>
<!-- MISSED AUTOMATAS -->
<xsl:variable name="missedAutomatas"
    select="//actionNode[@type='AUTOMATA_CALL']
[not(parameter[@name='automata']/@value=ancestor::model/stateMachine/@name)]
[not(parameter[@name='automata']/@value=preceding::actionNode[@type='AUTOMATA_CALL']
/parameter[@name='automata'] /@value)]"/>

<!-- APPLY TEMPLATES FOR EACH ACTION NODE (SELECTED DISTINCTLY) -->
<xsl:template name="distinctActionNodes">
    <xsl:param name="mode"/>
    <xsl:call-template name="distinctNode">
        <xsl:with-param name="mode" select="$mode"/>
        <xsl:with-param name="nodeName" select="'actionNode'"/>
```

```
    </xsl:call-template>
</xsl:template>

<!-- APPLY TEMPLATES FOR EACH CONDITION NODE (SELECTED DISTINCTLY) -->
<xsl:template name="distinctConditionNodes">
    <xsl:param name="mode"/>
    <xsl:call-template name="distinctNode">
        <xsl:with-param name="mode" select="$mode"/>
        <xsl:with-param name="nodeName" select="'conditionNode'"/>
    </xsl:call-template>
</xsl:template>


<!-- TRANSITIONS -->
<xsl:template name="transitions">
    <xsl:param name="mode"/>
    <xsl:param name="parameter1" select="''"/>
    <xsl:param name="parameter2" select="''"/>
    <xsl:for-each
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:call-template name="callWithMode">
            <xsl:with-param name="mode" select="$mode"/>
            <xsl:with-param name="parameter1" select="$parameter1" />
            <xsl:with-param name="parameter2" select="$parameter2" />
        </xsl:call-template>

    </xsl:for-each>
</xsl:template>


<xsl:template name="distinctNode">
    <xsl:param name="mode"/>
    <xsl:param name="nodeName"/>
    <xsl:param name="parameter1" select="''"/>
    <xsl:param name="parameter2" select="''"/>
    <xsl:for-each select="//stateMachine//*[name()=$nodeName]">
        <xsl:sort select="ancestor::stateMachine/@name"/>
        <xsl:sort select="@type"/>
        <xsl:sort select="@name"/>

        <xsl:if
test="count(preceding::*[name()=$nodeName][@type=current()/@type][@name =
current()/@name][ancestor::stateMachine/@name =
current()/ancestor::stateMachine/@name]) = 0">
            <xsl:call-template name="callWithMode">
                <xsl:with-param name="mode" select="$mode"/>
                <xsl:with-param name="parameter1" select="$parameter1" />
                <xsl:with-param name="parameter2" select="$parameter2" />
            </xsl:call-template>
        </xsl:if>
    </xsl:for-each>
</xsl:template>

<xsl:template name="callWithMode">
    <xsl:param name="mode"/>
    <xsl:param name="parameter1" select="''"/>
    <xsl:param name="parameter2" select="''"/>

    <xsl:choose>
        <xsl:when test="$mode='SWITCH_BLOCK'">
            <xsl:apply-templates select="." mode="SWITCH_BLOCK">
```

```
                <xsl:with-param name="parameter1" select="$parameter1"/>
                <xsl:with-param name="parameter2" select="$parameter2"/>
            </xsl:apply-templates>
        </xsl:when>
        <xsl:when test="$mode='FUNCTION_DECLARATION'">
            <xsl:apply-templates select="." mode="FUNCTION_DECLARATION">
                <xsl:with-param name="parameter1" select="$parameter1"/>
                <xsl:with-param name="parameter2" select="$parameter2"/>
            </xsl:apply-templates>
        </xsl:when>
        <xsl:when test="$mode='FUNCTION_DEFINITION'">
            <xsl:apply-templates select="." mode="FUNCTION_DEFINITION">
                <xsl:with-param name="parameter1" select="$parameter1"/>
                <xsl:with-param name="parameter2" select="$parameter2"/>
            </xsl:apply-templates>
        </xsl:when>
        <xsl:when test="$mode='STRING'">
            <xsl:apply-templates select="." mode="STRING">
                <xsl:with-param name="parameter1" select="$parameter1"/>
                <xsl:with-param name="parameter2" select="$parameter2"/>
            </xsl:apply-templates>
        </xsl:when>
        <xsl:otherwise>
            <xsl:apply-templates select=".">
                <xsl:with-param name="parameter1" select="$parameter1"/>
                <xsl:with-param name="parameter2" select="$parameter2"/>
            </xsl:apply-templates>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Этот шаблон позволяет извлекать из XML-документа

- список вызываемых, но не реализованных автоматов;
- список используемых автоматом действий и условий с различными значениями параметра mode;
- список переходов из состояния.

Многие из указанных списков реализуются как шаблоны, вызываемые по имени. В качестве параметров в эти шаблоны можно передавать mode, parameter1 и parameter2. Данные шаблоны будут обрабатывать все указанные узлы в режиме, указанном в переменной mode, передавая параметры parameter1 и parameter2. Например, если необходимо получить список всех используемых автоматом действий, необходимо вызвать шаблон distinctActionNodes и реализовать шаблоны для узлов actionNode.

Например, в шаблоне для файла proc.asm присутствует следующие строки:

```
<xsl:call-template name="distinctActionNodes">
    <xsl:with-param name="mode">FUNCTION_DEFINITION</xsl:with-param>
</xsl:call-template>
      …

<xsl:template match="actionNode[@type='OUTPUT_ACTION']"
                                         mode="FUNCTION_DEFINITION">
;place your comments here
<xsl:value-of select="ancestor::stateMachine/@name"/>_z<xsl:value-of
      select="@name"/> proc near
          ; place your code here
      ret 0
    <xsl:value-of select="ancestor::stateMachine/@name"/>_z<xsl:value-of
      select="@name"/> endp
</xsl:template>
```

Вызванный шаблон `distinctActionNodes` вызывает, в свою очередь, шаблоны для всех узлов `actionNode`, использованных в автомате, в контексте которого вызван шаблон. Шаблоны для узлов `actionNode` различаются по типу. В приведенном выше коде обрабатываются только узлы типа `OUTPUT_ACTION`.

Исполнение этих строк для автомата с двумя выходными переменными порождает следующий исходный код:

```
A0_z1X proc near
        ;place your code here
    ret 0
A0_z1X endp
A0_z2X proc near
        ;place your code here
    ret 0
A0_z2X endp
```

Приведем полный код преобразования, генерирующего файл `proc.asm` (листинг 5). Учитывая тот факт, что этот файл должен содержать как описания публичных процедур, так и сами процедуры, необходимо вызвать дважды шаблоны `distinctActionNodes` и `distinctConditionNodes` с разными значениями параметра `mode`. Код преобразования в связи с этим сводится к четырем вызовам (по два на каждый из шаблонов) и шаблонам обработки узлов всех типов для указанных двух случаев.

Листинг 5. XSLT-шаблон, генерирующее файл proc.asm

```
<?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:include href="MA_common.xslt"/>

<xsl:template match="/model">
;--- this file is machine generated ---
;Model <xsl:value-of select="@name"/>

INCLUDE Mymacros.inc
.model small
.586
    <xsl:call-template name="distinctActionNodes">
       <xsl:with-param name="mode">FUNCTION_DECLARATION</xsl:with-param>
    </xsl:call-template>

    <xsl:call-template name="distinctConditionNodes">
       <xsl:with-param name="mode">FUNCTION_DECLARATION</xsl:with-param>
    </xsl:call-template>
.data
    MACRO_DATA
.code
    <xsl:call-template name="distinctActionNodes">
       <xsl:with-param name="mode">FUNCTION_DEFINITION</xsl:with-param>
    </xsl:call-template>

    <xsl:call-template name="distinctConditionNodes">
       <xsl:with-param name="mode">FUNCTION_DEFINITION</xsl:with-param>
    </xsl:call-template>
end
</xsl:template>


<xsl:template match="actionNode[@type='OUTPUT_ACTION']" mode="FUNCTION_DECLARATION">
    public <xsl:value-of
         select="ancestor::stateMachine/@name"/>_z<xsl:value-of select="@name"/>
</xsl:template>

<xsl:template match="actionNode[@type='AUTOMATA_CALL']" mode="FUNCTION_DECLARATION"
></xsl:template>

<xsl:template match="actionNode" priority="0" mode="FUNCTION_DECLARATION" >
    ;ERROR: Unknown actionNode type: '<xsl:value-of select="@type"/>'
    ; Please, correct xslt file;
</xsl:template>

<xsl:template match="conditionNode[@type='AUTOMATA_CALL']"
                                                     mode="FUNCTION_DECLARATION">
</xsl:template>

<xsl:template match="conditionNode[@type='INPUT_VARIABLE']"
                                                    mode="FUNCTION_DECLARATION" >
    public <xsl:value-of select="ancestor::stateMachine/@name"/>_x<xsl:value-of
select="@name"/>
</xsl:template>

<xsl:template match="conditionNode[@type='EVENT']" mode="FUNCTION_DECLARATION"
></xsl:template>

<xsl:template match="conditionNode" priority="0" mode="FUNCTION_DECLARATION" >
    ;ERROR: Unknown conditionNode type: '<xsl:value-of select="@type"/>'
    ; Please, correct xslt file;
</xsl:template>


<xsl:template match="actionNode[@type='OUTPUT_ACTION']"
```

```
                                                      mode="FUNCTION_DEFINITION">
    ;
    <xsl:value-of select="ancestor::stateMachine/@name"/>_z<xsl:value-of
select="@name"/> proc near
       ;place your code here
       ret 0
    <xsl:value-of select="ancestor::stateMachine/@name"/>_z<xsl:value-of
select="@name"/> endp
</xsl:template>

<xsl:template match="actionNode[@type='AUTOMATA_CALL']" mode="FUNCTION_DEFINITION"
></xsl:template>

<xsl:template match="actionNode" priority="0" mode="FUNCTION_DEFINITION" >
    ;ERROR: Unknown actionNode type: '<xsl:value-of select="@type"/>'
    ; Please, correct xslt file;
</xsl:template>

<xsl:template match="conditionNode[@type='AUTOMATA_CALL']"
mode="FUNCTION_DEFINITION"></xsl:template>

<xsl:template match="conditionNode[@type='INPUT_VARIABLE']" mode="FUNCTION_DEFINITION" >
    ;
    <xsl:value-of select="ancestor::stateMachine/@name"/>_x<xsl:value-of
select="@name"/> proc near
       ;place your code here
       mov ax, 0
       ret 0
    <xsl:value-of select="ancestor::stateMachine/@name"/>_x<xsl:value-of
select="@name"/> endp
</xsl:template>

<xsl:template match="conditionNode[@type='EVENT']" mode="FUNCTION_DEFINITION"
></xsl:template>

<xsl:template match="conditionNode" priority="0" mode="FUNCTION_DEFINITION" >
    ;ERROR: Unknown conditionNode type: '<xsl:value-of select="@type"/>'
    ; Please, correct xslt file;
</xsl:template>

</xsl:stylesheet>
```

Преобразование, генерирующее файл `auto.asm`, использует те же функции. Единственной сложностью при написании этого преобразования была реализация логических выражений, связанная с особенностью кода на языке *Turbo Assembler*.

Логические выражения реализуются следующим образом:

- логическому выражению сопоставляются две точки выхода – для случая, когда выражение истинно и для случая, когда оно ложно;
- вызывается шаблон, обрабатывающий узел дерева разбора логического выражения. В качестве параметров передаются названия точек выхода;

- если обрабатываемый узел дерева – унарная или бинарная операция, происходит рекурсивная обработка дочерних элементов узла, но с другими значениями точек выхода. Например, если текущие точки выхода называются «OK» и «FAIL», то происходит обработка узла, соответствующего операции ИЛИ. При этом первый узел вызовется с параметрами «OK» и «FAIL_or», а второй – с параметрами «OK» и «FAIL». Точка выхода FAIL_or указывает на метку, соответствующий коду, полученному после обработки второго узла;
- если обрабатываемый узел – лист, то проверяется условие, а затем происходит переход в одну из точек выхода в зависимости от результата проверки.

Отметим, что для языков высокого уровня такая сложная обработка логических выражений не требуется. Для построения логического выражения в таких языках используется простая рекурсия, основанная на шаблонах и описанная во многих пособиях по языку преобразований `XSLT`.

Текст преобразования для генерации файла `auto.asm` содержится в приложении 3.

Приложение 4 содержит шаблон преобразования, генерирующий исходный код на языке *C*, идентичный коду, создаваемому конвертером *Visio2Switch*.

4.4. Создание конфигурационного файла

В разд. 3.6.5 был описан файл конфигурации, использующийся разработанным инструментальным средством. В приложении 1 приведен стандартный файл конфигурации. Рассмотрим пример применения этого файла. Например, создадим конфигурационный файл, поддерживающий названия входных переменных в стиле инструментального средства *UniMod*.

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

```
<название объекта управления>.<название входного/выходного события>
```

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

Такой способ наименования входных переменных не был предусмотрен при реализации конвертора *Visio2Switch*. Однако в рамках объектно-ориентированного программирования с явным выделением состояний он может получить широкое распространение.

Зададим формат таких входных переменных в конфигурационном файле. Положим, что, как и в языке *Java* (использующемся в *UniMod*), название объекта управления, как и название самой входной переменной, состоит из букв, цифр, знака подчеркивания и некоторых других допустимых символов. Тогда синтаксис такой входной переменной можно описать с помощью регулярных выражений следующим образом:

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

Здесь `\w` означает допустимый в имени 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 (например, строки `o1.x10`) данному регулярному выражению, при положительном исходе проверки (если соответствие было обнаружено) можно извлечь название объекта

управления (именованная область controlledObjectName) и название входной переменной (именованная область inputVariableName).

Узел, представленный в листинге 6, задает синтаксис входной переменной в стиле инструментального средства *UniMod*.

Листинг 6. Узел конфигурационного файла, задающий синтаксис входной переменной в стиле инструментального средства *UniMod*

```
<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>
```

Применив данный шаблон к строке o1.x10, получим объект класса NodeElement со следующими значениями свойств:

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

Данный объект будет отображен в XML-документе, описывающем граф переходов автомата следующим образом:

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

Заключение

Разработано инструментальное средство, позволяющее автоматизировать генерацию исходных кодов программ по графам переходов автоматов. Данное инструментальное средство имеет большое практическое значение, так как позволяет генерировать код на различных языках программирования и настраивать его для конкретных проектов. Это позволяет эффективно внедрять SWITCH-технологию в самые разные программные решения.

В качестве метода генерации исходного кода был выбран способ генерации, основанный на обработчиках данных регулярной структуры – `XSLT`-преобразованиях. Этот метод генерации кода учитывает специфику графов переходов – наличие групп состояний, вложенных групп состояний, наличие логических выражений.

В процессе работы над данным инструментальным средством, проверялась также его работоспособность. В частности, лексические и синтаксические анализаторы, использующиеся данным инструментальным средством, которые основаны на автоматном программировании, были получены с помощью прототипа данного инструментального средства. Также были разработаны преобразования для генерации исходных кодов на языках *C*, *C#* и *Turbo Assembler*. Разработанное инструментальное средство будет улучшаться в процессе его эксплуатации.

Список литературы

1. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1.
2. *Шалыто А.А, Туккель Н.И.* SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем //Программирование . 2001. № 5. http://is.ifmo.ru/works/switch/1/.
3. *Шалыто А.А., Туккель Н.И.* Реализация автоматов при программировании событийных систем //Программист. 2002. № 4. http://is.ifmo.ru/works/evsys/.
4. *Шалыто А.А.* Новая инициатива в программировании. Движение за открытую проектную документацию //Мир ПК. 2003. № 9. http://is.ifmo.ru/works/open_doc/.
5. *Шалыто А.А., Наумов Л.А.* Реализация автоматов в объектно-ориентированных программах //Искусственный интеллект. 2004. № 4. http://is.ifmo.ru/works/_aut_oop.pdf.
6. *Finite state machine. Wikipedia.* The free encyclopedia. http://en.wikipedia.org/wiki/Finite_automaton.
7. *Головешин А.* Использование конвертора *Visio2SWITCH*. http://is.ifmo.ru/progeny/visio2switch/.
8. *Гуров В.С., Мазин М.А., Шалыто А.А. UniMod* – программный пакет для разработки объектно-ориентированных приложений на основе автоматного подхода //Труды XI Всероссийской научно-методической конференции «Телематика-2004». 2004. Т.1. http://tm.ifmo.ru.
9. *Гуров В.С., Мазин М.А.* Сайт проекта *UniMod*. Раздел *Methodology*. http://UniMod.sourceforge.net.
10. *Сайт* проекта *Finite State Machine*. http://fsme.sourceforge.net.
11. *Фаулер М., Скотт К.* UML. Основы. СПб.: Символ-Плюс, 2002.

12. Материалы встречи Java User Group, посвященной J2ME от 26 февраля 2005 года. http://jug.ru.
13. *Селлз К.* Современные способы автоматизации повторяющихся задач программирования //MSDN Magazine, 2002, № 6.
14. *Dodds L.* Code generation using XSLT, ibm.com/developerWorks. http://www-106.ibm.com/developerworks/edu/x-dw-codexslt-i.html.
15. Официальный сайт инструментального средства CodeSmith. http://www.codesmithtools.com/.
16. *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.
17. *Herrington J.* Extensible Code Generation with Java. http://today.java.net/pub/a/today/2004/05/12/generation1.html.
18. *Herrington J*. Creating Portable Applications with Code Generation.
19. *Заякин Е., Шалыто А.* Метод устранения повторных фрагментов кода при реализации конечных автоматов. http://is.ifmo.ru/projects/life_app/.
20. *Microsoft Office Visio* ShapeSheet Reference, MSDN. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vissdk11/html/viconWhatsNewSS2003_HV01046253.asp.
21. *Штучкин А., Шалыто А.* Совместное использование теории построения компиляторов и SWITCH-технологии (на примере построения калькулятора). http://is.ifmo.ru/projects/calc/.
22. *Ковалев А.С., Лукьянова А.П., Шалыто А.А.* Новый метод вычисления булевых формул. http://is.ifmo.ru/projects/libgmx/.
23. Overview: *NMAKE* Reference. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcug98/html/_asug_overview.3a_.nmake_reference.asp

Приложение 1. Стандартный конфигурационный файл

```
<?xml version='1.0' encoding='Windows-1251' ?>
<configuration>
  <parsers>
    <nodes>
      <!-- Examples: e1, e2_23, e12_get -->
      <nodeTemplate template='EVENT' regexp='e(?&lt;name&gt;\w+)'
                    format='e#name#' name='${name}'>
        <parameter name='name' value='${name}' />
      </nodeTemplate>
      <!-- Examples: x1, x2_23, x12_equal -->
      <nodeTemplate template='INPUT_VARIABLE' regexp='x(?&lt;name&gt;\w+)'
                    format='x#name#' name='${name}'>
        <parameter name='name' value='${name}' />
      </nodeTemplate>
      <!-- Examples: y12 = 10, yB2 == BB_3, y12_3 = B_12  -->
      <nodeTemplate template='OTHER_AUTOMATA_EVENT_EQUAL'
              regexp='y(?&lt;automata&gt;\w+)(\ *)(=|==)(\ *)(?&lt;event&gt;\w+)'
              name='y${automata}e${event}' format='y#automata#==#event#'>
        <parameter name='automata' value='${automata}' />
        <parameter name='event' value='${event}' />
      </nodeTemplate>
      <!-- Examples: y12 != 10, yB2 <> BB_3, y12_3 != B_12  -->
      <nodeTemplate template='OTHER_AUTOMATA_EVENT_NOTEQUAL'
        regexp='y(?&lt;automata&gt;\w+)(\ *)(&lt;&gt;|!=)(\ *)(?&lt;event&gt;\w+)'
        name='y${automata}e${event}' format='y#automata#!=#event#'>
        <parameter name='automata' value='${automata}' />
        <parameter name='event' value='${event}' />
      </nodeTemplate>
      <!-- Examples: z1, z2_23, z12_doIt -->
      <nodeTemplate template='OUTPUT_ACTION' regexp='z(?&lt;name&gt;\w+)'
format='z#name#' name='${name}'>
        <parameter name='name' value='${name}' />
      </nodeTemplate>
      <!-- Examples: 1, 2_23, 12_doIt -->
      <nodeTemplate template='SIMPLE_OUTPUT_ACTION_IN_STATE'
       regexp='(?&lt;name&gt;([A-Za-y0-9_]\w*))' format='#name#' name='${name}'>
        <parameter name='name' value='${name}' />
      </nodeTemplate>
      <!-- Examples: A12(e12), Ac_3(B2)  -->
      <nodeTemplate template='AUTOMATA_CALL'
                    regexp='A(?&lt;automata&gt;\w+)\(e{0,1}(?&lt;event&gt;\w+)\)'
                    format='A#automata#(e#event#)' name='A${automata}e${event}'>
        <parameter name='automata' value='${automata}' />
        <parameter name='event' value='${event}' />
      </nodeTemplate>
      <!-- Examples: A12, Ac_3  -->
      <nodeTemplate template='SIMPLE_AUTOMATA_CALL_IN_STATE'
              regexp='(?&lt;automata&gt;([B-Za-z0-9_]\w*))' format='${automata}'
              name='A${automata}e0'>
        <parameter name='automata' value='${automata}' />
        <parameter name='event' value='0' />
      </nodeTemplate>
    </nodes>
    <descriptions>
      <node template='EVENT' />
```

```
      <node template='INPUT_VARIABLE' />
      <node template='OUTPUT_ACTION' />
    </descriptions>
    <state>
      <stateMachineRef>
        <node template='SIMPLE_AUTOMATA_CALL_IN_STATE' />
        <node template='AUTOMATA_CALL' />
        <delimeter regexp=';' format=';' />
        <delimeter regexp=',' format=',' />
        <space regexp='^A:|\s' />
      </stateMachineRef>
      <outputAction>
        <node template='OUTPUT_ACTION' />
        <node template='SIMPLE_OUTPUT_ACTION_IN_STATE' />
        <delimeter regexp=';|,' format=';' />
        <space regexp='^z:|\s' />
      </outputAction>
    </state>
    <transition>
      <condition>
        <node template='EVENT' />
        <node template='OTHER_AUTOMATA_EVENT_EQUAL' />
        <node template='OTHER_AUTOMATA_EVENT_NOTEQUAL' />
        <node template='INPUT_VARIABLE' />
        <binaryOperation regexp='&amp;{1,2}' format='&amp;&amp;' type='AND' />
        <binaryOperation regexp='\|{1,2}' format='||' type='OR' />
        <space regexp='\s|^(\s)*\*(\s)*$|^(\s)*1(\s)*$|^(\s)*ok(\s)*$' />
        <unaryOperation regexp='!' format='!' type='NOT' />
        <bracketStart regexp='\[|\(' format='(' />
        <bracketEnd regexp='\]|\)' format=')' />
      </condition>
      <outputAction>
        <node template='OUTPUT_ACTION' />
        <node template='AUTOMATA_CALL' />
        <space regexp='\s|^(\s)*-(\s)*$' />
        <delimeter regexp=';|,' format=';' />
      </outputAction>
    </transition>
  </parsers>
</configuration>
```

Приложение 2. XSLT-шаблон, генерирующий исходный код на языке *C#*

```
<?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:key name="distinctInputVariables"
match="//conditionNode[@type='INPUT_VARIABLE']"
use="parameter[@name='name']/@value" />
   <xsl:key name="distinctOutputActions"
match="//actionNode[@type='OUTPUT_ACTION']" use="parameter[@name='name']/@value"
/>
   <xsl:key name="distinctAutomataCalls"
match="//actionNode[@type='AUTOMATA_CALL']"
use="parameter[@name='automata']/@value" />
   <xsl:template match="/model">
```

```
//--- this file is machine generated ---
//Model: <xsl:value-of select="@name" />

namespace Automatas
{
    public class BaseAutomata
    {
    }

<xsl:apply-templates select="stateMachine"></xsl:apply-templates>
}

</xsl:template>
   <!--===========================================================-->
   <!--Automata processing-->
   <xsl:template match="stateMachine">
    /// &lt;summary&gt;
    /// <xsl:value-of select="@description" />
    /// &lt;/summary&gt;
    public abstract class <xsl:value-of select="@name" /> : BaseAutomata
    {
        protected string y = "0";

        public void A(int e)
        {
            switch (y)
            {
                <xsl:apply-templates select="state//state[count(state) = 0]"
mode="SWITCH_BLOCK">
         <xsl:sort select="@name" data-type="text" />
      </xsl:apply-templates>
            }
        }
       <xsl:variable name="stateMachineName" select="@name" />
      <xsl:apply-templates select="//actionNode[generate-id(.) = generate-
id(key('distinctOutputActions',
parameter[@name='name']/@value)[ancestor::stateMachine/@name=$stateMachineName])
]" mode="FUNCTION_DEFINITIONS">
         <xsl:sort select="@type" />
         <xsl:sort select="@name" />
      </xsl:apply-templates>
      <xsl:apply-templates select="//actionNode[generate-id(.) = generate-
id(key('distinctAutomataCalls',
parameter[@name='automata']/@value)[ancestor::stateMachine/@name=$stateMachineNa
me])]" mode="FUNCTION_DEFINITIONS">
         <xsl:sort select="@type" />
         <xsl:sort select="@name" />
      </xsl:apply-templates>
      <xsl:apply-templates select="//conditionNode[generate-id(.) = generate-
id(key('distinctInputVariables',
parameter[@name='name']/@value)[ancestor::stateMachine/@name=$stateMachineName])
]" mode="FUNCTION_DEFINITIONS">
         <xsl:sort select="@type" />
         <xsl:sort select="@name" />
      </xsl:apply-templates>
    }
</xsl:template>
   <!--End Automata processing-->
   <!--==================================================================-->
   <xsl:template match="state//state" mode="SWITCH_BLOCK">
                case "<xsl:value-of select="@name" />":
```

```
            <xsl:apply-templates
select="ancestor::stateMachine/transition[current()/ancestor-or-self:_u115
?tate/@name = @sourceRef]" mode="SWITCH_BLOCK">
         <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:template>
   <!--===================================================================-->
   <!-- Transitions processing for creatingΓswitch block -->
   <xsl:template match="transition[not(count(./condition) = 0)]"
mode="SWITCH_BLOCK">
      <xsl:choose>
         <xsl:when test="position() = 1">
                    if      (<xsl:apply-templates select="condition"
mode="SWITCH_BLOCK" />) &#x9;&#x9; {<xsl:apply-templates select="outputAction"
mode="SWITCH_BLOCK" /> y = "<xsl:value-of select="@targetRef" />";}
        </xsl:when>
         <xsl:otherwise>
                    else if (<xsl:apply-templates select="condition"
mode="SWITCH_BLOCK" />) &#x9;&#x9; {<xsl:apply-templates select="outputAction"
mode="SWITCH_BLOCK" /> y = "<xsl:value-of select="@targetRef" />";}
        </xsl:otherwise>
      </xsl:choose>
   </xsl:template>
   <xsl:template match="transition[count(./condition) = 0]" mode="SWITCH_BLOCK">
      <xsl:choose>
         <xsl:when test="position() = 1">
                    if      (true) &#x9;&#x9; {<xsl:apply-templates
select="outputAction" mode="SWITCH_BLOCK" /> y = "<xsl:value-of
select="@targetRef" />";}
        </xsl:when>
         <xsl:otherwise>
                    else if (true) &#x9;&#x9; {<xsl:apply-templates
select="outputAction" mode="SWITCH_BLOCK" /> y = "<xsl:value-of
select="@targetRef" />";}
        </xsl:otherwise>
      </xsl:choose>
   </xsl:template>
   <!-- End Transitions processing for creating switch block -->
   <!--=======================================================================-->
   <xsl:template match="condition" mode="SWITCH_BLOCK">
      <xsl:apply-templates mode="SWITCH_BLOCK" />
   </xsl:template>
   <xsl:template match="outputAction" mode="SWITCH_BLOCK">
      <xsl:apply-templates mode="SWITCH_BLOCK" />
   </xsl:template>
   <!--===================================================================-->
   <!-- Action nodes -->
   <xsl:template match="actionNode[@type='OUTPUT_ACTION']"
mode="SWITCH_BLOCK">z<xsl:value-of select="@name" />(); </xsl:template>
   <xsl:template match="actionNode[@type='AUTOMATA_CALL']"
mode="SWITCH_BLOCK">Call_<xsl:value-of
select="parameter[@name='automata']/@value" />(<xsl:value-of
select="parameter[@name='event']/@value" />); </xsl:template>
   <xsl:template match="actionNode" priority="0" mode="SWITCH_BLOCK">/* ERROR:
Unknown action Node type: '<xsl:value-of select="@type" />';*/</xsl:template>
```

```
   <!-- End Action nodes -->
   <!--==========================================================================-->
   <!--======================================================================-->
   <!-- Condition nodes and operations -->
   <xsl:template match="conditionNode[@type='INPUT_VARIABLE']"
mode="SWITCH_BLOCK">x<xsl:value-of select="@name" />()</xsl:template>
   <xsl:template match="conditionNode[@type='EVENT']" mode="SWITCH_BLOCK">e ==
<xsl:value-of select="@name" /></xsl:template>
   <xsl:template match="conditionNode" priority="0" mode="SWITCH_BLOCK">/*
ERROR: Unknown action Node type: '<xsl:value-of select="@type" />'; Please,
correct the xslt file; */</xsl:template>
   <xsl:template match="binaryOperation[@type='AND']"
mode="SWITCH_BLOCK">(<xsl:apply-templates select="child::*[position()=1]"
mode="SWITCH_BLOCK" />) &amp;&amp; (<xsl:apply-templates
select="child::*[position()=2]" mode="SWITCH_BLOCK" />)</xsl:template>
   <xsl:template match="binaryOperation[@type='OR']"
mode="SWITCH_BLOCK">(<xsl:apply-templates select="child::*[position()=1]"
mode="SWITCH_BLOCK" />) || (<xsl:apply-templates select="child::*[position()=2]"
mode="SWITCH_BLOCK" />)</xsl:template>
   <xsl:template match="unaryOperation[@type='NOT']"
mode="SWITCH_BLOCK">!(<xsl:apply-templates select="child::*[position()=1]"
mode="SWITCH_BLOCK" />)</xsl:template>
   <xsl:template match="binaryOperation" priority="0"
mode="SWITCH_BLOCK">/*Error: Unknown binary operation type: '<xsl:value-of
select="@type" />'; Please, correct the xslt file; */</xsl:template>
   <xsl:template match="unaryOperation" priority="0"
mode="SWITCH_BLOCK">/*Error: Unknown unary operation type:_u39 ?<xsl:value-of
select="@type" />'; Please, correct the xslt file; */</xsl:template>
   <!-- End Condition nodes and operations -->
   <!--==========================================================================-->
   <xsl:template match="actionNode[@type='OUTPUT_ACTION']"
mode="FUNCTION_DEFINITIONS">
      /// &lt;summary&gt;
      /// <xsl:value-of select="ancestor-or-self::stateMachine//node[(@type =
current()/@type) and (@name = current()/@name)]/@description" />
      /// &lt;/summary&gt;
      protected abstract void z<xsl:value-of select="@name" />();
</xsl:template>
   <xsl:template match="actionNode[@type='AUTOMATA_CALL']"
mode="FUNCTION_DEFINITIONS">
      /// &lt;summary&gt;
      /// Вызов реализации автомата <xsl:value-of
select="parameter[@name='automata']/@value" />.
      /// &lt;/summary&gt;
      protected abstract void Call_<xsl:value-of
select="parameter[@name='automata']/@value" />(int e);
</xsl:template>
   <xsl:template match="actionNode" priority="0" mode="FUNCTION_DEFINITIONS">
      /*ERROR: Unknown actionNode type: '<xsl:value-of select="@type" />';
Please, correct xslt file;*/
</xsl:template>
   <xsl:template match="conditionNode[@type='INPUT_VARIABLE']"
mode="FUNCTION_DEFINITIONS">
      /// &lt;summary&gt;
      /// <xsl:value-of select="ancestor-or-self::stateMachine//node[(@type =
current()/@type) and (@name = current()/@name)]/@description" />
      /// &lt;/summary&gt;
      /// &lt;returns&gt;True если условие выполнено, false - в противном
случае&lt;/returns&gt;
      protected abstract bool x<xsl:value-of select="@name" />();
</xsl:template>
```

```
   <xsl:template match="conditionNode[@type='EVENT']"
mode="FUNCTION_DEFINITIONS"></xsl:template>
   <xsl:template match="conditionNode" priority="0" mode="FUNCTION_DEFINITIONS">
      /*ERROR: Unknown conditionNode type: '<xsl:value-of select="@type" />';
Please, correct xslt file;*/
</xsl:template>
</xsl:stylesheet>
```

Приложение 3. XSLT-шаблон, генерирующий файл auto.asm

```
<?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:include href="MA_common.xslt" />
  <xsl:template match="/model">

;--- this file is machine generated ---
;Model: <xsl:value-of select="@name" />

INCLUDE mymacros.inc
.model small
.586

;PUBLIC declarations section============================================
    <xsl:for-each select="stateMachine">
        public call<xsl:value-of select="@name" />
    </xsl:for-each>

    <xsl:for-each select="$missedAutomatas">
        public call<xsl:value-of select="parameter[@name='automata']/@value" />
    </xsl:for-each>

;END PUBLIC declarations section========================================


.data
        MACRO_DATA
        ;string to output in case of concurrency error
        strBlock    db  "Concurrency error!!!$"

    <xsl:for-each select="stateMachine">
        y<xsl:value-of select="@name" /> dw 0         ;automata state
        block<xsl:value-of select="@name" /> dw 0    ;automata block variable
(for concurrency error check)
        str<xsl:value-of select="@name" /> db "<xsl:value-of select="@name" />",
"$" ;name of the automata to pass to log function
    </xsl:for-each>

    <xsl:for-each select="$missedAutomatas">
        str<xsl:value-of select="parameter[@name='automata']/@value" /> db
"<xsl:value-of select="parameter[@name='automata']/@value" />", "$"
    </xsl:for-each>

    <xsl:call-template name="distinctActionNodes">
     <xsl:with-param name="mode" select="'STRING'" />
   </xsl:call-template>
```

```
    <xsl:call-template name="distinctConditionNodes">
     <xsl:with-param name="mode" select="'STRING'" />
   </xsl:call-template>
;END Strings to output=================================================


.code

;======================================================================
;Extern functions declaration
;======================================================================
    <xsl:call-template name="distinctActionNodes">
     <xsl:with-param name="mode" select="'FUNCTION_DEFINITION'" />
   </xsl:call-template>

    <xsl:call-template name="distinctConditionNodes">
     <xsl:with-param name="mode" select="'FUNCTION_DEFINITION'" />
   </xsl:call-template>

        EXTRN   logZ :near
        EXTRN   logX :near
        EXTRN   logA :near
        EXTRN   logAend :near
;======================================================================
;END Extern functions declaration
;======================================================================


<xsl:apply-templates select="stateMachine"></xsl:apply-templates>


;======================================================================
;WARNING!!! Automatas descriptions are_u109 ?issed!!!
;======================================================================
<xsl:for-each select="$missedAutomatas">
      call<xsl:value-of select="parameter[@name='automata']/@value" /> proc near
            mov ax, 0
            ret 0
      call<xsl:value-of select="parameter[@name='automata']/@value" /> endp
</xsl:for-each>
;======================================================================
;WARNING!!! Automatas descriptions areГmissed!!!
;======================================================================

end

</xsl:template>
  <!--=====================================================================-->
  <!--Automata processing-->
  <xsl:template match="stateMachine">
     ; &lt;summary&gt;
     ; <xsl:value-of select="@description" />
     ; &lt;/summary&gt;
     ; &lt;param name="event"&gt;event to process&lt;/param&gt;
    call<xsl:value-of select="@name" /> proc

        PROC_START

        cmp     block<xsl:value-of select="@name" />, 0
        je      PROCESS_<xsl:value-of select="@name" />
```

```
        WRITE_STRING strBlock, ' '
        PROC_END
        mov     ax, 1
        ret     0

    PROCESS_<xsl:value-of select="@name" />:
        mov     block<xsl:value-of select="@name" />, 1
        mov     bx, -1
        xor     ax, ax
        mov     cx, [bp+4]   ;read event
        add     bp, 6

        mov     ax, y<xsl:value-of select="@name" />
        push    ax
        push    cx
        PUSH_STRING str<xsl:value-of select="@name" />
        call    logA
        POP_STRING
        add     sp, 4

        <xsl:apply-templates select="state//state[count(child::state) = 0]"
mode="SWITCH_BLOCK">
       <xsl:sort select="@name" data-type="text" />
    </xsl:apply-templates>

    <!--Previous apply-templates will generate a reference to that point-->
    CASE_<xsl:call-template name="automataName" />_<xsl:value-of
select="count(state//state[count(child::state) = 0])" />:

    END_CASE_BLOCK_<xsl:call-template name="automataName" />:

        mov     ax, y<xsl:value-of select="@name" />
        push    ax
        PUSH_STRING str<xsl:value-of select="@name" />
        call    logAend
        POP_STRING
        add     sp, 2

        mov     block<xsl:value-of select="@name" />, 0

        PROC_END
        mov ax, 0
        ret 0
    call<xsl:value-of select="@name" /> endp
</xsl:template>
  <!--End Automata processing-->
  <!--
================================================================================
=============-->
  <xsl:template match="state//state" mode="SWITCH_BLOCK">
    CASE_<xsl:call-template name="automataName" />_<xsl:value-of
select="position()-1" />:
        cmp y<xsl:value-of select="ancestor::stateMachine/@name" />, <xsl:value-
of select="@name" />
        jne  CASE_<xsl:call-template name="automataName" />_<xsl:value-of
select="position()" />

        ;process transitions
        <xsl:call-template name="transitions">
      <xsl:with-param name="mode" select="'SWITCH_BLOCK'" />
```

```
      <xsl:with-param name="parameter1" select="position()" />
    </xsl:call-template>
     jmp END_CASE_BLOCK_<xsl:call-template name="automataName" />
</xsl:template>
  <!--
================================================================================
============-->
  <!-- Transitions processing for creatingГswitch block -->
  <xsl:template match="transition[not(count(./condition) = 0)]"
mode="SWITCH_BLOCK">
     <xsl:param name="parameter1" />
     <xsl:variable name="OK">OK_<xsl:call-template name="automataName"
/>_<xsl:value-of select="$parameter1" />_<xsl:value-of
select="count(preceding::transition)" /></xsl:variable>
     <xsl:variable name="FAIL">FAIL_<xsl:call-template name="automataName"
/>_<xsl:value-of select="$parameter1" />_<xsl:value-of
select="count(preceding::transition)" /></xsl:variable>


     ;IF (   <xsl:apply-templates select="condition" mode="SWITCH_BLOCK">
      <xsl:with-param name="OK_point" select="$OK" />
      <xsl:with-param name="FAIL_point" select="$FAIL" />
    </xsl:apply-templates>
     ;) THEN BEGIN
         <xsl:value-of select="$OK" />:

             <xsl:apply-templates select="outputAction" mode="SWITCH_BLOCK" />
             mov y<xsl:value-of select="ancestor::stateMachine/@name" />,
<xsl:value-of select="@targetRef" />
             jmp END_CASE_BLOCK_<xsl:call-template name="automataName" />
     ;END
         <xsl:value-of select="$FAIL" />:
</xsl:template>
  <xsl:template match="transition[count(./condition) = 0]" mode="SWITCH_BLOCK">
      ;IF (TRUE) BEGIN
          <xsl:apply-templates select="outputAction" mode="SWITCH_BLOCK" />
          mov y<xsl:value-of select="ancestor::stateMachine/@name" />,
<xsl:value-of select="@targetRef" />
      ;END
</xsl:template>
  <!-- End Transitions processing for creating switch block -->
  <!--==================================================================-->
  <xsl:template match="condition" mode="SWITCH_BLOCK">
    <xsl:param name="OK_point" select="OK_" />
    <xsl:param name="FAIL_point" select="FAIL_" />
    <xsl:apply-templates mode="SWITCH_BLOCK">
      <xsl:with-param name="OK_point">
        <xsl:value-of select="$OK_point" />
      </xsl:with-param>
      <xsl:with-param name="FAIL_point">
        <xsl:value-of select="$FAIL_point" />
      </xsl:with-param>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="outputAction" mode="SWITCH_BLOCK">
    <xsl:apply-templates mode="SWITCH_BLOCK" />
  </xsl:template>
  <!--================================================================-->
  <!-- Action nodes -->
  <xsl:template match="actionNode[@type='OUTPUT_ACTION']" mode="SWITCH_BLOCK">
            ;log output action
```

```
            PUSH_STRING str<xsl:value-of select="ancestor::stateMachine/@name"
/>_z<xsl:value-of select="@name" />
            call    logZ
            POP_STRING
            call <xsl:value-of select="ancestor::stateMachine/@name"
/>_z<xsl:value-of select="@name" />
</xsl:template>
  <xsl:template match="actionNode[@type='AUTOMATA_CALL']" mode="SWITCH_BLOCK">
            mov     ax, <xsl:value-of select="parameter[@name='event']/@value"
/>
            push    ax
            call    call<xsl:value-of
select="parameter[@name='automata']/@value" />
            pop     ax
</xsl:template>
  <xsl:template match="actionNode" priority="0" mode="SWITCH_BLOCK">/* ERROR:
Unknown action Node type: '<xsl:value-of select="@type" />';*/</xsl:template>
  <!-- End Action nodes -->
  <!--==============================================================-->
  <!--==================================================================-->
  <!-- Condition nodes and operations -->
  <xsl:template match="conditionNode[@type='INPUT_VARIABLE']"
mode="SWITCH_BLOCK">
        <xsl:param name="OK_point" select="OK_" />
        <xsl:param name="FAIL_point" select="FAIL_" />
            call <xsl:value-of select="ancestor::stateMachine/@name"
/>_x<xsl:value-of select="@name" />
            mov     cx, ax
            push    cx
            PUSH_STRING str<xsl:value-of select="ancestor::stateMachine/@name"
/>_x<xsl:value-of select="@name" />
            call    logX
            POP_STRING
            add     sp, 2
            cmp     cx, 0
            je      <xsl:value-of select="$OK_point" />
            jne     <xsl:value-of select="$FAIL_point" />
</xsl:template>
  <xsl:template match="conditionNode[@type='EVENT']" mode="SWITCH_BLOCK">
    <xsl:param name="OK_point" select="OK_" />
    <xsl:param name="FAIL_point" select="FAIL_" />
    <xsl:choose>
      <xsl:when test="contains(@name, 'X')">
                cmp     cx, <xsl:value-of select="@name" />
                je      <xsl:value-of select="$OK_point" />
                jne     <xsl:value-of select="$FAIL_point" />
            </xsl:when>
      <xsl:otherwise>
                cmp     cx, <xsl:value-of select="translate(@name, 'X', '0')" />
                jl      <xsl:value-of select="$FAIL_point" />
                cmp     cx, <xsl:value-of select="translate(@name, 'X', '9')" />
                jg      <xsl:value-of select="$FAIL_point" />
                mov     bx, cx
                jmp     <xsl:value-of select="$OK_point" />
            </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
  <xsl:template match="conditionNode" priority="0" mode="SWITCH_BLOCK">; ERROR:
Unknown action Node type: '<xsl:value-of select="@type" />'; Please, correct the
xslt file; </xsl:template>
  <!--AND-->
```

```
  <xsl:template match="operation[@type='AND']" mode="SWITCH_BLOCK">
     <xsl:param name="OK_point" select="OK_" />
     <xsl:param name="FAIL_point" select="FAIL_" />

     <xsl:apply-templates select="child::*[position()=1]" mode="SWITCH_BLOCK">
      <xsl:with-param name="OK_point">and_<xsl:call-template name="automataName"
/>_<xsl:value-of select="$OK_point" /></xsl:with-param>
      <xsl:with-param name="FAIL_point">
        <xsl:value-of select="$FAIL_point" />
      </xsl:with-param>
    </xsl:apply-templates>

     <xsl:text></xsl:text>
          and_<xsl:call-template name="automataName" />_<xsl:value-of
select="$OK_point" />:

     <xsl:apply-templates select="child::*[position()=2]" mode="SWITCH_BLOCK">
      <xsl:with-param name="OK_point">
        <xsl:value-of select="$OK_point" />
      </xsl:with-param>
      <xsl:with-param name="FAIL_point">
        <xsl:value-of select="$FAIL_point" />
      </xsl:with-param>
    </xsl:apply-templates>
</xsl:template>
  <!--OR-->
  <xsl:template match="operation[@type='OR']" mode="SWITCH_BLOCK">
     <xsl:param name="OK_point" select="OK_" />
     <xsl:param name="FAIL_point" select="FAIL_" />

     <xsl:apply-templates select="child::*[position()=1]" mode="SWITCH_BLOCK">
      <xsl:with-param name="OK_point">
        <xsl:value-of select="$OK_point" />
      </xsl:with-param>
      <xsl:with-param name="FAIL_point">or_<xsl:call-template name="automataName"
/>_<xsl:value-of select="$FAIL_point" /></xsl:with-param>
    </xsl:apply-templates>

     <xsl:text></xsl:text>
          or_<xsl:call-template name="automataName" />_<xsl:value-of
select="$FAIL_point" />:

     <xsl:apply-templates select="child::*[position()=2]" mode="SWITCH_BLOCK">
      <xsl:with-param name="OK_point">
        <xsl:value-of select="$OK_point" />
      </xsl:with-param>
      <xsl:with-param name="FAIL_point">
        <xsl:value-of select="$FAIL_point" />
      </xsl:with-param>
    </xsl:apply-templates>
</xsl:template>
  <!--NOT-->
  <xsl:template match="operation[@type='NOT']" mode="SWITCH_BLOCK">
    <xsl:param name="OK_point" select="OK_" />
    <xsl:param name="FAIL_point" select="FAIL_" />
    <xsl:apply-templates select="child::*[position()=1]" mode="SWITCH_BLOCK">
      <xsl:with-param name="OK_point">
        <xsl:value-of select="$FAIL_point" />
      </xsl:with-param>
      <xsl:with-param name="FAIL_point">
        <xsl:value-of select="$OK_point" />
```

```
    </xsl:with-param>
   </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="operation" priority="0" mode="SWITCH_BLOCK">;Error:
Unknown operation type: '<xsl:value-of select="@type" />'; Please, correct the
xslt file; </xsl:template>
  <!-- End Condition nodes and operations -->
  <!--=================================================================-->
  <!--==================================================================-->
  <!--FUNCTION_DEFINITION-->
  <!--==================================================================-->
  <xsl:template match="actionNode[@type='OUTPUT_ACTION']"
mode="FUNCTION_DEFINITION">
        EXTRN  <xsl:value-of select="ancestor::stateMachine/@name"
/>_z<xsl:value-of select="@name" /> :near
</xsl:template>
  <xsl:template match="actionNode[@type='AUTOMATA_CALL']"
mode="FUNCTION_DEFINITION"></xsl:template>
  <xsl:template match="actionNode" priority="0" mode="FUNCTION_DEFINITION">
        ;ERROR: Unknown actionNode type: '<xsl:value-of select="@type" />';
Please, correct xslt file;
</xsl:template>
  <xsl:template match="conditionNode[@type='AUTOMATA_CALL']"
mode="FUNCTION_DEFINITION"></xsl:template>
  <xsl:template match="conditionNode[@type='INPUT_VARIABLE']"
mode="FUNCTION_DEFINITION">
        EXTRN  <xsl:value-of select="ancestor::stateMachine/@name"
/>_x<xsl:value-of select="@name" /> :near
</xsl:template>
  <xsl:template match="conditionNode[@type='EVENT']"
mode="FUNCTION_DEFINITION"></xsl:template>
  <xsl:template match="conditionNode" priority="0" mode="FUNCTION_DEFINITION">
        ;ERROR: Unknown conditionNode type: '<xsl:value-of select="@type" />';
Please, correct xslt file;
</xsl:template>
  <!--==================================================================-->
  <!--FUNCTION_DEFINITION END-->
  <!--==================================================================-->
  <!--==================================================================-->
  <!--STRING-->
  <!--==================================================================-->
  <xsl:template match="actionNode[@type='OUTPUT_ACTION']" mode="STRING">
        str<xsl:value-of select="ancestor::stateMachine/@name" />_z<xsl:value-of
select="@name" /> db "<xsl:value-of select="ancestor::stateMachine/@name"
/>_z<xsl:value-of select="@name" />", '$'
</xsl:template>
  <xsl:template match="actionNode[@type='AUTOMATA_CALL']"
mode="STRING"></xsl:template>
  <xsl:template match="actionNode" priority="0" mode="STRING">
        ;ERROR: Unknown actionNode type: '<xsl:value-of select="@type" />';
Please, correct xslt file;
</xsl:template>
  <xsl:template match="conditionNode[@type='AUTOMATA_CALL']"
mode="STRING"></xsl:template>
  <xsl:template match="conditionNode[@type='INPUT_VARIABLE']" mode="STRING">
        str<xsl:value-of select="ancestor::stateMachine/@name" />_x<xsl:value-of
select="@name" /> db "<xsl:value-of select="ancestor::stateMachine/@name"
/>_x<xsl:value-of select="@name" />", '$'
</xsl:template>
  <xsl:template match="conditionNode[@type='EVENT']"
mode="STRING"></xsl:template>
```

```
  <xsl:template match="conditionNode" priority="0" mode="STRING">
        ;ERROR: Unknown conditionNode type: '<xsl:value-of select="@type" />';
Please, correct xslt file;
</xsl:template>
  <!--==========================================================================-->
  <!--STRING END-->
  <!--==========================================================================-->
</xsl:stylesheet>
```

Приложение 4. XSLT-шаблоны, генерирующие код на языке *C*, идентичный коду, генерируемому конвертером *Visio2Switch*

common.cpp

```
<?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">
//--- this file is machine generated ---

#include "StdAfx.h"
#include "common.h"
#include "log.h"

common_t cm;

<xsl:for-each select="stateMachine">

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

  #ifdef A<xsl:value-of select="@name" />_BEGIN_LOGGING
    log_a_begin(<xsl:value-of select="@name" />, y_old, e);
  #endif

  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:value-of select="description" />_

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

```
        <xsl:apply-templates
select="ancestor::stateMachine/transition[current()/ancestor-or-self:_u115
?tate/@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 automata state!");
      #else
        ;
      #endif
  }

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

  #ifdef A<xsl:value-of select="@name" />_TRANS_LOGGING
    log_a_trans(<xsl:value-of select="@name" />, y_old, cm.y<xsl:value-of
select="@name" />);
  #endif

  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:value-of
select="description" />
            <xsl:apply-templates select="stateMachineRef/actionNode" />
            <xsl:apply-templates select="outputAction/actionNode" />
      break;
    </xsl:for-each>
  }

  A<xsl:value-of select="@name" />_end: ;
  #ifdef A<xsl:value-of select="@name" />_END_LOGGING
    log_a_end(<xsl:value-of select="@name" />, cm.y<xsl:value-of select="@name"
/>, e);
  #endif
}

</xsl:for-each>

<xsl:variable name="relatedAutomatas" select="//*[ ((name()='actionNode') and
((@type='AUTOMATA_CALL') or (@type='SIMPLE_AUTOMATA_CALL_IN_STATE'))) or

      ((name()='conditionNode') and ((@type='OTHER_AUTOMATA_EVENT_NOTEQUAL') or
(@type='OTHER_AUTOMATA_EVENT_EQUAL')))]
      [not(parameter[@name='automata']/@value=ancestor::model/stateMachine/@name
)]
      [not(parameter[@name='automata']/@value=preceding::*[
((name()='actionNode') and ((@type='AUTOMATA_CALL') or
(@type='SIMPLE_AUTOMATA_CALL_IN_STATE'))) or
```

```
            ((name()='conditionNode') and
((@type='OTHER_AUTOMATA_EVENT_NOTEQUAL') or
(@type='OTHER_AUTOMATA_EVENT_EQUAL')))]/parameter[@name='automata']/@value)]" />

<xsl:for-each select="$relatedAutomatas">
  void A<xsl:value-of select="parameter[@name='automata']/@value" />(ubyte e) {}
</xsl:for-each>

</xsl:template>
      <xsl:template match="transition">

  <xsl:choose>
                  <xsl:when test="not(condition)">
        if (1)
      </xsl:when>
                  <xsl:otherwise>
      if ( <xsl:apply-templates select="condition" /> )
      </xsl:otherwise>
            </xsl:choose>
      {
        <xsl:apply-templates select="action/actionNode" />
          cm.y<xsl:value-of select="ancestor::stateMachine/@name" /> =
<xsl:value-of select="@targetRef" />;
      }
  <xsl:if test="not(position()=last())">
        else
  </xsl:if>


</xsl:template>
      <xsl:template match="actionNode[(@type='AUTOMATA_CALL') or
(@type='SIMPLE_AUTOMATA_CALL_IN_STATE')]">
                  //
            A<xsl:value-of select="parameter[@name='automata']/@value" />(0);

</xsl:template>
      <xsl:template match="actionNode[@type='OUTPUT_ACTION']">
            z<xsl:value-of select="@name" />();
</xsl:template>
      <xsl:template match="condition">
            <xsl:apply-templates select="*" />
      </xsl:template>
      <xsl:template match="binaryOperation" priority="0">   ( <xsl:apply-
templates select="child::*[position()=1]" /> ) /*Unknown binary operation*/ (
<xsl:apply-templates select="child::*[position()=2]" /> ) </xsl:template>
      <xsl:template match="unaryOperation" priority="0">    ( /*Unknown unary
operation*/ <xsl:apply-templates select="child::*[position()=1]" /> )
</xsl:template>
      <xsl:template match="binaryOperation[@type='AND']"> ( <xsl:apply-templates
select="child::*[position()=1]" /> ) &amp;&amp; ( <xsl:apply-templates
select="child::*[position()=2]" /> ) </xsl:template>
      <xsl:template match="binaryOperation[@type='OR']">  ( <xsl:apply-templates
select="child::*[position()=1]" /> ) || ( <xsl:apply-templates
select="child::*[position()=2]" /> ) </xsl:template>
      <xsl:template match="unaryOperation[@type='NOT']"> !( <xsl:apply-templates
select="child::*[position()=1]" /> ) </xsl:template>
      <xsl:template
match="conditionNode[@type='OTHER_AUTOMATA_EVENT_EQUAL']">cm.y<xsl:value-of
select="parameter[@name='automata']/@value" /> == <xsl:value-of
select="parameter[@name='event']/@value" />
```

```
</xsl:template>
      <xsl:template
match="conditionNode[@type='OTHER_AUTOMATA_EVENT_NOTEQUAL']">cm.y<xsl:value-of
select="parameter[@name='automata']/@value" /> != <xsl:value-of
select="parameter[@name='event']/@value" />
</xsl:template>
      <xsl:template match="conditionNode[@type='EVENT']">e = <xsl:value-of
select="@name" />
</xsl:template>
      <xsl:template match="conditionNode[@type='INPUT_VARIABLE']">x<xsl:value-of
select="@name" />()</xsl:template>
      <xsl:template match="conditionNode" priority="0">/* Unknown condition node
*/</xsl:template>
</xsl:stylesheet>
```

common.h

```
<?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:key name="distinctEvents" match="//conditionNode[@type='EVENT']"
use="parameter[@name='name']/@value" />
      <xsl:variable name="events" select="//conditionNode[generate-id(.) =
generate-id(key('distinctEvents', parameter[@name='name']/@value))]" />
      <xsl:key name="distinctInputVariables"
match="//conditionNode[@type='INPUT_VARIABLE']"
use="parameter[@name='name']/@value" />
      <xsl:variable name="inputVariables" select="//conditionNode[generate-id(.)
= generate-id(key('distinctInputVariables', parameter[@name='name']/@value))]"
/>
      <xsl:key name="distinctOutputActions"
match="//actionNode[@type='OUTPUT_ACTION']" use="parameter[@name='name']/@value"
/>
      <xsl:variable name="outputVariables" select="//actionNode[generate-id(.) =
generate-id(key('distinctOutputActions', parameter[@name='name']/@value))]" />
      <xsl:template match="/model">

//--- this file is machine generated ---

#ifndef CommonH
#define CommonH

#include "types.h"

typedef struct{
 <xsl:for-each select="stateMachine">
  ubyte y<xsl:value-of select="@name" />; // <xsl:value-of select="@description"
/>
 </xsl:for-each>

<xsl:variable name="relatedAutomatas" select="//*[ ((name()='actionNode') and
((@type='AUTOMATA_CALL') or (@type='SIMPLE_AUTOMATA_CALL_IN_STATE'))) or

       ((name()='conditionNode')Гand ((@type='OTHER_AUTOMATA_EVENT_NOTEQUAL') or
(@type='OTHER_AUTOMATA_EVENT_EQUAL')))]
       [not(parameter[@name='automata']/@value=ancestor::model/stateMachine/@name
)]
       [not(parameter[@name='automata']/@value=preceding::*[
((name()='actionNode') and ((@type='AUTOMATA_CALL') or
(@type='SIMPLE_AUTOMATA_CALL_IN_STATE'))) or
```

```
            ((name()='conditionNode') and
((@type='OTHER_AUTOMATA_EVENT_NOTEQUAL') or
(@type='OTHER_AUTOMATA_EVENT_EQUAL')))]/parameter[@name='automata']/@value)]" />

 <xsl:for-each select="$relatedAutomatas">
  ubyte y<xsl:value-of select="parameter[@name='automata']/@value" />;
 </xsl:for-each>

} common_t;

extern common_t cm;

// Automatas A
 <xsl:for-each select="stateMachine">
  void A<xsl:value-of select="@name" />( ubyte e);
 </xsl:for-each>

 <xsl:for-each select="$relatedAutomatas">
  void A<xsl:value-of select="parameter[@name='automata']/@value" />( ubyte e );
 </xsl:for-each>

// Variables X
<xsl:for-each select="$inputVariables">
    <xsl:sort select="@name" />
    ubyte x<xsl:value-of select="@name" />(void); // <xsl:value-of
select="ancestor::stateMachine/nodeDescriptions/node[@name=current()/@name][@typ
e=current()/@type]/@description" />
</xsl:for-each>


// Actions Z
<xsl:for-each select="$outputVariables">
    <xsl:sort select="@name" />
    void z<xsl:value-of select="@name" />(void); // <xsl:value-of
select="ancestor::stateMachine/nodeDescriptions/node[@name=current()/@name][@typ
e=current()/@type]/@description" />
</xsl:for-each>

// Events E
<xsl:for-each select="$events">
    <xsl:sort select="@name" />
    // e<xsl:value-of select="@name" /> - <xsl:value-of
select="ancestor::stateMachine/nodeDescriptions/node[@name=current()/@name][@typ
e=current()/@type]/@description" />
</xsl:for-each>


#endif

</xsl:template>
      </xsl:stylesheet>
```

log.cpp

```
<?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:key name="distinctEvents" match="//conditionNode[@type='EVENT']"
use="parameter[@name='name']/@value" />
```

```
      <xsl:key name="distinctInputVariables"
match="//conditionNode[@type='INPUT_VARIABLE']"
use="parameter[@name='name']/@value" />
      <xsl:key name="distinctOutputActions"
match="//actionNode[@type='OUTPUT_ACTION']" use="parameter[@name='name']/@value"
/>
      <xsl:template match="/model">
//--- this file is machine generated ---

#include "StdAfx.h"
#include "log_user.h"

#ifdef SWITCH_LOGGING

typedef struct{
  ubyte       dig;
  const char* n;
  const char* n_name;
} int_str2_t;

typedef struct{
  const char* n;
  const char* n_name;
} str2_t;

typedef struct{
  const char* n;
  const char* n_name;
  str2_t* str2;
} str3_t;

<xsl:variable name="events" select="//conditionNode[generate-id(.) = generate-
id(key('distinctEvents', parameter[@name='name']/@value))]" />
int_str2_t e_str2[<xsl:value-of select="count($events) + 1" />] =
{
  {  0, "e0", "_initializing_" }
 <xsl:choose>
                 <xsl:when test="count($events) > 0">,
  <xsl:for-each select="//conditionNode[generate-id(.) = generate-
id(key('distinctEvents', @name))][@type='EVENT']">
      <xsl:sort select="@name" />
  {  <xsl:value-of select="@name" />, "e<xsl:value-of select="@name" />",
"<xsl:value-of
select="ancestor::stateMachine/nodeDescriptions/node[@name=current()/@name][@typ
e=current()/@type]/@description" />" }
      <xsl:if test="not(position()=last())">,</xsl:if>
  </xsl:for-each>
 </xsl:when>
                 <xsl:otherwise></xsl:otherwise>
            </xsl:choose>
};

<xsl:variable name="inputVariables" select="//conditionNode[generate-id(.) =
generate-id(key('distinctInputVariables', parameter[@name='name']/@value))]" />
<xsl:choose>
                 <xsl:when test="count($inputVariables) > 0">
str2_t x_str2[<xsl:value-of select="count($inputVariables)" />] =
{
  <xsl:for-each select="$inputVariables">
    <xsl:sort select="@name" />
```

```
    { "x<xsl:value-of select="@name" />", "<xsl:value-of
select="ancestor::stateMachine/nodeDescriptions/node[@name=current()/@name][@typ
e=current()/@type]/@description" />" }
    <xsl:if test="not(position()=last())">,</xsl:if>
  </xsl:for-each>
};
</xsl:when>
                  <xsl:otherwise>
str2_t* x_str2 = NULL;
</xsl:otherwise>
            </xsl:choose>

<xsl:variable name="outputVariables" select="//actionNode[generate-id(.) =
generate-id(key('distinctOutputActions', parameter[@name='name']/@value))]" />
<xsl:choose>
                  <xsl:when test="count($outputVariables) > 0">
str2_t z_str2[<xsl:value-of select="count($outputVariables)" />] =
{
  <xsl:for-each select="$outputVariables">
    <xsl:sort select="@name" />
    { "z<xsl:value-of select="@name" />", "<xsl:value-of
select="ancestor::stateMachine/nodeDescriptions/node[@name=current()/@name][@typ
e=current()/@type]/@description" />" }
    <xsl:if test="not(position()=last())">,</xsl:if>
  </xsl:for-each>
};
</xsl:when>
                  <xsl:otherwise>
str2_t* z_str2 = NULL;
</xsl:otherwise>
            </xsl:choose>

<xsl:for-each select="stateMachine">
str2_t a<xsl:value-of select="@name" />_str2[<xsl:value-of
select="//state[count(state)=0]" />] =
{
  <xsl:for-each select="//state[count(state)=0]">
    { "<xsl:value-of select="@name" />", "<xsl:value-of select="@description"
/>" }
    <xsl:if test="not(position()=last())">,</xsl:if>
  </xsl:for-each>
};
</xsl:for-each>


str3_t A_str3[<xsl:value-of select="count(stateMachine)" />] =
{
<xsl:for-each select="stateMachine">
  { "A<xsl:value-of select="@name" />", "<xsl:value-of select="description" />",
a<xsl:value-of select="@name" />_str2 }
  <xsl:if test="not(position()=last())">,</xsl:if>
</xsl:for-each>
};


//----------------------------------------------------------------------------
void e_find(ubyte e, const char** n, const char** n_name)
{
  static const char* nothing = "No description found!";
  *n = nothing;
  *n_name = nothing;
```

```
  for(uint i = 0; i &lt; 2; i++)
    if(e_str2[i].dig == e){
      *n = e_str2[i].n; *n_name = e_str2[i].n_name; return;
    }
}
//----------------------------------------------------------------------------

//----------------------------------------------------------------------------
void log_a_begin_user(const char* a, const char* a_name, const char* y, const
char* y_name, const char* e, const char* e_name);
void log_a_begin(ubyte a, ubyte y, ubyte e)
{
  const char *e_n, *e_n_name;
  e_find(e, &amp;e_n, &amp;e_n_name);
  log_a_begin_user(A_str3[a].n, A_str3[a].n_name, A_str3[a].str2[y].n,
A_str3[a].str2[y].n_name, e_n, e_n_name);
}
//----------------------------------------------------------------------------
void log_a_trans_user(const char* a, const char* a_name, const char* yo, const
char* yo_name, const char* yn, const char* yn_name);
void log_a_trans(ubyte a, ubyte yo, ubyte yn)
{
  log_a_trans_user(A_str3[a].n, A_str3[a].n_name, A_str3[a].str2[yo].n,
A_str3[a].str2[yo].n_name, A_str3[a].str2[yn].n, A_str3[a].str2[yn].n_name);
}
//----------------------------------------------------------------------------
void log_a_end_user(const char* a, const char* a_name, const char* y, const
char* y_name, const char* e, const char* e_name);
void log_a_end(ubyte a, ubyte y, ubyte_u101 ?)
{
  const char *e_n, *e_n_name;
  e_find(e, &amp;e_n, &amp;e_n_name);
  log_a_end_user(A_str3[a].n, A_str3[a].n_name, A_str3[a].str2[y].n,
A_str3[a].str2[y].n_name, e_n, e_n_name);
}
//----------------------------------------------------------------------------
void log_x_user(const char* x, const char* x_name, ubyte res);
void log_x(ubyte x, ubyte res)
{
  log_x_user(x_str2[x].n, x_str2[x].n_name, res);
}
//----------------------------------------------------------------------------
void log_z_user(const char* z, const char* z_name);
void log_z(ubyte z)
{
  log_z_user(z_str2[z].n, z_str2[z].n_name);
}
//----------------------------------------------------------------------------
void log_write_user(char ch, const char* str);
void log_write(char ch, const char* str)
{
  log_write_user(ch, str);
}
//----------------------------------------------------------------------------

#endif
</xsl:template>
      </xsl:stylesheet>
```

log.h

```
<?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">
//--- this file is machine generated ---

#ifndef LogH
#define LogH

#include "types.h"

#define SWITCH_LOGGING

#ifdef SWITCH_LOGGING
  #define Z_LOGGING
  #define X_LOGGING
  #define A_BEGINS_LOGGING
  #define A_TRANS_LOGGING
  #define A_ENDS_LOGGING
  #define A_ERRORS_LOGGING

  enum{
    LOG_Z = '*',
    LOG_X = '>',
    LOG_GRAPH_BEGIN = '{',
    LOG_GRAPH_TRANS = 'T',
    LOG_GRAPH_END = '}',
    LOG_GRAPH_ERROR = 'E'
  };

  void log_a_begin(ubyte a, ubyte y, ubyte e);
  void log_a_trans(ubyte a, ubyte yo, ubyte yn);
  void log_a_end(ubyte a, ubyte y, ubyte e);
  void log_x(ubyte x, ubyte res);
  void log_z(ubyte z);
  void log_write(char, const char* str);
#endif

#ifdef A_BEGINS_LOGGING
  <xsl:for-each select="stateMachine">
      #define A<xsl:value-of select="@name" />_BEGIN_LOGGING
  </xsl:for-each>

  <xsl:variable name="relatedAutomatas" select="//*[ ((name()='actionNode') and
((@type='AUTOMATA_CALL') or (@type='SIMPLE_AUTOMATA_CALL_IN_STATE'))) or

((name()='conditionNode') and ((@type='OTHER_AUTOMATA_EVENT_NOTEQUAL') or
(@type='OTHER_AUTOMATA_EVENT_EQUAL')))]

      [not(parameter[@name='automata']/@value=ancestor::model/stateMachine/@name
)]
            [not(parameter[@name='automata']/@value=preceding::*[
((name()='actionNode') and ((@type='AUTOMATA_CALL') or
(@type='SIMPLE_AUTOMATA_CALL_IN_STATE'))) or

             ((name()='conditionNode') and
((@type='OTHER_AUTOMATA_EVENT_NOTEQUAL') or
(@type='OTHER_AUTOMATA_EVENT_EQUAL')))]/parameter[@name='automata']/@value)]" />
  <xsl:for-each select="$relatedAutomatas">
```

```
      <xsl:sort select="parameter[@name='automata']/@value" />
      #define A<xsl:value-of select="parameter[@name='automata']/@value"
/>_BEGIN_LOGGING
  </xsl:for-each>

#endif

#ifdef A_TRANS_LOGGING
  <xsl:for-each select="stateMachine">
      #define A<xsl:value-of select="@name" />_TRANS_LOGGING
  </xsl:for-each>

  <xsl:for-each select="$relatedAutomatas">
      <xsl:sort select="parameter[@name='automata']/@value" />
      #define A<xsl:value-of select="parameter[@name='automata']/@value"
/>_TRANS_LOGGING
  </xsl:for-each>

#endif

#ifdef A_ENDS_LOGGING
  <xsl:for-each select="stateMachine">
      #define A<xsl:value-of select="@name" />_END_LOGGING
  </xsl:for-each>

  <xsl:for-each select="$relatedAutomatas">
      <xsl:sort select="parameter[@name='automata']/@value" />
      #define A<xsl:value-of select="parameter[@name='automata']/@value"
/>_END_LOGGING
  </xsl:for-each>

#endif

#ifdef A_ERRORS_LOGGING
  <xsl:for-each select="stateMachine">
      #define A<xsl:value-of select="@name" />_ERRORS_LOGGING
  </xsl:for-each>


  <xsl:for-each select="$relatedAutomatas">
      <xsl:sort select="parameter[@name='automata']/@value" />
      #define A<xsl:value-of select="parameter[@name='automata']/@value"
/>_ERRORS_LOGGING
  </xsl:for-each>
#endif

#endif
</xsl:template>
      </xsl:stylesheet>
```

x.cpp

```
<?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:key name="distinctInputVariables"
match="//conditionNode[@type='INPUT_VARIABLE']"
use="parameter[@name='name']/@value" />
      <xsl:template match="/model">
//--- this file is machine generated ---
```

```
#include "StdAfx.h"
#include "common.h"
#include "log.h"

<xsl:for-each select="//conditionNode[generate-id(.) = generate-
id(key('distinctInputVariables', parameter[@name='name']/@value))]">
    <xsl:sort select="@type" />
    <xsl:sort select="@name" />

    //-----------------------------------------------------------------------------
-
    ubyte x<xsl:value-of select="@name" />_user(void);
    ubyte x<xsl:value-of select="@name" />(void)
    {
      ubyte b = x<xsl:value-of select="@name" />_user();

      #ifdef X_LOGGING
        log_x(<xsl:value-of select="position()-1" />, b);
      #endif

      return b;
    }
</xsl:for-each>

</xsl:template>
      </xsl:stylesheet>
```

x_user.cpp

```
<?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:key name="distinctInputVariables"
match="//conditionNode[@type='INPUT_VARIABLE']"
use="parameter[@name='name']/@value" />
      <xsl:template match="/model">
#include "StdAfx.h"
#include "common.h"

<xsl:for-each select="//conditionNode[generate-id(.) = generate-
id(key('distinctInputVariables', parameter[@name='name']/@value))]">
    <xsl:sort select="@type" />
    <xsl:sort select="@name" />

    //-----------------------------------------------------------------------------
    ubyte x<xsl:value-of select="@name" />_user(void)
    { // <!--xsl:value-of
select="ancestor::stateMachine/nodeDescriptions/node[@name=current()/@name][@typ
e=current()/@type]/@description"/-->

      //---- Place your code here ----

      return 1;
    }
</xsl:for-each>


</xsl:template>
</xsl:stylesheet>
```

z.cpp

```
<?xml version='1.0' ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:output method='text' />
      <xsl:key name="distinctOutputActions"
match="//actionNode[@type='OUTPUT_ACTION']" use="parameter[@name='name']/@value"
/>
      <xsl:template match="/model">
//--- this file is machine generated ---

#include "StdAfx.h"
#include "common.h"
#include "log.h"

<xsl:for-each select="//actionNode[generate-id(.) = generate-
id(key('distinctOutputActions', parameter[@name='name']/@value))]">
    <xsl:sort select="@type" />
    <xsl:sort select="@name" />

    //--------------------------------------------------------------------------
-
    void z<xsl:value-of select="@name" />_user(void);
    void z<xsl:value-of select="@name" />(void)
    {
      z<xsl:value-of select="@name" />_user();

      #ifdef Z_LOGGING
        log_z(<xsl:value-of select="position()-1" />);
      #endif
    }
</xsl:for-each>

</xsl:template>
      </xsl:stylesheet>
```

z_user.cpp

```
<?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:key name="distinctOutputActions"
match="//actionNode[@type='OUTPUT_ACTION']" use="parameter[@name='name']/@value"
/>
      <xsl:template match="/model">
#include "StdAfx.h"
#include "common.h"


<xsl:for-each select="//actionNode[generate-id(.) = generate-
id(key('distinctOutputActions', parameter[@name='name']/@value))]">
    <xsl:sort select="@type" />
    <xsl:sort select="@name" />

    //--------------------------------------------------------------------------
-
    void z<xsl:value-of select="@name" />_user(void)
```

```
    { // <xsl:value-of
select="ancestor::stateMachine/nodeDescriptions/node[@name=current()/@name][@typ
e=current()/@type]/@description" />

      //---- Place your code here ----
    }
</xsl:for-each>

class A
{
    static int Main(char** args)
    {
        return 1;
    }
} a;

</xsl:template>
      </xsl:stylesheet>
```