САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ МЕХАНИКИ И ОПТИКИ

Кафедра «Компьютерных технологий»

Е. О. Решетников

Инструментальное средство для поддержки автоматного программирования в среде разработки Microsoft Visual Studio 2008

Санкт-Петербург

2009

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время все большую популярность приобретают технологии разработки программного обеспечения, включающие в себя этапы моделирования, валидации, а также верификации разрабатываемого приложения. На этапе моделирования формируется архитектура приложения. В зависимости от типа модели, возможна спецификация той или иной части приложения. Наибольшее распространение получило моделирование приложений при помощи *UML*-диаграмм [1]. Наличие моделей, тесно связанных с исходным кодом, позволяет облегчить проверку правильности приложения, написание документации по любой части приложения, а также процесс рефакторинга и улучшения внутренней структуры приложения.

Валидация моделей предотвращает допущение ошибок еще на этапе проектирования приложения, что оказывает благотворное влияние на время разработки приложения. Верификация полученных моделей более широко использует семантику разрабатываемого приложения, специфику языка программирования, на котором ведется разработка, и прочие тонкости разработки конкретных алгоритмов и программ. Такой этап проверки модели приложения выявляет часть возможных «подводных камней», позволяя еще больше уменьшить количество потенциальных ошибок в исходном коде приложения.

Значительная доля разрабатываемого программного обеспечения приходится на языки программирования высокого уровня, такие как: *Java*, *C++*, *C#* и прочие. Среда разработки *Microsoft Visual Studio* [2] является основной средой разработки программных продуктов для семейства операционных систем *Microsoft Windows*. Версия *Visual Studio* 2008 содержит несколько компонентов для построения моделей: дизайнер распределенных систем, дизайнер баз данных, дизайнер классов, новые дизайнеры редактирования диаграммы последовательностей *(Sequential Workflow)* и диаграммы состояний *(State Machine Workflow)*, а также дополнительные

модули, позволяющие разрабатывать собственные визуальные редакторы моделей.

Одной из основных *UML*-диаграмм, позволяющих отображать поведение системы, является диаграмма состояний. Задача разработки удобного компонента для создания и редактирования диаграмм состояний и генерации по ним исходного кода становится все более актуальной.

1. ПОСТАНОВКА ЗАДАЧИ

В 2007 году автором данной работы была написана работа [3] и разработано инструментальное средство, позволяющее использовать конечные автоматы при написании приложений в среде программирования *Microsoft Visual Studio 2005*. Описанное инструментальное средство не было лишено недостатков. К этим недостаткам можно отнести, например, отсутствие возможности описывать на диаграмме вложенные автоматы, повторно использовать спроектированные автоматы, а также отсутствие синхронного режима обработки входных воздействий. После версии 2005 года, была выпущена новая версия среды разработки *Microsoft Visual Studio 2008*. Компонент специализированных языков предметной области *Microsoft Domain-Specific Language Tools* претерпел ряд изменений, благодаря которым стало возможно расширение визуальных возможностей инструментального средства.

Однако, в случае расширения функциональности инструментального средства, должен быть улучшен и механизм генерации кода. При наличии вложенных автоматов на диаграмме, сгенерированный код при исполнении должен реализовывать работу таких автоматов. Таким образом, появляется задача выбора метода представления и реализации конечных автоматов для обеспечения возможности добавления нового поведения автоматов и их взаимодействия.

ВЫВОДЫ ПО ГЛАВЕ 1

Автоматные модели, которые могут быть разработаны с использованием инструментального средства, разработанного в работе [3], имеют большие ограничения. Данные ограничения не допустимы в случае проектирования сложных автоматных систем, имеющих такие особенности, как: наличие вложенных автоматов, многократное использование одних и тех же автоматов в различных частях приложения, исполнение автоматов в различных режимах обработки входных, выходных воздействий и других.

Для устранения вышеописанных ограничений, должны быть расширены возможности инструментального средства по визуальному проектированию автоматов и их взаимодействия. Так как разрабатываемые автоматные системы будут обладать большей сложностью, по сравнению с автоматами работы [3], должен быть улучшен механизм генерации кода. Для этого следует выбрать подходящий метод представления и реализации конечных автоматов.

В следующих главах рассмотрены основные способы реализации конечных автоматов, приведен обзор существующих инструментальных средств, поддерживающих автоматное программирование и не вошедших в обзор работы [3], описана разработанная автором библиотека для автоматного программирования *FSMLib*, а также улучшенное инструментальное средство *State Machine Designer*, позволяющее проектирование сложных автоматных моделей.

2. МЕТОДЫ РЕАЛИЗАЦИИ КОНЕЧНЫХ АВТОМАТОВ

На протяжении многих лет модель конечных автоматов применяли в программировании. Наиболее полезной для программирования с использованием автоматов является модель автоматов Мили [4], то есть модель, в которой задано множество состояний, входных и выходных воздействий, а также функции переходов и выходных воздействий, в зависимости от текущего состояния автомата и текущего входного воздействия.

В настоящее время известно несколько подходов к реализации конечных автоматов. При рассмотрении этих методов нас будут интересовать такие параметры, как: количество описываемых классов и создаваемых объектов в зависимости от количества состояний автомата, общая производительность, степень адаптивности метода к новым требованиям.

2.1. SWITCH-технология

Технология была предложена А. А. Шалыто в 1991 году [5]. В данной технологии базовыми понятиями являются термины: «состояние», «входное воздействие», «выходное воздействие». Описываемый конечный автомат состоит из заданного набора состояний. Переход из одного состояния в другое осуществляется по заданному входному воздействию при выполнении заданного условия. Вся логика описываемого автомата реализуется в одном методе класса, которому добавляется автоматное поведение. Метод реализуется с помощью операторов ветвления switch и условных операторов if. В общем виде последовательность действий метода следующая:

1. Получение нового входного воздействия.
2. Выполнение оператора switch для поиска текущего состояния в части кода, отвечающей за переход в новое состояние.
3. Выполнение оператора switch для поиска текущего входного воздействия в части кода; отвечающей за переход в новое состояние из текущего состояния.

4. Выполнение оператора if для определения удовлетворенности необходимых для перехода условий.
5. Выполнение заданного выходного воздействия.
6. Переход в новое состояние.

Основное преимущество *SWITCH*-технологии перед другими методами представления автоматов в том, что вся логика переходов и изменения состояний описывается внутри одного метода. Таким образом, не создается дополнительных классов и объектов. Единственным расширением класса, которому добавляется автоматное поведение, должен стать объект для хранения текущего состояния автомата. Причем, в случае кодирования состояний целыми числами, появится всего одна переменная целого типа. Такой подход экономит память и увеличивает скорость выполнения кода автомата.

2.2. Паттерн State

Паттерн *State* является методом реализации объекта, изменяющего свое поведение в зависимости от состояния [6]. Так как в указанной работе данный паттерн описан недостаточно полно, то в различных работах и источниках реализуют данный паттерн по-разному.

В общем, так или иначе, реализацию паттерна *State* можно представить в следующем виде:

1. Существует интерфейс выходных воздействий, которые определены для всего автомата, реализуемый классом автомата или интерфейс выходного воздействия, реализуемого каждым состоянием.
2. Каждое состояние автомата определяется в отдельном классе, реализующем общий для всех состояний интерфейс.
3. Класс автомата хранит в себе текущее состояние автомата и предоставляет функциональность для передачи автомату входного воздействия.

4. При появлении входного воздействия, текущее состояние обрабатывает данное входное воздействие и меняет текущее состояние автомата на состояние, в которое должен быть осуществлен переход.

Таким образом, логика по выполнению переходов из состояния, находится в самом состоянии. Данная концепция представления автомата является непрозрачной в ситуации, когда по написанному коду требуется понять поведение автомата в целом, либо в ситуации, когда автомат должен быть изменен: например, добавление или удаление состояний и переходов, потребует изменения сразу нескольких классов.

2.3. Паттерн State Machine

Качественное улучшение паттерна *State* реализовано в работе [7]. Полученный паттерн был назван *State Machine*. Отличием данного паттерна от паттерна *State* является в том, что логика по переходу между состояниями и выполнением выходных воздействий вынесена в класс автомата. Также автомат реализует интерфейс уведомления о входных воздействиях для взаимодействия объектов состояний с автоматом.

Такой подход к реализации автомата существенно упрощает процесс модификации автомата: добавление новых состояний, переходов и т.д. Также авторы работы [7] предлагают возможные модификации паттерна *State Machine* для хранения модели данных, быстрого поиска состояния, в которое должен быть осуществлен переход, а также протоколирования действий автомата.

Еще одним расширением паттерна *State* является работа [8]. В данной работе при помощи виртуальных вложенных классов появляется возможность расширения функциональности автоматного объекта с помощью механизма наследования, а также обобщения и повторного использования похожего поведения в разных режимах работы.

В работе [9] рассмотрена проблема совместного использования объектно-ориентированного и автоматно-ориентированного подходов программирования. Расширение логики автоматов в данной работе происходит с помощью механизма наследования виртуальных методов.

2.4. Декларативный подход

В работе [10] был предложен декларативный подход к вложению и наследованию автоматных классов при использовании императивных языков программирования. В данной работе предлагается использовать технологию *Reflection* для обеспечения работы автомата. При этом инициализация автомата происходит в декларативной части исходного кода. Примеры реализации конечных автоматов в работе [10] написаны на языке *C#*. Для того чтобы реализовать конечный автомат с использованием данного подхода, необходимо воспользоваться библиотекой, разработанной в рамках той же работы. Каждый класс, представляющий состояние автомата, должен быть унаследован от класса *State* библиотеки. Затем такой класс может быть добавлен в качестве атрибута языка *C#* к другому классу-наследнику *State*. Таким образом, в рассматриваемом подходе одни состояния добавляются внутрь других. Состояние, которое не содержится ни в одном из других состояний, можно считать автоматом. Интерфейсы и методы взаимодействующих классов при этом реализуются разработчиком. Технология упрощает только процедуру перенаправления вызова метода текущему состоянию автомата, в то время как функции переходов должны быть заданы внутри методов классов, представляющих состояния автомата.

2.5. Библиотеки для автоматного программирования

Другим способом задания модели конечного автомата является написание библиотеки, содержащей классы, соответствующие основным элементам автомата. Главным преимуществом такого подхода является наглядность создания автомата, а также относительная легкость добавления в автомат новой функциональности. Также к плюсам такого подхода следует

отнести возможность валидации и верификации построенной модели автомата. В случае использования *SWITCH*-технологии или паттерна *State* и его модификаций, по данному исходному коду автомата, затруднительно провести любую проверку правильности построенной модели. Возможными проблемами и неудобствами при введении библиотеки классов автоматных сущностей, являются проблемы предоставления общих данных внутри различных частей одного автомата, а также низкое быстродействие и большее количество используемой памяти в связи с работой с большим количеством создаваемых объектов.

2.5.1. Библиотека *Jilles van Gurp* и *Jan Bosch*

В работе [11] разработана библиотека классов для описания и использования модели конечных автоматов при программировании. Библиотека состоит из четырех классов и одного интерфейса:

1. *FSMContext* – класс, представляющий автомат и поддерживающий ссылки на текущее состояние автомата и коллекцию объектов, представляющих собой выходные воздействия автомата.
2. *Transition* – класс, представляющий переход. Данный класс имеет единственный метод *Execute,* который вызывается из состояния при выполнении данного перехода.
3. *State* – класс, отображающий состояние автомата. Все состояния автомата являются экземплярами данного класса. Класс предоставляет метод для связывания переходов с входными воздействиями автомата, а также метод обработки входного воздействия. При поступлении входного воздействия, автоматически происходит поиск нужного перехода, затем выполняется действие на переходе, а затем конечное состояние перехода становится текущим состоянием автомата и выполняется действие при входе в новое состояние.

4. *FSM* – данный класс предназначен для создания автомата и инициализации его свойств.
5. *FSMAction* – это интерфейс, который должен быть реализован всеми классами, объекты которых представляют собой выходные воздействия автомата.

Таким образом, автомат представляет собой иерархическую структуру объектов заданных классов. Большим неудобством данной библиотеки является необходимость создания отдельного класса на каждое выходное воздействие. В случае, когда различных выходных воздействий много, полученный код становится трудно поддерживаемым. Авторы работы сами отмечают некоторые недостатки библиотеки, например отсутствие механизма работы с условными переходами.

2.5.2. Библиотека *STOOL*

Более мощная библиотека для реализации автоматов является библиотека *STOOL*, подробно описанная в работе [12]. Данная библиотека предоставляет автоматическое протоколирование исполнения автомата, а также механизм обработки ошибок, возникающих при работе системы. Приведем описание основных классов библиотеки:

1. *Auto* – базовый класс для разработки классов автоматов. Класс разрабатываемого автомата должен переопределять метод, реализующий граф переходов.
2. *State* – класс, представляющий состояние автомата. Данный класс обеспечивает протоколирование любого изменения состояния автомата.
3. *Input* – базовый класс для реализации входных воздействий.
4. *Output* – базовый класс для реализации выходных воздействий.

Библиотека содержит еще ряд классов, обеспечивающих контролирование процесса выполнения выходного воздействия, осуществляющих управление системой автоматов, управление системными

переходами, а также созданием и уничтожением переходов для использования в однопоточных и многопоточных системах.

Данная библиотека является полноценной с точки зрения ее применения при решении широкого спектра задач. К неудобствам работы с данной библиотекой можно причислить необходимость ручного определения функции перехода для автомата (способ ее реализации ложится на плечи разработчика, использующего библиотеку), а также наличие отдельных классов для каждого входного и выходного воздействия.

ВЫВОДЫ ПО ГЛАВЕ 2

В настоящий момент существует несколько способов реализации конечных автоматов. Все эти способы можно разделить на две принципиально различные группы. К первой группе можно отнести способы задания автоматной модели с явным выделением автоматных сущностей. То есть такие подходы к реализации конечных автоматов, как: паттерн *State* и его модификации, библиотеки для автоматного программирования. Ко второй группе можно отнести *SWITCH*-технологию, в которой состояния автомата кодируются, а вся логика конечного автомата заключена в одном методе класса. Методы представления конечных автоматов, относящиеся к первой группе, требуют большего количества системных ресурсов, но в то же время позволяют разработчику проще вносить изменения в логику конечного автомата.

В случае, когда метод реализации конечных автоматов служит основой для автоматической генерации исходного кода по визуальной модели, меняются и критерии, по которым следует выбирать данный метод. Теоретически, любой из вышеописанных способов представления коечных автоматов может быть использован в качестве такой основы. Однако, учитывая, что при любом расширении функциональности визуального редактора, соответствующие изменения должны быть осуществлены в генерируемом коде, использование библиотечного подхода к представлению конечных автоматов является более предпочтительным.

3. ОБЗОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ

В работах [3, 13] был выполнен обзор инструментальных средств, позволяющих визуальное построение моделей конечных автоматов и автоматическую генерацию кода по полученным моделям. Рассмотрим инструментальные средства, которые не были упомянуты в обзоре.

3.1. IAR visualSTATE

Данный продукт разработан компанией *IAR Systems* [14]. Он позволяет проектирование моделей конечных автоматов и генерацию по ним кода на языках *Assembler*, *C/C++*. Продукт является мощной средой разработки приложений с помощью моделей конечных автоматов и содержит в себе: графическую среду разработки, инструменты для тестирования приложения, компонент для генерации кода по моделям, компонент для генерации документации по моделям. Данный продукт также поддерживает встроенную валидацию и верификацию моделей для предотвращения допущения ошибок в момент разработки приложения. Для представления моделей автоматов используются *State Chart UML*-диаграммы. Внешний вид инструментального средства отображен на рис. 1.

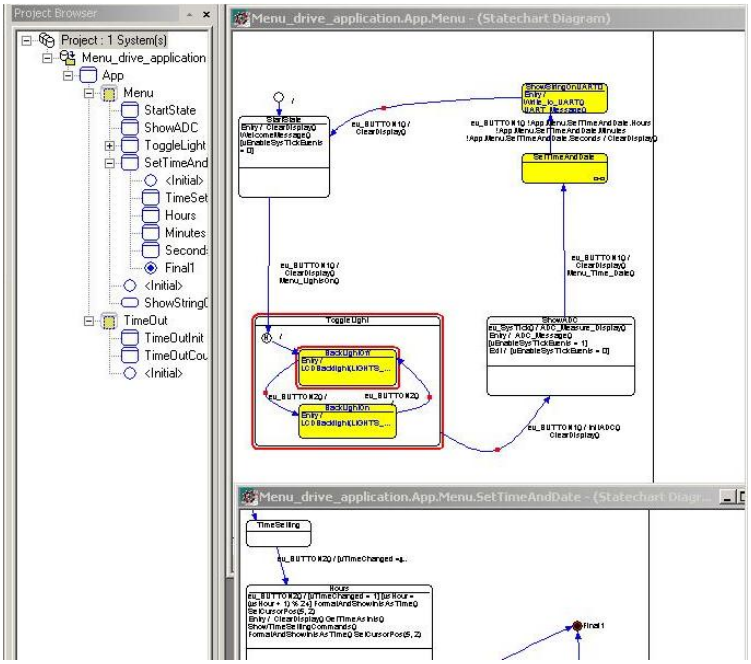

Рис. 1. Использование инструментального средства *IAR visualState* в среде разработки *IAR Embedded Workbench*

Большим плюсом данного продукта является наличие встроенного компилятора и дебаггера, что позволяет полноценно разрабатывать приложения на языках *Assembler*, *C* и *C++*. Минусом является отсутствие поддержки других языков программирования, например *Java*, *C#* и т.д. Также стоит отметить бедность визуального интерфейса инструментального средства. Главное предназначение средства – написание программ для встроенных систем, инструмент вряд ли может быть использован при написании обычного приложения. Однако даже при написании высокоуровневых пользовательских приложений очень часто применение конечных автоматов при разработке является практически необходимым.

3.2. Finite State Machine Editor

Инструментальное средство *Finite State Machine Editor* [15] разрабатывалось в 2004–2006 годах. Средство довольно слабое, позволяет генерировать исходный код автомата по построенной модели на языках *C++* и *Python*. Средство не интегрируется ни в одну из широко распространенных сред разработки, но предоставляет возможность компиляции и отладки написанного кода. Любопытным фактом является то, что подвигло автора на создание данного инструмента. Вот, что он пишет: «Мой шеф добыл немного информации о конечных автоматах и посоветовал мне прочитать книгу «*SWITCH*-технология» Шалыто. В ней я увидел способ использования конечных автоматов для обычного программирования. Я написал первую версию программы *Finite State Machine Editor*. Она была ужасной и содержала много ошибок. Но программы, которые получались с ее помощью, были надежными». Внешний вид инструментального средства представлен на рис. 2.

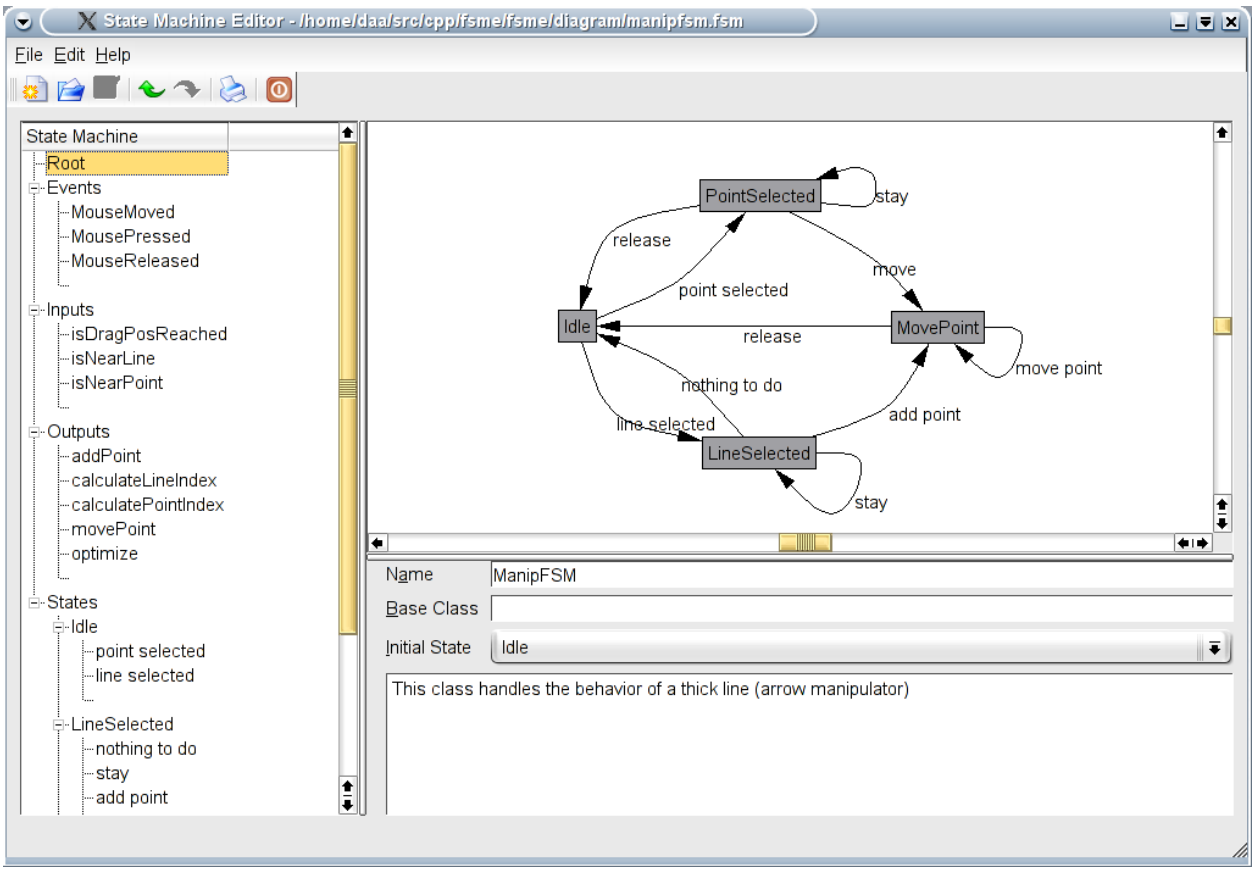

Рис. 2. Проектирование конечных автоматов в инструментальном средстве *Finite State Machine Editor*

3.3. StarUML

Инструментальное средство *StarUML* [16] разрабатывалось с 1996 года группой корейских разработчиков. Данное средство разрабатывалось с целью заменить такие коммерческие продукты, как *Rational Rose*, *Together* и прочие инструменты, поддерживающие редактирование множества *UML*-диаграмм с возможностью генерации по ним текста. В средстве *StarUML* также возможно редактирование множества *UML*-диаграмм, например диаграммы классов, диаграммы последовательностей, диаграммы конечных автоматов и многих других. Генерация текстов по данным диаграммам осуществляется с помощью наличие шаблонов для трансформации диаграмм. Таким образом, данная среда подходит лишь для моделирования различных частей приложения и создания структурированной документации по этим частям, но никак не для разработки приложений с ее использованием. На рис. 3

представлен пример диаграммы, созданной в инструментальном средстве *StarUML*.

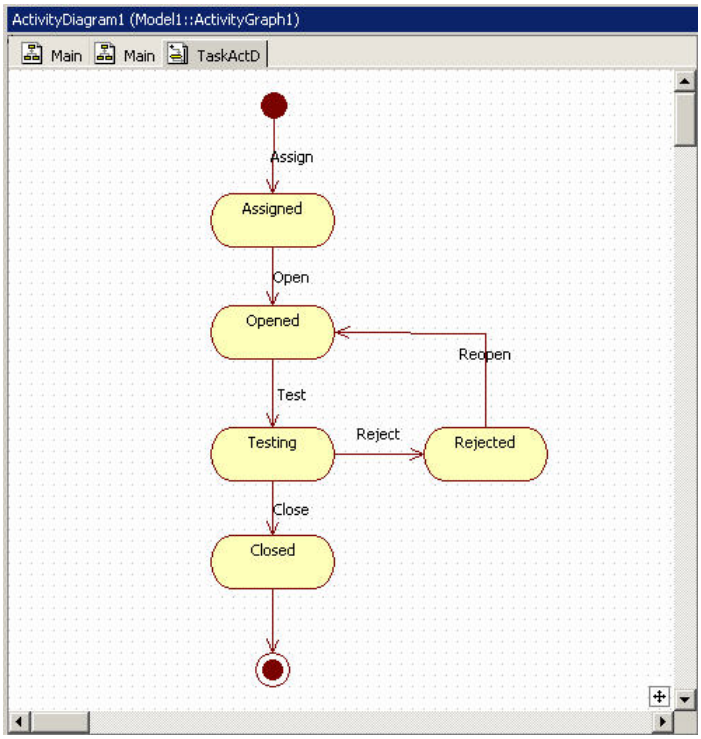

Рис. 3. Диаграмма в *StarUML*

3.4. BOUML

Инструментальное средство *BOUML* [17] разрабатывает французский программист Bruno Pages. Данное средство, как и предыдущее, не имеет возможности исполнения кода, поддерживая лишь его генерацию. Спектр возможностей данного средства шире, чем просто проектирование диаграмм автоматов, видимо, поэтому ничего серьезного для разработки автоматных программ в этом средстве сделать не удастся. Внешний вид приложения представлен на рис. 4.

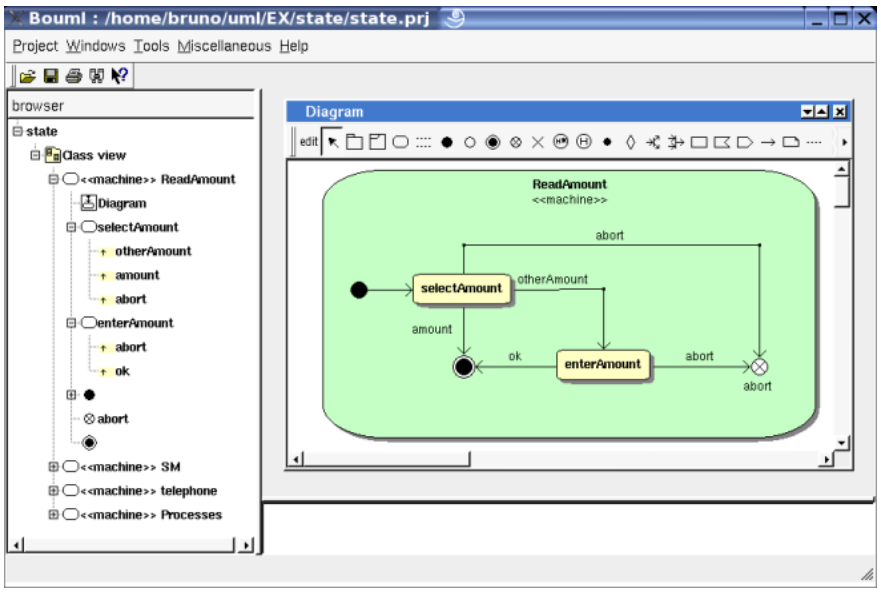

Рис. 4. Проектирование конечного автомата в инструментальном средстве *BOUML*

3.5. State Machine Designer

Инструментальное средство *State Machine Designer* было разработано автором в работе [3]. Данное средство является плагином для среды разработки *Microsoft Visual Studio 2005*. С его помощью возможно редактирование классов проекта с использованием разработанной диаграммы классов, автоматное поведение классам можно задавать с помощью диаграммы автомата. Также в работе поддержана диаграмма объектов для представления начальной конфигурации системы, которая может быть полезна, в случае если в ней фиксированное и относительно небольшое количество объектов.

К недостаткам данного продукта можно отнести отсутствие возможности проектирования вложенных автоматов, а также отсутствие различных режимов обработки входных воздействий автоматом. Автомат, получаемый при помощи автоматической генерации кода по разрабатываемым моделям, всегда работает в асинхронном режиме, обрабатывая входные воздействия в отдельном потоке. Такой подход далеко не всегда удобен в программировании, в частности, в ситуациях, когда для системы важен факт того, что переданное входное воздействие уже было обработано автоматом. Внешний вид среды разработки *Microsoft Visual*

Studio 2005 в момент редактирования диаграммы автомата представлен на рис. 5.

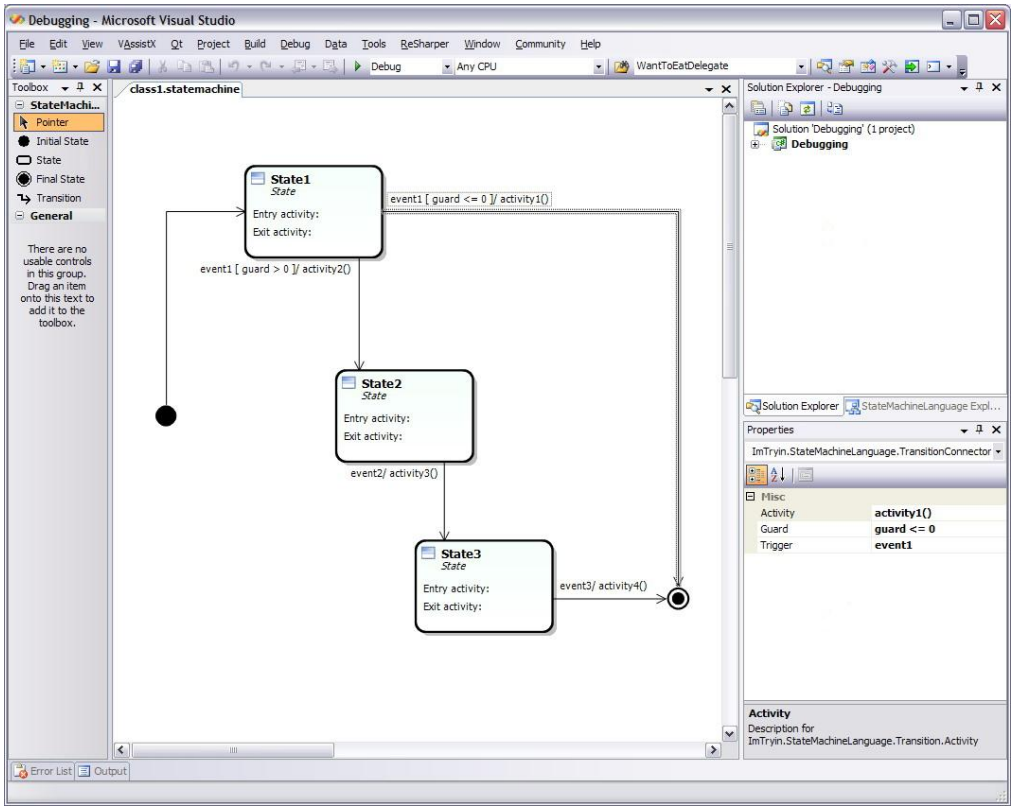

Рис. 5. Редактирование диаграммы автомата с помощью *State Machine Designer* в среде разработки *Microsoft Visual Studio 2005*

ВЫВОДЫ ПО ГЛАВЕ 3

В главе 3 были рассмотрены инструментальные средства для визуального проектирования и использования конечных автоматов при программировании на языках высокого уровня. Каждое из рассмотренных средств имеет свои достоинства и недостатки. Большинство из описанных средств являются бесплатными, что является несомненным плюсом этих продуктов. Однако, ни бесплатные, ни коммерческие инструментальные средства не предоставляют возможности проектирования сложных автоматных систем в одной из наиболее распространенных сред программирования. Всем рассмотренным инструментальным средствам недостает таких возможностей как: проектирование систем с вложенными автоматами, предоставление различных режимов обработки входных и выходных воздействий, повторное использование разработанных автоматов в различных частях приложения.

В главе 4 описывается реализация инструментального средства, которое лишено описанных выше недостатков.

4. РЕАЛИЗАЦИЯ

4.1. Библиотека FSMLib

Автором данной работы разработана библиотека, которая должна, с одной стороны, быть самостоятельной библиотекой для реализации конечных автоматов, а с другой послужить подходящим фундаментом для генерации на ней исходного кода по метамодели автомата, разработанной в графическом редакторе. В данном случае речь идет о графическом редакторе моделей конечных автоматов инструментального средства, описанного в работе ниже.

К библиотеке были предъявлены следующие требования:

1. Классы библиотеки должны образовывать замкнутую систему автомата и предоставлять возможность использования этих классов без необходимости наследования некоторых из них.
2. Автомат должен поддерживать синхронный и асинхронный режимы обработки входных воздействий.
3. Состояния автомата должны естественным образом поддерживать возможность существования вложенных автоматов.
4. Должна быть реализована возможность задания различных режимов для обработки входных воздействий вложенными автоматами.
5. Должна быть поддержана возможность группировки состояний по общим переходам (то есть возможность определять общий переход для группы состояний, а не для каждого состояния в отдельности).

Подробное описание библиотеки приводится в следующем разделе.

4.1.1. Структура классов

Данная библиотека написана на языке программирования *C#*. На рис. 6 приведена диаграмма классов библиотеки.

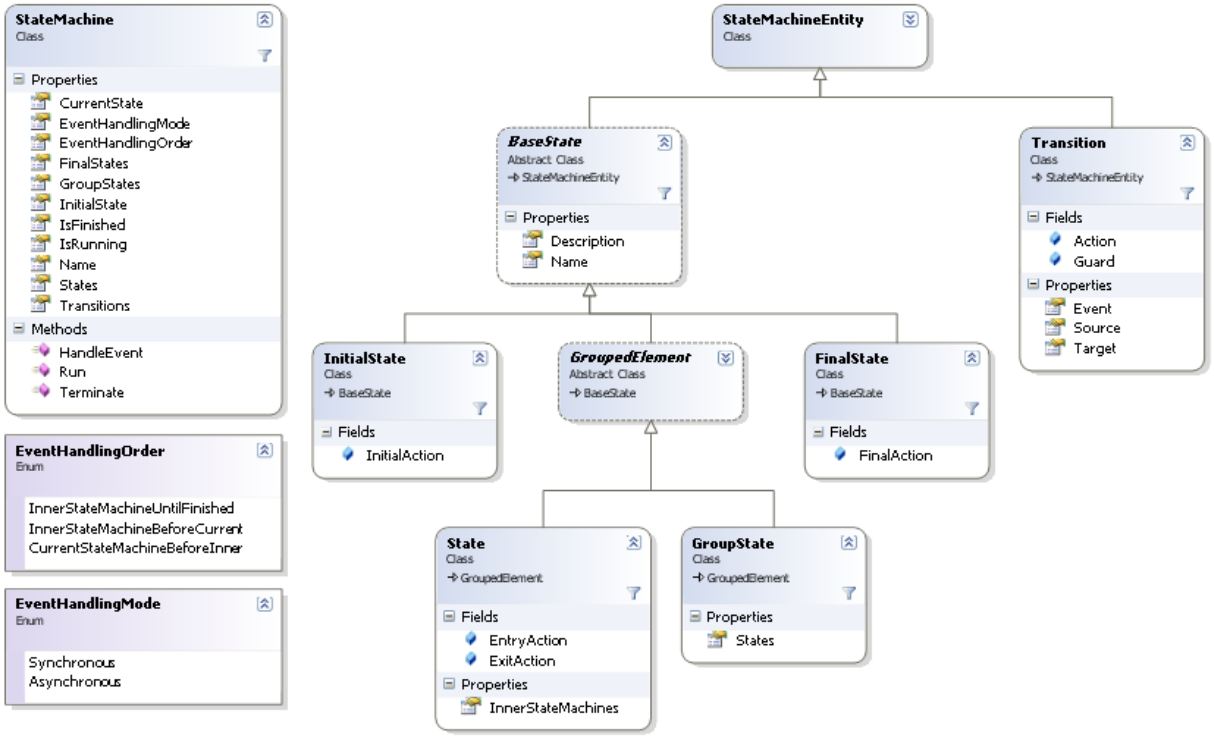

Рис. 6. Диаграмма классов библиотеки *FSMLib*

Рассмотрим подробнее каждый из классов библиотеки:

1. *StateMachine* – класс, представляющий автомат. Предоставляет возможность задания начального состояния *InitialState*, обычных состояний *States*, конечных состояний *FinalStates*, групп состояний *GroupStates*, а также возможность узнать текущее состояние автомата *CurrentState*, запущен ли автомат *IsRunning* и завершил ли он свою работу *IsFinished*. Дополнительно для автомата может быть задано имя *Name*, режим обработки входных воздействий *EventHandlingMode*, а также порядок обработки входных *EventHandlingOrder* воздействий вложенными автоматами.
2. *StateMachineEntity* – базовый класс для всех сущностей автомата. Данный класс не содержит открытых данных.

3. *Transition* – класс, представляющий переход. Данный класс содержит информацию о входном воздействии *Event*, по которому будет совершен переход, начальном состоянии или группе *Source* и конечном состоянии *Target*, а также условии *Guard*, которое должно быть выполнено для перехода и действии на переходе *Action*.
4. *BaseState* – базовый класс для различных состояний автомата. Содержит информацию об имени *Name* и описании состояния *Description*.
5. *GroupedElement* – базовый класс для состояний автомата типа *State* и *GroupState*, которые могут быть сгруппированы.
6. *InitialState* – класс, представляющий начальное состояние автомата. Предоставляет возможность задания метода инициализации автомата *InitialAction*.
7. *State* – класс, представляющий обычное состояние автомата (не являющееся ни начальным, ни конечным, ни группой состояний). Предоставляет возможность задания метода *EntryAction*, выполняемого при входе в состояние, при выходе из состояния *ExitAction*, а также вложенного автомата *InnerStateMachine*.
8. *FinalState* – класс, представляющий конечное состояние автомата. Предоставляет возможность задания завершающего метода *FinalAction* автомата.
9. *GroupState* – класс, представляющий группу состояний. Предоставляет возможность группирования состояний. Каждое состояние может входить не более чем в одну группу. Данный класс был введен в автомат для упрощения проектирования автоматов, в которых несколько состояний имеют общие переходы.

10. *EventHandlingOrder* – класс, представляющий порядок обработки входящих воздействий вложенными автоматами. Подробнее данный класс описан в главе 4.1.4.
11. *EventHandlingMode* – класс, представляющий способ обработки автоматом входящих и выходных воздействий. Подробнее данный класс описан в главе 4.1.5.

Библиотека содержит всего три открытых метода, которые находятся внутри класса *StateMachine*:

1. Метод *Run* запускает выполнение автомата.
2. Метод *HandleEvent* вызывается разработчиком каждый раз, когда автомат должен обработать заданное входное воздействие. Входное воздействие передается строковым параметром.
3. Метод *Terminate* прекращает выполнение автомата.

Пример исходного кода создания нового автомата с использованием описанной библиотеки представлен ниже:

```
public class TestStateMachine : StateMachine
{
    //accessor for TestForm
    public TestForm TestForm { get; set; }

    public TestStateMachine()
    {
        //Initial state
        InitialState initialState = new InitialState();
        initialState.Name = "Initial state";
        initialState.InitialAction = Handler;
        //States
        State state1 = new State();
        state1.Name = "State1";
        state1.EntryAction = Handler;

        //State2
        State state2 = new State();
        state2.Name = "State2";
        state2.EntryAction = Handler;

        //Final state
        FinalState finalState = new FinalState();
        finalState.Name = "Final state";
        finalState.FinalAction = Handler;

        //Transitions
        Transition transition1 = new Transition( "*", initialState, state1 );
        Transition transition2 = new Transition( "e1", state1, state2 );
        Transition transition3 = new Transition( "e2", state2, finalState );

        //Assignments
        InitialState = initialState;

        States.Add( state1 );
```

```
            States.Add( state2 );

            FinalStates.Add( finalState );

            Transitions.Add( transition1 );
            Transitions.Add( transition2 );
            Transitions.Add( transition3 );
        }

        //this method prints current state name on the specified textbox of TestForm
        public void Handler() { TestForm.textBox1.Text = CurrentState.Name; }
    }
```

Приведенный код описывает автомат с четырьмя состояниями: *initialState*, *state1*, *state2*, *finalState*. Из начального состояния *initialState* автомат сразу переходит в состояние *state1*. По сообщению *e1* автомат переходит из состояния *state1* в состояние *state2*. По сообщению *e2* автомат завершает свою работу. При входе в каждое состояние выполняется метод *Handler*, который присваивает имя текущего состояния автомата полю *Text* одного из объектов формы *TestForm*.

Исходный код классов библиотеки *FSMLib* приведен в приложении 1.

4.1.2. Группирование состояний

Очень часто при проектировании автоматных моделей появляется необходимость многократно использовать одинаковые переходы из различных состояний. Например, практически любое приложение может быть завершено независимо от того, в каком состоянии оно находится. На рис.7 приведен схематический пример автомата, реализующего логику такого приложения.

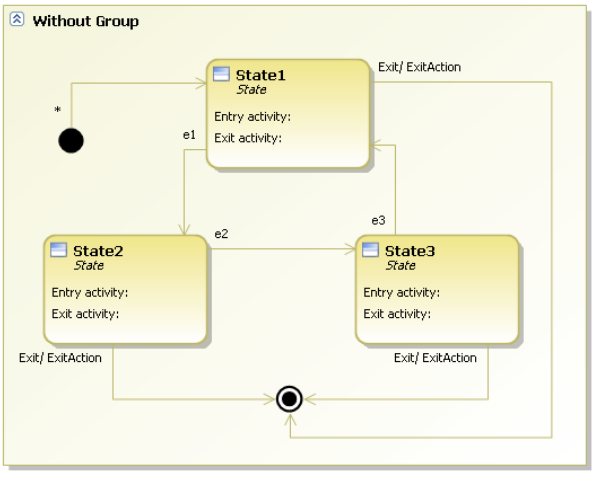

Рис. 7. Пример конечного автомата без использования группы состояний

Как можно заметить из рисунка, автомат содержит три одинаковых перехода из состояний *State1*, *State2*, *State3*, происходящих по входному воздействию *Exit* и выполняющих на переходе действие *ExitAction*. Упоминание в каждом состоянии о наличие перехода в конечное состояние выглядит избыточным. На рис. 8 приведен пример того же автомата, но с использованием группировки состояний.

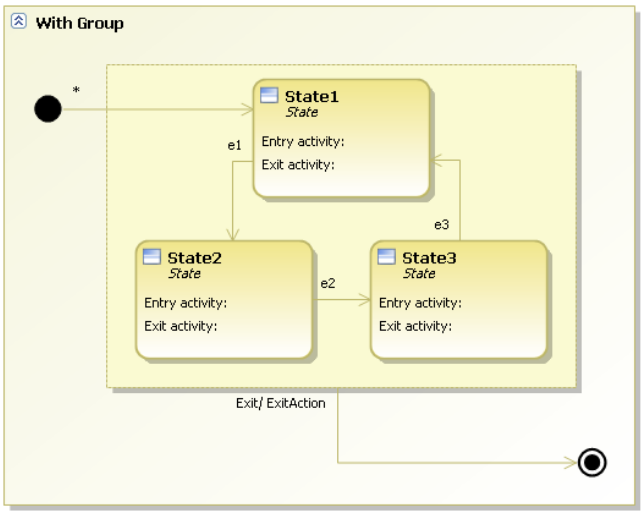

Рис. 8. Пример конечного автомата с использованием группы состояний

Заметим, что состояния *State1*, *State2* и *State3* теперь образуют одну группу с единственным переходом из данной группы по входному воздействию *Exit* в конечное состояние. Автомат на рис. 8 идентичен автомату на рис. 7, но теперь он лишен избыточности благодарю введению группы состояний.

В библиотеке *FSMLib* группировка состояний возможна благодаря введению класса *GroupState*. Данный класс поддерживает добавление в него состояний автомата, а также других групп. При этом для каждого добавленного состояния или добавленной группы автоматически проставляется указатель на родительскую группу. В случае, когда разработчик пытается добавить одно состояние в несколько групп, генерируется исключение. Наличие в автоматной модели объектов типа *GroupState* позволяет задавать переходы в некоторое состояние из соответствующей группы. В случае, когда автомат получает входное воздействие, текущее состояние принадлежит какой-либо группе, и

существует переход по полученному входному воздействию из группы, то такой переход выполняется первым. В случае многократной вложенности групп, воздействие обрабатывается в порядке от непосредственной родительской группы к ее родительской группе и т.д.

Таким образом, используя группы состояний при разработке можно избежать дублирования переходов из нескольких состояний в одно по одним и тем же входным воздействиям с одинаковыми условиями на переходах.

4.1.3. Использование вложенных автоматов

Среди задач, которые могут быть эффективно решены с применением конечных автоматов, существует ряд задач, где оправдано применение вложенных автоматов. Вложенный автомат – это автомат, который начинает свое выполнение в момент, когда родительский автомат приходит в некоторое состояние. Применение вложенных автоматов довольно часто встречается в задачах автоматизации производства. При этом главный автомат представляет собой схему основного технологического процесса, а вложенные автоматы работу механизмов, которые приводятся в действия по достижении главным автоматом определенного состояния и завершаются при выходе главного автомата из этого состояния.

Задача реализации работы вложенных автоматов не является такой простой, как кажется на первый взгляд. Простым случаем является наличие не более чем одного вложенного автомата для каждого состояния. В таком случае выделение отдельного автомата служит лишь для упрощения общей схемы автомата, а также для возможности повторного использования вложенного автомата. Сложнее выглядит ситуация, когда состояние главного автомата имеет несколько вложенных автоматов. Такое возможно, например, при технологическом процессе, в некоторый момент которого должны быть запущены автоматы, обеспечивающие вспомогательные действия. Рассмотрим технологический процесс, в котором должна быть проведена вентиляция нескольких элементов автоматизируемой установки. Если

вентиляцию каждого элемента установки представить в виде автомата, то одновременное вложение всех таких автоматов в состояние главного автомата приведет к одновременной вентиляции всех деталей установки. Схематическое изображение решение задачи вентилирования элементов установки без применения вложенных автоматов представлено на рис. 9.

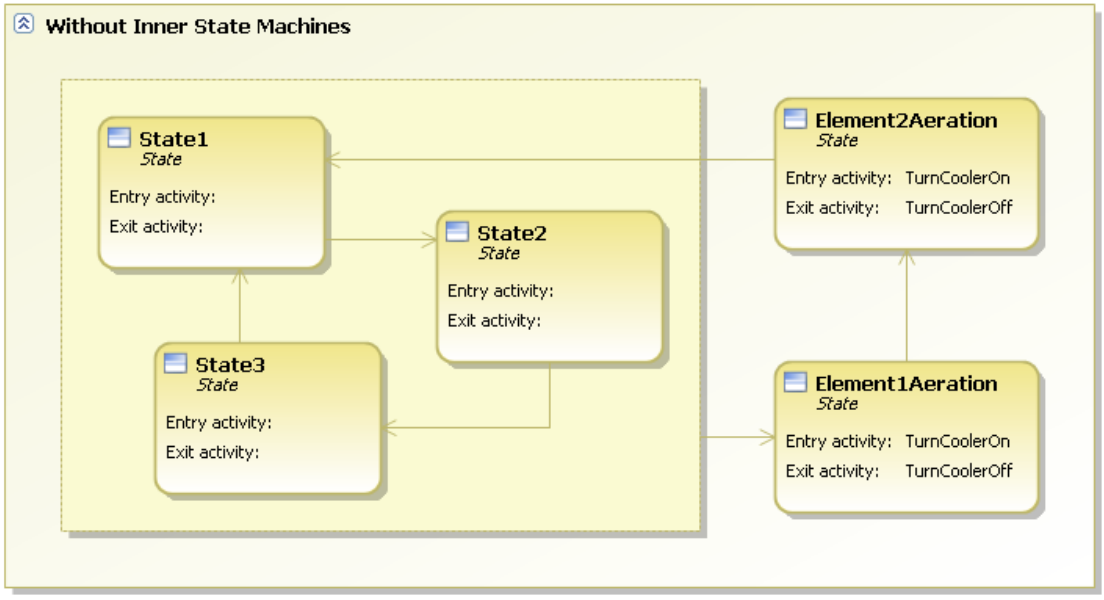

Рис. 9. Схематичное изображение автомата вентилирования двух элементов без применения вложенных автоматов

Состояния *Element1Aeration* и *Element2Aeration* соответствуют последовательностям действий по вентиляции первого и второго элементов соответственно. При отсутствии вложенных автоматов, на диаграмме может быть представлено только последовательное выполнение таких действий.

На рис. 10 изображено решение той же задачи, но с применением вложенных автоматов.

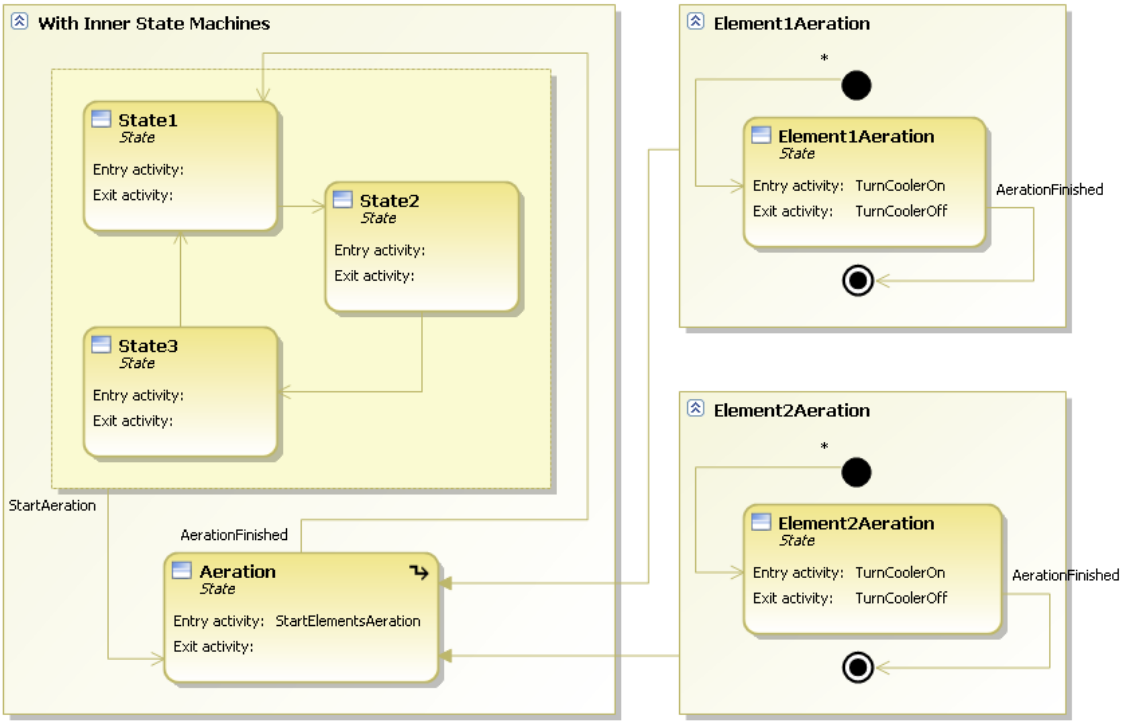

Рис. 10. Схематичное изображение автомата вентилирования двух элементов с использованием вложенных автоматов

Из рисунков заметно, что в случае применения вложенных автоматов, модель системы получилась несколько сложнее. Но несомненным преимуществом такого подхода является то, что в таком случае автоматически достигается процесс одновременного вентилирования нескольких элементов установки.

Таким образом, наличие вложенных автоматов позволяет вводить в модель автоматной системы понятие параллельного выполнения автоматов, которое часто требуется для решения практических задач, особенно задач автоматизации. Также, с помощью вложения одного автомата в несколько состояний, появляется возможность многократного использования одного автомата внутри проектируемой диаграммы.

Класс *State* описываемой библиотеки содержит свойство *InnerStateMachines*, которое является коллекцией объектов типа *StateMachine*. В случае, когда объекты указанного типа добавлены в данную коллекцию, соответствующее состояние имеет вложенные автоматы. Когда автомат

находится в таком состоянии, то входные воздействия могут быть обработаны вложенными автоматами.

4.1.4. Обработка входных воздействий вложенными автоматами

В случае, когда у состояния присутствует один или более вложенных автоматов, встает вопрос, каким автоматом будет обрабатываться следующее входное воздействие: главным автоматом или вложенными автоматами. Данный вопрос не имеет однозначного ответа. Некоторые задачи требуют подхода, при котором главный автомат не может обрабатывать входные воздействия до того момента, пока все вложенные автоматы не будут находиться в конечных состояниях. Другие задачи требуют того, чтобы сначала главный автомат обрабатывал входные воздействия, и только в том случае, когда автомат не имеет требуемого перехода, входное воздействие передавалось вложенным автоматам.

Для решения описанной задачи в класс *StateMachine* было добавлено свойство *EventHandlingOrder,* которое специфицирует способ обработки входных воздействий вложенными автоматами. В зависимости от значения свойства *EventHandlingOrder*, сообщения могут обрабатываться тремя возможными вариантами:

1. *EventHandlingOrder.InnerStateMachineUntilFinished* – входные воздействия обрабатываются вложенными автоматами до тех пор, пока все они не будут находиться в конечном состоянии. В случае многократной вложенности автоматов, маршрутизация сообщений в данном режиме напоминает алгоритм поиска в глубину.
2. *EventHandlingOrder.InnerStateMachineBeforeCurrent* – входное воздействие будет обработано вложенными автоматами и только в том случае, если ни один автомат не совершил переход, воздействие будет обработано главным автоматом.

3. *EventHandlingOrder.CurrentStateMachineBeforeInner* – способ обработки, противоположный предыдущему. Если входное воздействие может быть обработано главным автоматом, прерывается выполнение вложенных автоматов, выполняется соответствующий переход главного автомата. Если входное воздействие не может быть обработано главным автоматом, оно передается вложенным автоматам.

4.1.5. Синхронный и асинхронный режимы работы

Еще одним важным моментом при разработке реальных приложений является режим обработки входящих воздействий классом автомата. Широко распространены два метода обработки сообщений при работе с событийными системами: синхронный и асинхронный. При синхронном методе обработки, автомат выполняет действия в том же потоке, который передал входное воздействие. Таким образом, пока автомат не завершит обработку поступившего входного воздействия и не выполнит необходимые действия, поток, вызвавший обработку входного воздействия, будет ждать. Такая модель поведения необходима в случае, если дальнейшее поведение приложения зависит от принятого автоматом решения. В случае же, когда время выполнения того или иного действия не принципиально, возможно использование асинхронного метода обработки сообщений. При таком подходе к обработке сообщений, входное воздействие передается в дополнительный поток автомата, а основной поток продолжает свою деятельность. Дополнительный поток автомата при этом выполняет функции менеджера входных воздействий: он содержит очередь необработанных входных воздействий и обрабатывает их по порядку. При любом режиме обработки входных воздействий автоматом, методы *Run, HandleEvent* и *Terminate* являются потокобезопасными. Последнее обеспечено стандартными средствами встроенной библиотеки классов *Microsoft Visual Studio .Net Framework.*

Для поддержки описанных методов обработки входных воздействий в класс автомата *StateMachine* добавлено свойство *EventHandlingMode*. Данное свойство может принимать значения *EventHandlingMode.Synchronous* и *EventHandlingMode.Asynchronous*, при которых автомат будет обрабатывать сообщения в синхронном и асинхронном режимах соответственно.

4.1.6. Дальнейшее развитие *FSMLib*

Особенным преимуществом существования классов сущностей автомата является то, что использование такой библиотеки не требует от разработчика специфических знаний паттернов разработки программного обеспечения, структур данных и прочего. Разработка конечного автомата с использованием классов *FSMLib* ничем не отличается от разработки любого приложения с использованием классов какой-либо библиотеки.

Возможными улучшениями библиотеки могут стать введение дополнительного компонента, осуществляющего протоколирование действий автомата, расширение событийной модели автомата (то есть введение сообщений на каждое действие, совершаемое автоматом), а также добавление механизма верификации автоматных систем, реализуемых с использованием рассматриваемой библиотеки.

4.2. Инструментальное средство

В работе [3] было разработано инструментальное средство для среды разработки *Microsoft Visual Studio 2005*. Графические редакторы в работы реализовывались с помощью компонента *Инструментов Специализированных Языков Предметной Области* (*Domain-Specific Language Tools*) [18], входящих в *Microsoft Visual Studio SDK*. Этот компонент позволяет создание собственных визуальных редакторов, приспособленных для конкретной области. Также благодаря этому компоненту, разработанный редактор встраивается в *Microsoft Visual Studio* и становится редактором по умолчанию заданных типов файлов,

соответствующих заданным диаграммам. *Microsoft Visual Studio SDK 2008* содержит обновленную версию инструментов *DSL Tools*.

Для создания собственного визуального редактора, необходимо описание его метамодели. При преобразовании встроенных в *DSL Tools* текстовых шаблонов, генерируются классы, соответствующие классам данных и классов фигур, заданных на метамодели. Метамодель содержит механизмы обеспечения некоторых правил размещения фигур на диаграмме, а также правила их добавления, удаления или указания связей между фигурами. Но, несмотря на богатую функциональность компонента, часть кода, отвечающая за валидацию диаграммы, а также за нестандартное поведение фигур на диаграмме, должна быть написана вручную.

По разработанной метамодели также возможна генерация текстов. В данной работе метамодель описывает модель конечного автомата, поэтому по такой модели возможна генерация исходного кода на любом из высокоуровневых языков программирования. Так как описываемый в работе инструмент реализован на языке *C#*, разработан текстовый шаблон, который генерирует код на том же языке. Код автомата, который получается при трансформации разработанного текстового шаблона, использует библиотеку *FSMLib*.

Рассмотрим подробнее построенную метамодель диаграммы автомата, а также разработанные правила валидации и механизм генерации исходного кода.

4.2.1. Метамодель диаграммы

На рис. 11 представлен вид окна среды разработки *Microsoft Visual Studio 2008* во время проектирования метамодели диаграммы автомата.

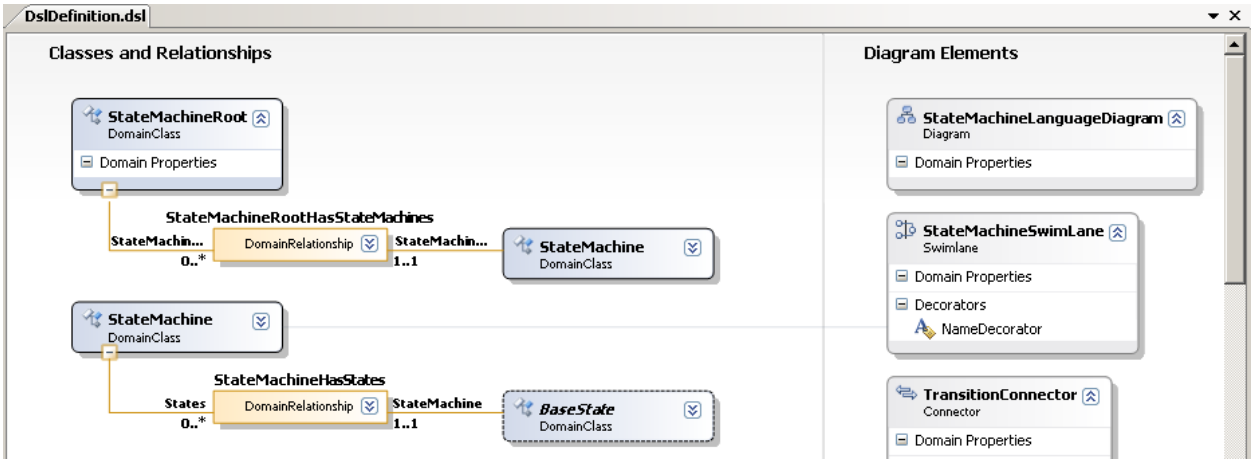

Рис. 11. Редактирование метамодели диаграммы классов в *Microsoft Visual Studio 2008*

Как можно заметить из рисунка, диаграмма состоит из двух вертикальных частей: области определения сущностей (классов) и их связей и области определения графических элементов диаграммы. На этой же диаграмме осуществляется связь между классами сущностей и графическим представлением класса. В случае, когда классу необходимо добавить какую-либо логику, например, вычислять значение свойства в зависимости от других свойств, такие изменения вносятся вручную в соответствующий класс диаграммы.

На рис. 12–16 представлена метамодель разработанной диаграммы.

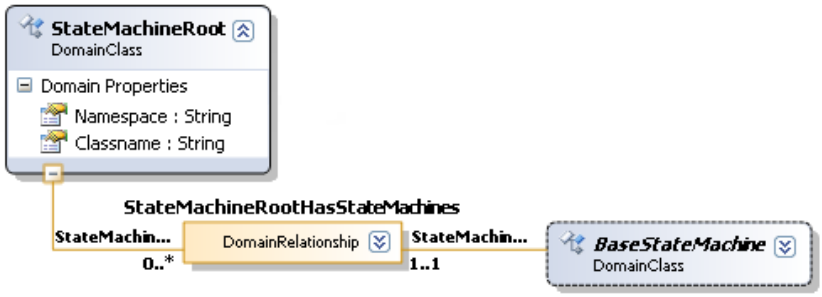

Рис. 12 Отношение классов *StateMachineRoot* и *BaseStateMachine*

На рис.12 изображен корневой элемент диаграммы *StateMachineRoot*, который соответствует «чистому листу», на который могут быть добавлены элементы диаграммы. Отношением композиции *StateMachineRootHasStateMachines* корневой элемент связан с элементом *BaseStateMachine*. Последний соответствует базовому классу для двух типов автоматов, которые могут быть представлены на диаграмме полученного

редактора. Заметим, что на фигуре отношения проставлена кратность. В данном случае, обозначение «0..*» слева от фигуры отношения означает, что диаграмма может содержать от нуля и больше конечных автоматов, а «1..1» означает, что конечный автомат может принадлежать только одному объекту класса *StateMachineRoot*. Класс *StateMachineRoot* содержит свойства *Namespace* и *Classname*, задающие полный путь до класса, в который должно быть добавлено автоматное поведение.

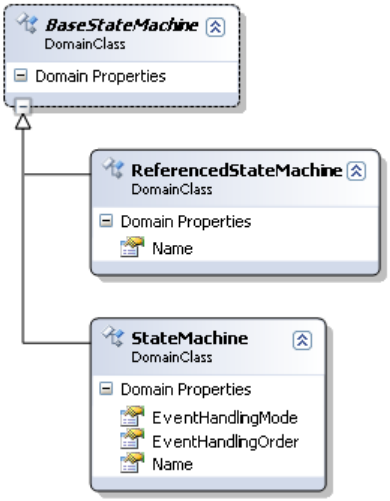

Рис. 13. Отношение классов *BaseStateMachine, StateMachine* и *ReferencedStateMachine*

На рис. 13 представлено отношение наследования классов *StateMachine* и *ReferencedStateMachine* от класса *BaseStateMachine*. Класс *StateMachine* представляет конечный автомат, конструируемый на разрабатываемой диаграмме. Класс *ReferencedStateMachine* представляет автомат, который реализован вне разрабатываемой диаграммы. Класс *ReferencedStateMachine* введен для того, чтобы обеспечить возможность повторного использования автоматов в различных частях приложения без повторного моделирования таких автоматов на нескольких диаграммах. Оба класса автоматов имеют свойство *Name*, задающее имя автомата. Класс *StateMachine* также содержит свойства *EventHandlingMode* и *EventHandlingOrder*, задающие режим обработки автоматом входных воздействий и порядок обработки входных воздействий вложенными автоматами соответственно.

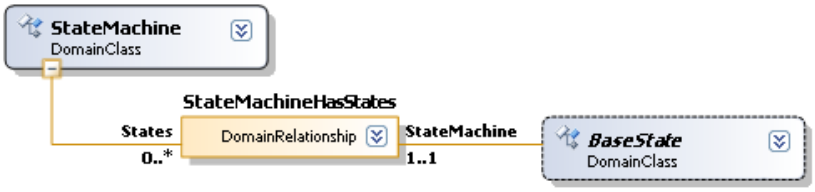

Рис. 14. Отношение классов *StateMachine* и *BaseState*

На рис. 14 представлены классы автомата *StateMachine* и базовый класс состояния *BaseState*. Отношение композиции в данном случае имеет ту же кратность, что и на рис. 12 и означает, что автомат может иметь от нуля и больше состояний, а состояние может принадлежать только одному автомату.

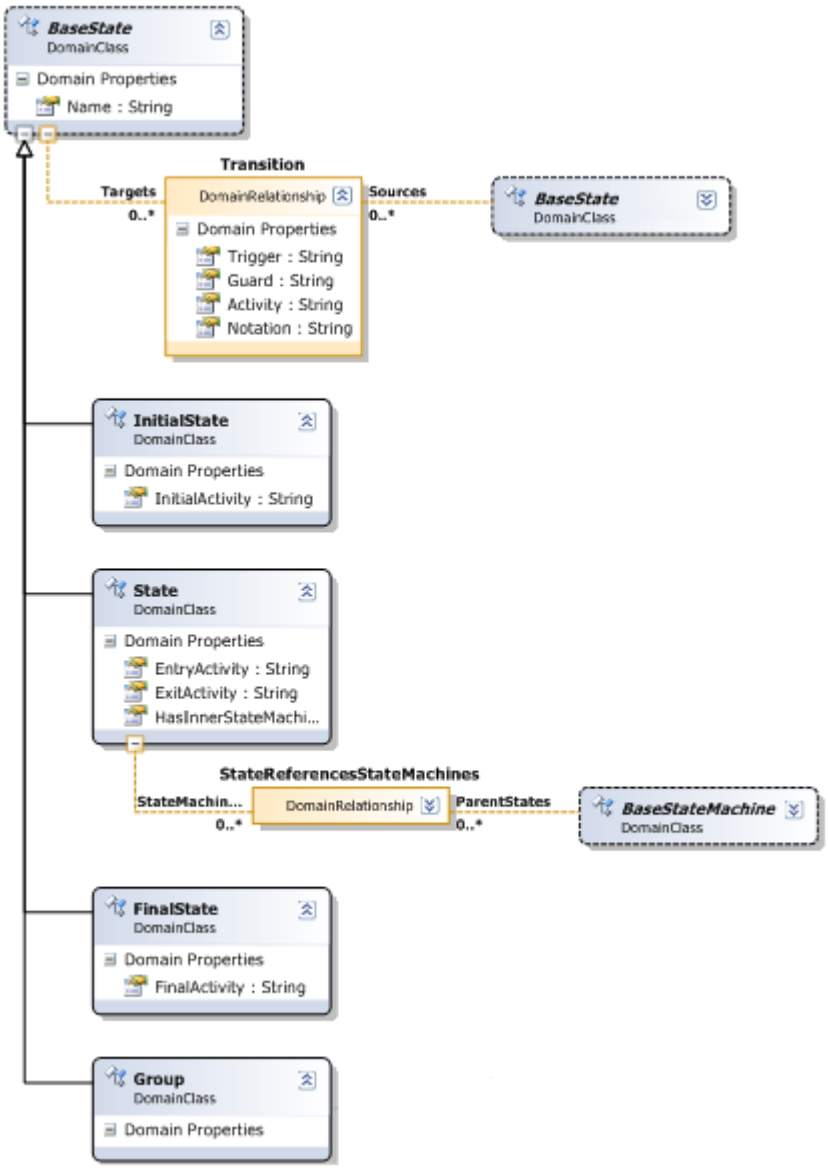

Рис. 15. Метамодель диаграммы автомата. Наследники класса *BaseState*

На рис. 15 представлена ветвь наследников класса *BaseState*, то есть классы, соответствующие различным состояниям автомата. Классы *InitialState*, *State*, *FinalState* представляют начальное состояние, обычное состояние и конечное состояние соответственно. Для начального состояния может быть установлено инициализирующее автомат действие *InitialActivity*, для обычного состояния действие на входе в состояние *EntryActivity* и на выходе *ExitActivity*, конечному состоянию только завершающее действие *FinalActivity*. Заметим, что класс *State* связан отношением зависимости *StateReferencesStateMachines* с классом *BaseStateMachine*. Кратность отношения «0..*» для класса *State* и «0..*» для класса *StateMachine*. Это отношение означает, что состояние может ссылаться на ноль и более вложенных автоматов, а каждый вложенный автомат может иметь от нуля и больше состояний, которые на него ссылаются. На метамодели присутствует также отношение зависимости *Transition* между двумя классами *BaseState*. Данное отношение реализует класс переходов между состояниями. Класс *Transition* содержит следующие свойства:

1. Описание входного воздействия *Trigger*, по которому будет осуществляться переход.
2. Описание условия *Guard*, которое должно быть выполнено для перехода.
3. Действие *Activity*, которое будет совершено во время перехода.
4. Свойство *Notation*, отображающее полную информацию о переходе. Данное свойство вычисляется на основе значений первых трех свойств.

Класс *Group* на метамодели представляет собой группу состояний. Отношение для класса *Group* изображено на рис. 16.

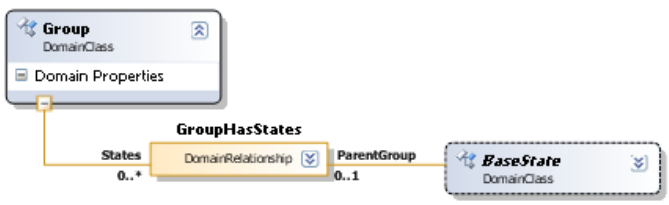

Рис. 16. Метамодель диаграммы автомата. Отношение для класса *Group*

Отношение композиции *GroupHasState* означает, что объект класса *Group* может содержать от нуля и больше состояний, а у объекта класс *BaseState* может быть не более одного объекта родительской группы *ParentGroup*. Благодаря данному отношению возможно группирование состояний автомата. Назначение группы состояний описано выше в разделе 4.1.2. С другой стороны, так как класс *Group* является наследником класса *BaseState*, то возможно создание переходов *Transition* из группы в любое состояние. Вообще говоря, метамодель никак не запрещает и обратный переход из заданного состояния в группу, или из группы в группу. Но, благодаря разработке дополнительных правил для редактирования переходов, создать некорректную конструкцию, с точки зрения модели конечного автомата, просто невозможно.

Заметим, что приведенная метамодель отличается от метамодели для диаграммы автомата, описанной в работе [3]. Изменения обусловлены тем, что в данной работе поддержано проектирование вложенных автоматов, а также введено понятие группы состояний.

Несмотря на то, что введение правил для метамодели исключает возможность создания неверных конструкций, например создание перехода из конечного состояния в начальное состояние, полученный редактор все еще допускает создание моделей, для которых не возможна автоматическая генерация правильного исполняемого кода конечного автомата. Чтобы решить данную проблему, разработаны методы валидации диаграммы.

4.2.2. Валидация диаграммы

Для гарантии корректности разработанной модели и возможности автоматической генерации правильного исполняемого кода конечного автомата, должны быть выполнены следующие условия:

1. Автомат должен иметь единственное начальное состояние.

2. Автомат должен содержать хотя бы одно конечное состояние (в случае невыполнения данного условия, разработчику будет выдано лишь предупреждение, а не ошибка).
3. Все переходы должны происходить по непустому событию.
4. Все состояния должны быть достижимы из начального состояния.
5. Из любого состояния автомата должно быть достижимо хотя бы одно конечное состояние (в случае невыполнения данного условия, разработчику будет выдано лишь предупреждение, а не ошибка).
6. Все состояния должны иметь уникальные имена.
7. Все группы состояний должны иметь уникальные имена.
8. Каждая группа состояний автомата должна содержать хотя бы одно состояние.
9. Для автомата должны быть определены непустые пространство имен *Namespace* и имя класса *Classname*.
10. Все ссылочные автоматы (*ReferencedStateMachine*) должны иметь уникальные имена.

При каждом открытии, сохранении диаграммы или попытке генерации исходного кода по диаграмме, в случае когда не выполнено какое-либо из вышеописанных условий, соответствующее сообщение появляется в стандартном окне ошибок среды разработки *Visual Studio*.

4.2.3. Генерация исходного кода

Генерация исходного кода конечного автомата по модели производится с помощью трансформации разработанного текстового шаблона. Технология трансформации шаблонов в тексты появляется также после установки пакета *Microsoft Visual Studio SDK*. Для любой разрабатываемой с помощью *DSL Tools* диаграммы, возможно создание шаблона для генерации текста по метамодели данной диаграммы. В данной работе разработан шаблон для генерации исходного кода конечного автомата на языке *C#* с использованием библиотеки *FSMLib* (приложение 2).

Генерация кода автомата по диаграмме происходит во время открытия, сохранения модели или по принудительному запуску генерации кода пользователем. Перед началом генерации, производится валидация диаграммы и в случае, когда какое-либо из необходимых условий не выполнено, разработчику выдается сообщение о том, что модель содержит ошибки. Далее разработчик может сгенерировать исходный код по разработанной модели, несмотря на присутствующие ошибки, либо отказаться от генерации и попытаться их исправить.

ВЫВОДЫ ПО ГЛАВЕ 4

В главе 4 разработана библиотека классов для реализации сложных систем конечных автоматов. Особое внимание было уделено механизму взаимодействия главного и вложенных автоматов, а также способу обработки входных воздействий. Разработанная библиотека позволяет проектировать сложные автоматные системы, в том числе системы, в которых несколько автоматов могут работать в различных потоках. Благодаря использованию данной библиотеки, существенно упрощается исходный код, который должен быть написан разработчиком вручную.

Также в главе приведен способ реализации инструментального средства для поддержки автоматного программирования в среде разработки *Microsoft Visual Studio 2008*. Предложено использование компонента *Domain-Specific Language Tools*, входящего в *Microsoft Visual Studio 2008 SDK*, для реализации собственного визуального редактора для проектирования моделей конечных автоматов. Описанные компоненты и подходы позволяют создать инструментальное средство, которое обладает многими преимуществами по сравнению с аналогичными продуктами.

5. ОПИСАНИЕ РАЗРАБОТАННОГО ИНСТРУМЕНТАЛЬНОГО СРЕДСТВА

5.1. Установка инструментального средства

Разработанное инструментальное средство поставляется в качестве инсталляционного файла, который может быть выполнен без помощи сторонних программ на любом компьютере с установленной операционной системой *Microsoft Windows Vista* и средой разработки программного обеспечения *Microsoft Visual Studio 2008*. После инсталляции продукта, необходимые расширения файлов, будут зарегистрированы в системе. Возможность создания и редактирования диаграммы конечных автоматов в среде разработки *Microsoft Visual Studio 2008* появится автоматически.

5.2. Создание диаграммы автомата

Для создания диаграммы автомата, в проекте среды разработки *Visual Studio 2008* должен быть добавлен элемент с расширением «*.stateMachine*». Добавление может быть осуществлено с использованием стандартного меню среды разработки (рис. 17).

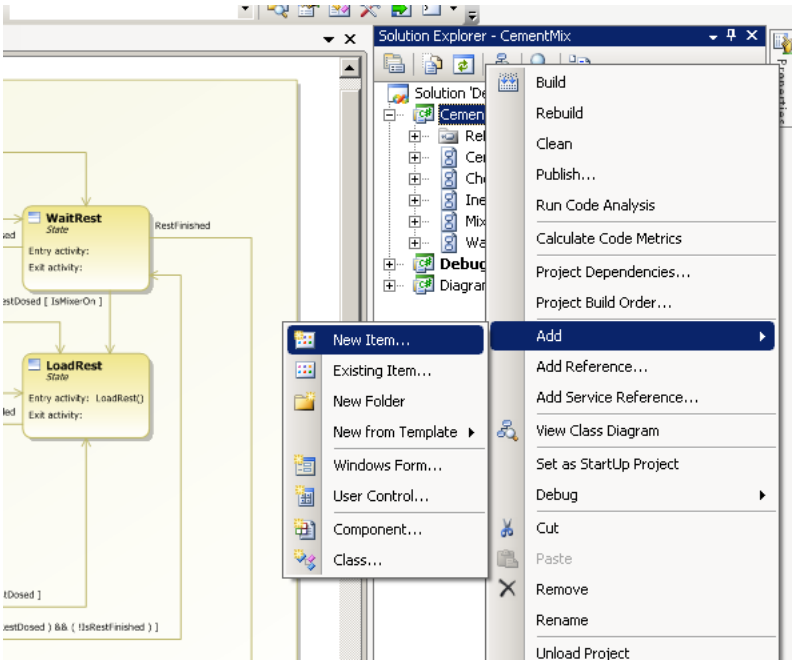

Рис. 17. Стандартное меню *Visual Studio* для добавления элемента в проект

Затем в появившемся диалоге следует выбрать элемент типа *StateMachineLanguage* (рис. 18).

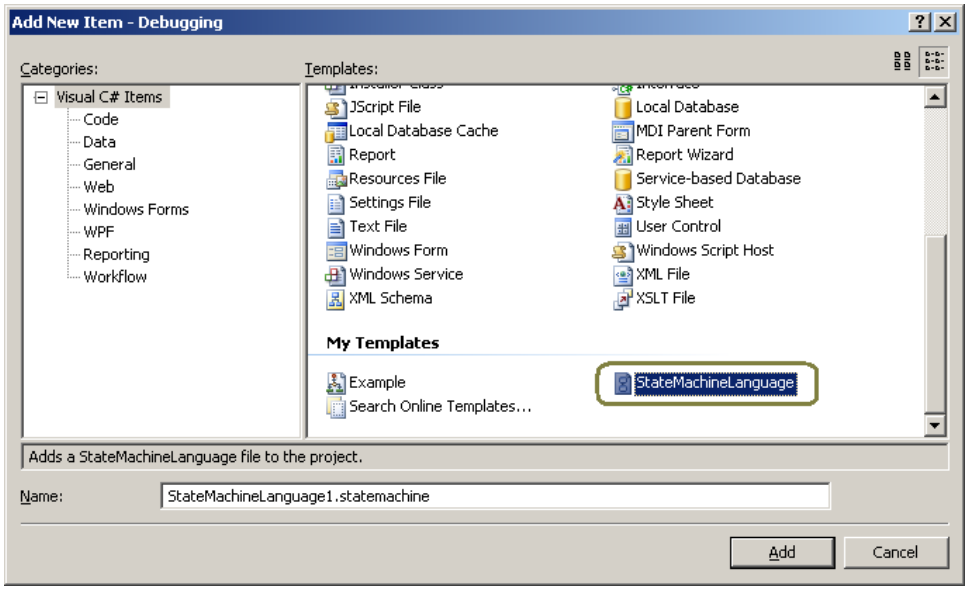

Рис. 18. Добавление диаграммы конечного автомата в проект

После добавления данного элемента в проект, он будет автоматически открываться с помощью разработанного графического редактора.

5.3. Редактирование диаграммы автомата

На рис. 19 изображена среда разработки *Microsoft Visual Studio 2008* при редактировании диаграммы автомата.

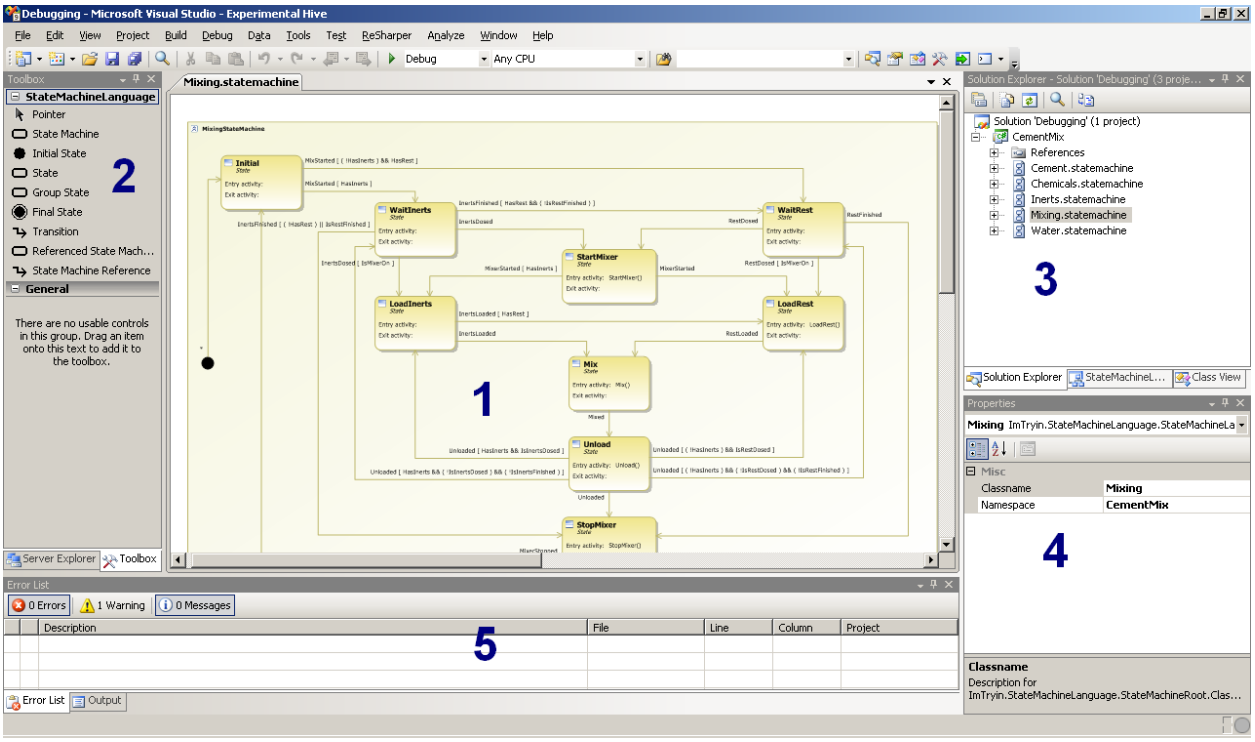

Рис. 19. Редактирование диаграммы конечного автомата в *Visual Studio 2008*

Цифрами на рисунке отмечены следующие области:

1. Окно диаграммы автомата. Добавление, удаление, а также редактирования положения и размеров элементов модели автомата происходит в данном окне.
2. Инструментарий разработанного редактора содержит восемь инструментов для редактирования автомата. Инструмент *Initial State* служит для создания начальных состояний на диаграмме. Инструмент *Group State* для создания групп состояний. Инструмент *State* для создания обычных состояний. Инструмент *Final State* для создания конечных состояний. Инструменты *Transition* и *State Machine Reference* для создания переходов между состояниями и связи между состоянием и вложенным автоматом соответственно. Инструменты *State Machine* и *Referenced State Machine* позволяют добавлять на диаграмму автоматы и ссылки на автоматы, которые позже будут инициализированы разработчиком.
3. Стандартное для среды разработки *Microsoft Visual Studio* дерево проекта. Элементы типа *StateMachineLanguage* так же, как и основные элементы проектов *Visual Studio,* добавляются в это дерево.
4. Стандартное окно свойств объекта. При редактировании диаграммы автоматов, данное окно использования для задания свойств состояниям, переходам и т.д.
5. Стандартное окно ошибок. При редактировании диаграммы автоматов, все ошибки валидации разрабатываемой модели отображаются в этом окне.

5.4. Генерируемый код

С помощью реализованного инструментального средства, весь исходный код, отвечающий за логику работы конечных автоматов,

генерируется автоматически. Полученный код использует библиотеку классов *FSMLib* для реализации систем конечных автоматов. Каждый автомат на диаграмме преобразуется в свойство языка *C#* типа *StateMachine* и с именем, которое указано на диаграмме в качестве имени автомата. В разделе 4.2.1. описаны свойства диаграммы *Namespace* и *Classname*. При генерации кода значения данных свойств диаграммы задают путь к классу, к которому будет добавлено свойство, представляющее разработанный автомат.

На рис 20 приведен пример автомата, реализованного с помощью разработанного инструментального средства.

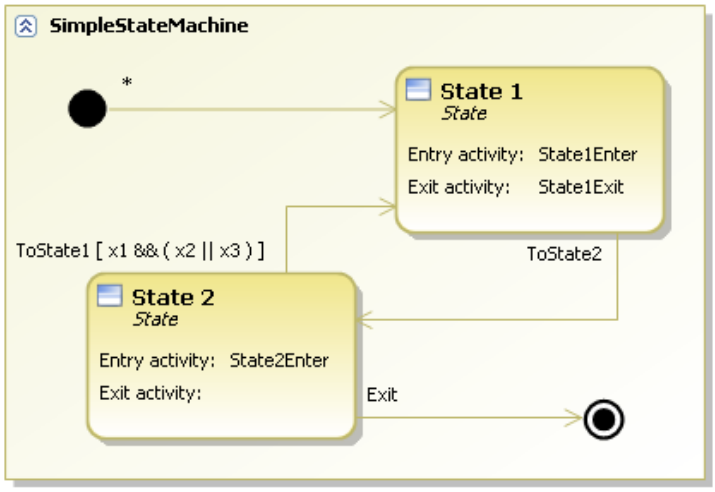

Рис. 20. Конечный автомат, спроектированный в разработанном инструментальном средстве

Данный автомат имеет четыре состояния. После запуска автомата, он сразу же переходит в состояние *State1*. Далее, по входным воздействиям *ToState1* и *ToState2* осуществляются переходы в состояния *State1* и *State2* соответственно. Состояние *State1* имеет действия при входе в состояние и выходе из состояния. Состояние *State2* имеет действие только при входе в состояние. На переходе из состояния *State2* в *State1* присутствует условие (x1 && (x2 || x3)), выполнение которого необходимо для осуществления перехода. Если автомат находится в состоянии *State2*, то по входному воздействию *Exit* автомат завершает свою работу. Свойствам *Namespace* и *Classname* диаграммы были присвоены значения *StateMachine.Samples*

SimpleStateMachineContext соответственно, поэтому автомат будет встроен в класс *StateMachine.Samples.SimpleStateMachineContext*. Ниже приведен исходный код, который был сгенерирован по заданному автомату:

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

using FSM.Core.Model;

namespace StateMachines.Samples
{
    partial class SimpleStateMachineContext
    {
        #region SimpleStateMachine

        #region Initialization

        private StateMachine _simpleStateMachine;
        public StateMachine SimpleStateMachine
        {
            get
            {
                if ( _simpleStateMachine != null )
                    return _simpleStateMachine;
                //
                //InitialState1
                //
                InitialState InitialState1 = new InitialState();
                InitialState1.Name = "InitialState1";
                //
                //State 1
                //
                State State1 = new State();
                State1.Name = "State 1";
                State1.EntryAction = OnState1Enter;
                State1.ExitAction = OnState1Exit;
                //
                //State 2
                //
                State State2 = new State();
                State2.Name = "State 2";
                State2.EntryAction = OnState2Enter;
                //
                //FinalState1
                //
                FinalState FinalState1 = new FinalState();
                FinalState1.Name = "FinalState1";
                //
                //InitialState1->State 1
                //
                Transition transition1 = new Transition();
                transition1.Source = InitialState1;
                transition1.Target = State1;
                transition1.Event = "*";
                //
                //State 1->State 2
                //
                Transition transition2 = new Transition();
                transition2.Source = State1;
                transition2.Target = State2;
                transition2.Event = "ToState2";
                //
```

```
                    //State 2->FinalState1
                    //
                    Transition transition3 = new Transition();
                    transition3.Source = State2;
                    transition3.Target = FinalState1;
                    transition3.Event = "Exit";
                    //
                    //State 2->State 1
                    //
                    Transition transition4 = new Transition();
                    transition4.Source = State2;
                    transition4.Target = State1;
                    transition4.Event = "ToState1";
                    transition4.Guard = IsFromState2ToState1ConditionSatisfied;
                    //
                    //SimpleStateMachine
                    //
                    _simpleStateMachine = new StateMachine();
                    _simpleStateMachine.Name = "SimpleStateMachine";
                    _simpleStateMachine.EventHandlingMode                              =
EventHandlingMode.Synchronous;
                    _simpleStateMachine.EventHandlingOrder                             =
EventHandlingOrder.InnerStateMachineUntilFinished;
                    _simpleStateMachine.InitialState = InitialState1;
                    _simpleStateMachine.States.Add( State1 );
                    _simpleStateMachine.States.Add( State2 );
                    _simpleStateMachine.FinalStates.Add( FinalState1 );
                    _simpleStateMachine.Transitions.Add( transition1 );
                    _simpleStateMachine.Transitions.Add( transition2 );
                    _simpleStateMachine.Transitions.Add( transition3 );
                    _simpleStateMachine.Transitions.Add( transition4 );
                    return _simpleStateMachine;
                }
            }

            #endregion

            #region Actions

            protected virtual void OnState1Enter()
            {
                State1Enter;
            }

            protected virtual void OnState1Exit()
            {
                State1Exit;
            }

            protected virtual void OnState2Enter()
            {
                State2Enter;
            }

            #endregion

            #region Predicates

            protected virtual bool IsFromState2ToState1ConditionSatisfied()
            {
                return x1 && ( x2 || x3 );
            }

            #endregion

            #endregion

        }
    }
```

Для того, чтобы запустить исполнение автомата, разработчик должен будет вызвать метод *Run* у объекта, на который указывает свойство *SimpleStateMachine*.

5.5. Реализация выходных воздействий и условий переходов

Рассмотрим рис. 20. На диаграмме автомата указаны действия при входе и выходе из состояний *State1Enter*, *State2Enter*, *State1Exit*, а также условие на переходе (x1 && (x2 || x3)). По сгенерированному коду предыдущего раздела, можно увидеть, что для всех действий и для условия на переходе были созданы обработчики, которые вызывают указанные методы или проверяют значения заданных переменных. Эти методы и переменные должны присутствовать в классе, частью которого является разработанный автомат, и должны быть реализованы вручную разработчиком.

Чаще всего автомат проектируется как часть класса, который уже содержит все необходимые переменные и действия. В таком случае, после проектирования диаграммы автомата и генерации исходного кода, реализация каких-либо дополнительных действий или переменных не требуется.

ВЫВОДЫ ПО ГЛАВЕ 5

Разработанное инструментальное средство является удобным и полезным средством для разработки программ с использованием автоматного подхода в среде разработки *Microsoft Visual Studio 2008*. Данное средство позволяет разработку программ с применением автоматов, не углубляясь при этом в специфику работы конечных автоматов и их взаимодействия. Единственная часть автомата, которая должна быть реализована разработчиком – это действия, выполняемые автоматом на переходах, а также при входе и выходе из состояния, а также переменные, значения которых проверяются автоматом для осуществления переходов.

Разработанное средство позволяет существенно уменьшить количество исходного кода, а также время на разработку приложений, в которых оправдано использование автоматного подхода к программированию.

6. АПРОБАЦИЯ РАЗРАБОТАННОГО ИНСТРУМЕНТАЛЬНОГО СРЕДСТВА

Представленное в работе инструментальное средство применено для задачи автоматизации бетонного завода. Применение было осуществлено в рамках проекта Санкт-Петербургской компании по разработке программного обеспечения *NordWestSoft* [19] для автоматизации работы бетонного завода.

Стоит отметить, что с помощью конечных автоматов в данной работе реализована только логика приложения, отвечающего за управление механическими частями установки бетонного завода. Низкоуровневое взаимодействие с контроллерами установки реализовано на языке *C++*. Графический пользовательский интерфейс приложения реализован на языке *C#* с помощью стандартной библиотеки *Microsoft .NET Framework Windows.Forms* без применения автоматов.

В следующих разделах более подробно описан технологический процесс приготовления раствора бетона. В описании используются названия входных воздействий автоматов, приводящих к осуществлению переходов, а также некоторые свойства и методы класса данных, доступных автомату. Также в разделах приводятся диаграммы автоматов, использованных при реализации процесса производства раствора бетона.

6.1. Процесс смешивания бетона, основные положения

На рис. 21 представлена диаграмма автомата, реализующего логику процесса смешивания материалов для приготовления раствора бетона.

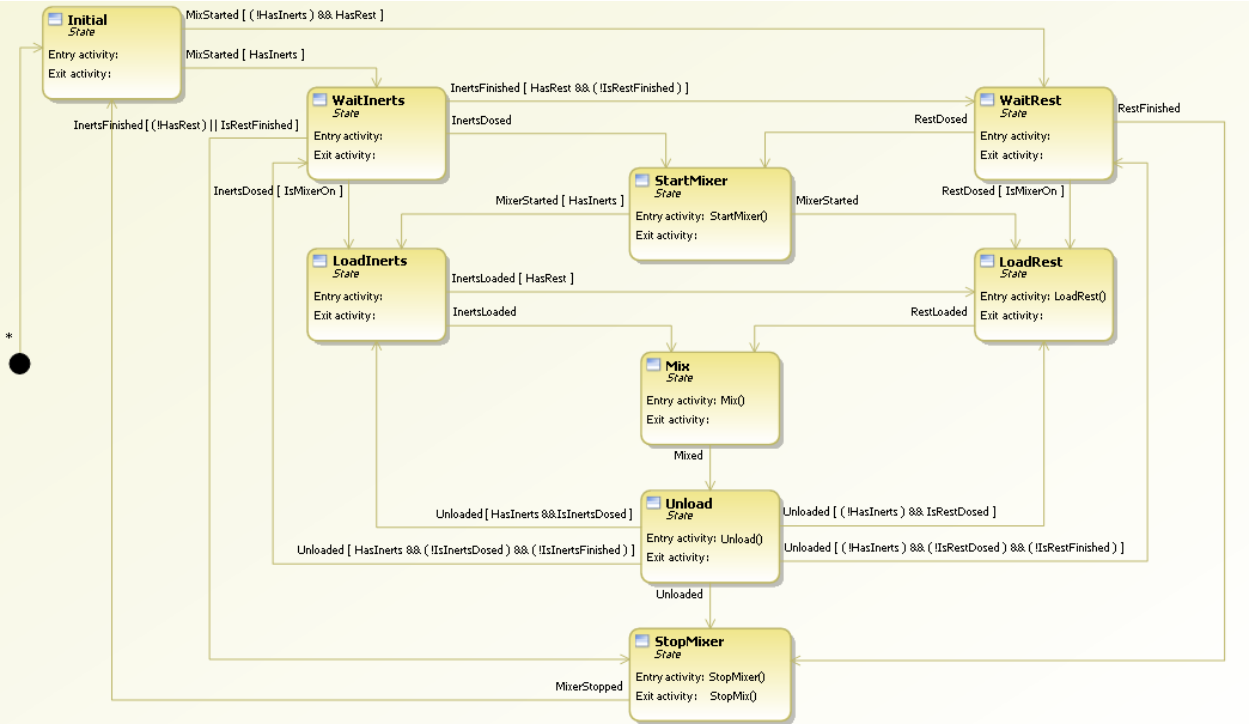

Рис. 21. Диаграмма автомата смешивания материалов раствора

Смешивание материалов для раствора бетона начинается при обработке входного воздействия *MixStarted*. Процедура смешивания производится в несколько циклов. Каждый цикл состоит из дозировки материалов, загрузки их в смеситель, смешивания и выгрузки бетона. Для приготовления раствора бетона используются различные метериалы: инертные (щебень и песок), химические добавки, а также вода и цемент. В рецепте раствора может отсутствовать тот или иной материал. Для определения присутствия материала в рецепте используются свойства *HasInerts*, *HasChemicals*, *HasWater*, *HasCement*, означающие наличие в рецепте инертных материалов, химических добавок, воды и цемента соответственно. Свойство *MixFinishing* означает отсутствие следующего цикла. По окончанию смешивания вызывается метод *StopMix*.

6.2. Дозировка

Дозировка каждого материала, участвующего в приготовлении раствора, должна происходить независимо от дозировки других материалов, так как в противном случае процесс дозировки необходимых ингредиентов будет занимать слишком много времени. Требуемым условием также

является еще то, что дозировка одного материала для следующего цикла смешивания должна начинаться сразу же после выгрузки этого материала.

Дозировка инертных материалов начинается после вызова метода *DoseInerts*, а по окончании генерирует входное воздействие *InertsDosed*. Аналогичным образом происходит начало и окончание дозировки химических добавок, воды и цемента. За начало и окончание их дозировки отвечают методы *DoseChemicals*, *DoseWater*, *DoseCement* и генерируемые входные *ChemicalsDosed*, *WaterDosed* и *CementDosed*. Химических добавок может быть до трех штук, поэтому для некоторых рецептов, в процессе дозировки химических веществ, одновременно задействовано несколько насосов. Цемент дозируется через одну или сразу через две трубы. Так как запуск насосов или начало подачи материала по трубе можно считать мгновенным событием, использование сразу нескольких механизмов подачи материалов осуществляется последовательным включением каждого из них в методах *DoseChemicals* и *DoseCement*.

6.3. Загрузка

Загрузка материалов производится только во включенном состоянии и закрытом затворе смесителя. Загрузка происходит следующим образом: сначала происходит загрузка инертных материалов (если они присутствуют в рецепте), после чего происходит загрузка остальных материалов. Неинертные материалы загружаются независимо друг от друга. Загрузка инертных материалов происходит после вызова метода *LoadInerts* и заканчивается событием *InertsLoaded*. Загрузка химических добавок, воды и цемента осуществляются вызовами методов *LoadChemicals*, *LoadWater*, *LoadCement* и заканчивается генерацией входных воздействия *ChemicalsLoaded*, *WaterLoaded* и *CementLoaded* соответственно.

На рис. 22–24 представлены диаграммы автоматов для дозирования и загрузки химических добавок, воды и цемента.

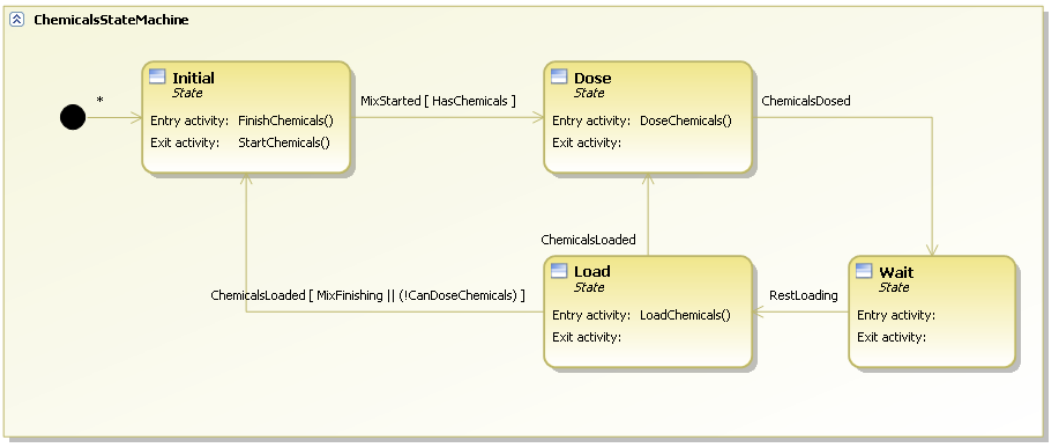

Рис. 22. Диаграмма автомата для дозирования и загрузки химических добавок

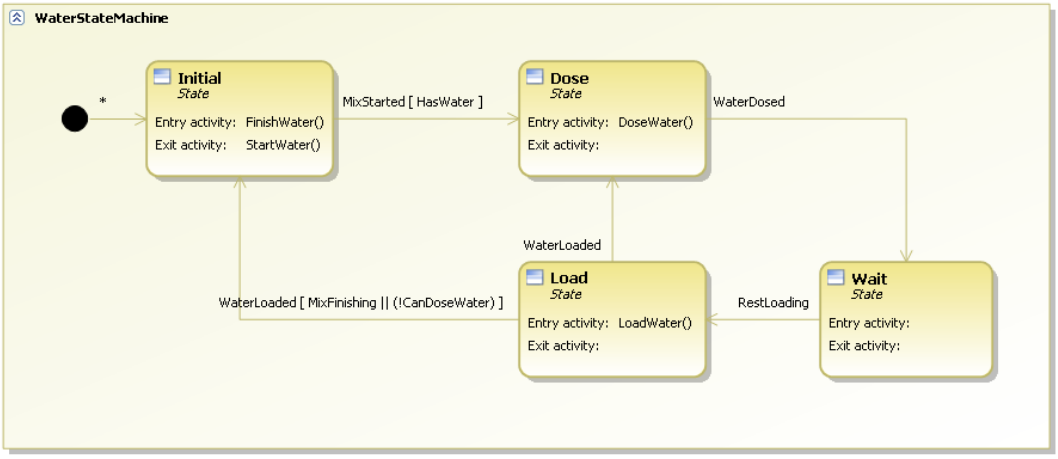

Рис. 23. Диаграмма автомата для дозирования и загрузки воды

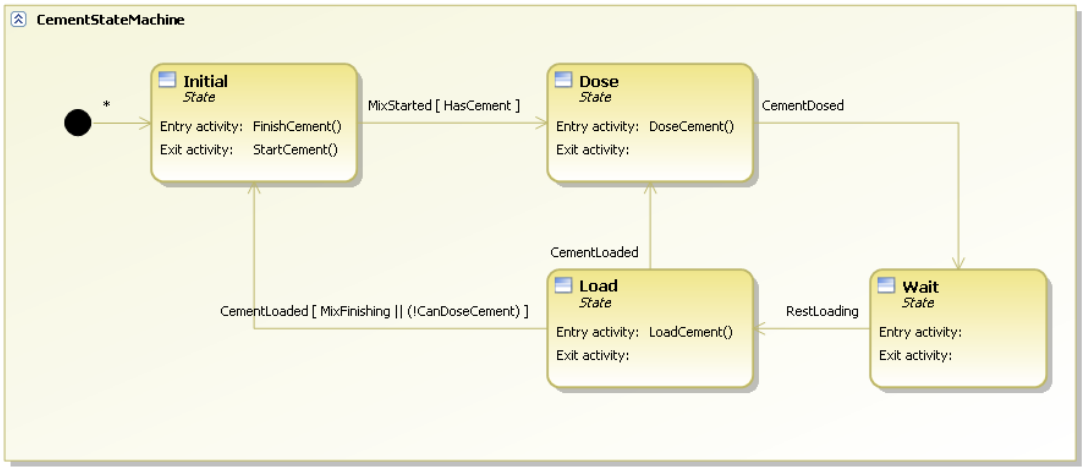

Рис. 24. Диаграмма автомата для дозирования и загрузки цемента

6.4. Управление инертными материалами

На рис. 25 представлена диаграмма автомата, управляющего дозированием и загрузкой инертных материалов.

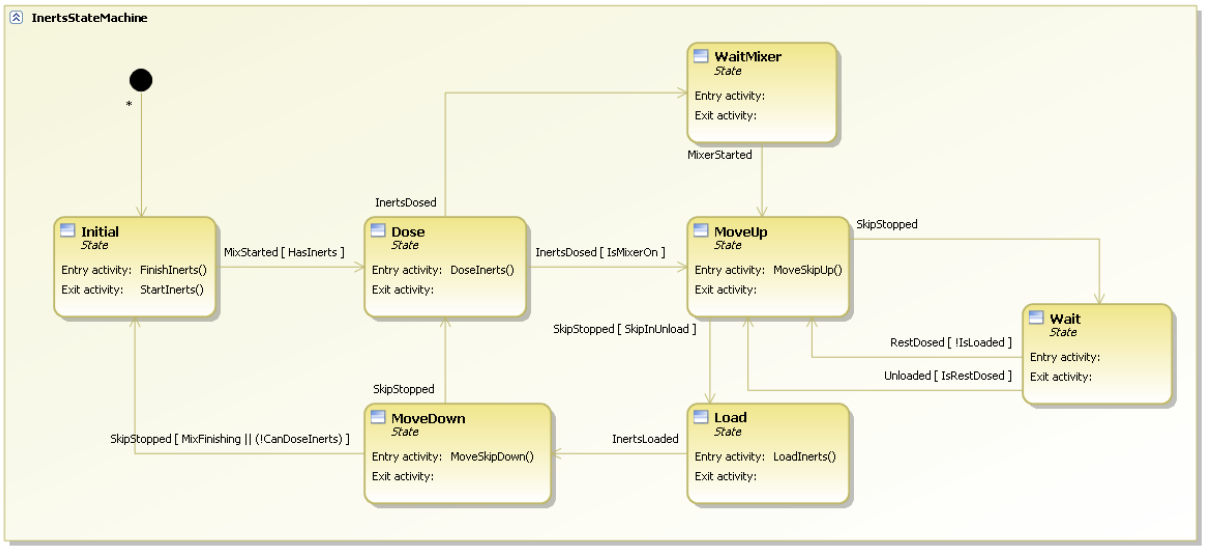

Рис. 25. Диаграмма автомата дозирования и загрузки инертных материалов

Инертные материалы дозируются в скиповый подъемник. В связи с этим, кроме состояний, отражающих части основного процесса, присутствуют методы управления скиповым подъемником (*MoveSkipUp* и *MoveSkipDown* для движения скипа вверх и вниз соответственно) и состояния, отражающие процесс движения подъемника (*MoveUp* и *MoveDown*).

При движении скипового подъемника вверх, он автоматически останавливается в положении ожидания при дозировке любого другого материала, а также при условии загруженности смесителя, при выключенном смесителе или при открытом затворе смесителя. Проверить, что скиповый подъемник остановился раньше положения выгрузки или в самом положении выгрузки, можно с помощью свойства *SkipInUnload*.

При движении скипового подъемника вверх при достижении положения ожидания или выгрузки, подъемник останавливается и генерируется входное воздействие *SkipStopped*. Аналогично, при движении скипового подъемника вниз при достижении положения загрузки, он останавливается и генерируется входное воздействие *SkipStopped*.

Выгрузка инертных материалов из скипового подъемника происходит автоматически при достижении положения выгрузки. Метод *LoadInerts* и событие *InertsLoaded* необходимы только для отчета времени выгрузки и реакции на завершение этого времени.

6.5. Управление смесителем

Смеситель включается вызовом метода *StartMixer*. По завершению включения миксера генерируется входное воздействие *MixerStarted*. Смеситель выключается вызовом метода *StopMixer* с последующей генерацией входного воздействия *MixerStopped*. Информацию о состоянии смесителя предоставляет свойство *IsMixerOn*, которое имеет значение true при включенном смесителе и false в обратном случае. В описываемом процессе смеситель необходимо включать после того как дозированы инертные материалы или, если инертных материалов нет в рецепте, после дозировки всех материалов. Смеситель необходимо выключить после того как произойдет выгрузка последнего цикла.

6.6. Смешивание

После того как будут выгружены все материалы, должен быть начат процесс смешивания. Он начинается по вызову метода *Mix* и заканчивается генерацией входного воздействия *Mixed*. Данные метод и событие необходимы только для отчета времени смешивания.

6.7. Выгрузка

После смешивания следует произвести процесс выгрузки. Он начинается по вызову метода *Unload* и заканчивается генерацией входного воздействия *Unloaded*.

ВЫВОДЫ ПО ГЛАВЕ 6

Разработанное инструментальное средство было применено для реализации реальной задачи. С помощью визуального редактора для редактирования моделей конечных автоматов, построены диаграммы автоматов дозирования и загрузки компонентов раствора, а также смешивания полученных компонентов. По полученным диаграммам сгенерирован исходный код для управления процессом приготовления бетонного раствора. Приложение было протестировано на бетонном заводе. Благодаря проведенным испытаниям, была доказана работоспособность разработанного средства. Реализация логики управления установкой для приготовления раствора с помощью разработанных диаграмм существенно уменьшила количество кода, написанного вручную. Наличие диаграмм ведет к лучшему понимаю принципа работы установки и способствует быстрому устранению ошибок, в случае их появления.

ЗАКЛЮЧЕНИЕ

В результате выполненной работы создана библиотека классов для реализации конечных автоматов. Возможности разработанной библиотеки шире возможностей библиотек-аналогов, написанных для программирования с их помощью на языке C# в среде разработки *Microsoft Visual Studio*. Данная библиотека не лишена недостатков, но без особых трудностей может быть улучшена такими введениями, как: автоматическое протоколирование действий автомата, обработка ошибок выполнения методов выходных воздействий, введение собственного хранилища данных, для использования его внутри автомата и т. д.

В результате выполненной работы также создано инструментальное средство визуального проектирования конечных автоматов в среде разработки программного обеспечения *Microsoft Visual Studio 2008*. С помощью этого средства возможно использование автоматного программирования при разработке приложений любой направленности. Возможность генерации исходного кода визуально проектируемых автоматов существенно уменьшает количество написанного вручную кода, а также повышает наглядность и надежность программ, в которых применены модели конечных автоматов.

Возможности инструментального средства визуально проектировать системы вложенных автоматов, группировать состояния, а также устанавливать для каждого автомата способ обработки входных воздействий, до настоящего момента отсутствовали в программах-аналогах. Взаимодействие родительского и вложенного автомата разработчик должен был каждый раз реализовывать самостоятельно. Введение новых возможностей в визуальный редактор стало возможным благодаря поддержке соответствующей функциональности библиотекой *FSMLib*.

ИСТОЧНИКИ

1. *Object Management Group.* Unified Modeling Language. http://www.uml.org
2. *MSDN.* Visual Studio. http://msdn.microsoft.com/ru-ru/vstudio/
3. *Решетников Е.О.* Инструментальное средство для визуального проектирования автоматных программ на основе Microsoft Domain-Specific Language Tools. http://is.ifmo.ru/papers/reshetnikov_bachelor/
4. *Mealy G.*, A Method for Synthesizing Sequential Circuits. //Bell System Technical Journal. 1955. V. 34. № 5, pp.1045 – 1079.
5. *Шалыто А.А*, SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. с. 628.
6. *Gamma E., Helm R., Johnson R.*, Vlissides J. Design Patterns. MA: Addison-Wesley Professional. 2001. – 395 p.
7. *Шамгунов Н.Н., Корнеев Г.А., Шалыто А.А.*, State Machine – новый паттерн объектно-ориентированного проектирования. //Информационно-управляющие системы. 2004. №5, с.13 – 25.
8. *Шопырин Д.Г.*, Метод проектирования и реализации конечных автоматов на основе виртуальных вложенных классов. //Информационно-управляющие системы. 2005. № 1, с.87 – 96.
9. *Шопырин Д.Г.*, Объектно-ориентированная реализация конечных автоматов на основе виртуальных методов. //Информационно-управляющие системы. 2005. № 3, с. 36 – 40.
10. *Астафуров А.А.*, Декларативный подход к вложению и наследованию автоматных классов при использовании императивных языков программирования / Software Engineering Conference (Russia) (SEC(R) 2007). М.: ТЕКАМА. 2007, с. 230 – 238.

11. *van Gurp J., Jan Bosch J.* On the Implementation of Finite State Machines. /3rd Annual IASTED International Conference Software Engineering and Applications. Scottsdale. Arizona. 1999, pp.172 – 178. http://www.jillesvangurp.com/static/fsm-sea99.pdf
12. *Шопырин Д., Шалыто А.* Объектно-ориентированный подход к автоматному программированию. //Информационно-управляющие системы. 2003. № 5, с. 29 – 39.
13. *Ларионов А.* Визуальный язык автоматного программирования для *Microsoft Visual Studio 2005*. http://is.ifmo.ru/papers/larionov/
14. *IAR Systems*. IAR visualSTATE. http://iar.com/website1/1.0.1.0/371/1/
15. Finite State Machine Editor. http://fsme.sourceforge.net/
16. StarUML. The Open Source UML/MDA Platform. http://staruml.sourceforge.net/
17. BOUML. http://bouml.free.fr/
18. MSDN. *Domain-Specific Language Tools*. http://msdn.microsoft.com/en-us/library/bb126235(VS.80).aspx
19. NordWestSoft. Software Development Company. http://nordwestsoft.com/

ПРИЛОЖЕНИЯ

Приложение 1. Исходный код классов библиотеки *FSMLib*.

Класс *StateMachine*

```
using System;
using System.Collections.Generic;
using System.Threading;
using FSM.Core.Model.Collections;

namespace FSM.Core.Model
{
    public enum EventHandlingMode
    {
        Synchronous,
        Asynchronous
    }

    public enum EventHandlingOrder
    {
        InnerStateMachineUntilFinished,
        InnerStateMachineBeforeCurrent,
        CurrentStateMachineBeforeInner
    }

    public class StateMachine
    {
        public StateMachine()
        {
            _states = new BaseStateList<State> { StateMachine = this };
            _transitions = new TransitionList { StateMachine = this };
            _initialState = new BaseStateList<InitialState> { StateMachine =
this };
            _finalStates = new BaseStateList<FinalState> { StateMachine = this
};
            _groupStates = new BaseStateList<GroupState> { StateMachine = this
};
        }

        #region Public properties

        private string _name;
        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        private readonly BaseStateList<State> _states;
        public BaseStateList<State> States { get { return _states; } }

        private readonly TransitionList _transitions;
        public TransitionList Transitions { get { return _transitions; } }

        private readonly BaseStateList<InitialState> _initialState;
        public InitialState InitialState
        {
            get { return _initialState[0]; }
            set
            {
                if ( _initialState[0] == value )
                    return;

                _initialState.Remove( _initialState[0] );
                _initialState.Add( value );
            }
```

```
            }

            private readonly BaseStateList<FinalState> _finalStates;
            public BaseStateList<FinalState> FinalStates { get { return
_finalStates; } }

            private readonly BaseStateList<GroupState> _groupStates;
            public BaseStateList<GroupState> GroupStates { get { return
_groupStates; } }

            private BaseState _currentState;
            public BaseState CurrentState { get { return _currentState; } }

            private EventHandlingMode _eventHandlingMode =
EventHandlingMode.Synchronous;
            public EventHandlingMode EventHandlingMode
            {
                get
                {
                    return _eventHandlingMode;
                }
                set
                {
                    Utils.Utils.Assert( !_isRunning, "event handling mode could not
be set when state machine is running" );

                    _eventHandlingMode = value;
                }
            }

            private EventHandlingOrder _eventHandlingOrder =
EventHandlingOrder.InnerStateMachineUntilFinished;
            public EventHandlingOrder EventHandlingOrder
            {
                get
                {
                    return _eventHandlingOrder;
                }
                set
                {
                    Utils.Utils.Assert( !_isRunning, "event handling order could not
be set when state machine is running" );

                    _eventHandlingOrder = value;
                }
            }

            private bool _isRunning;
            public bool IsRunning
            {
                get { return _isRunning; }
            }

            public bool IsFinished
            {
                get { return ( _currentState as FinalState ) != null; }
            }

            #endregion

            #region Public methods

            private readonly object _runLocker = new object();
            private Thread _thread;
            public void Run()
            {
                lock ( _runLocker )
                {
                    Utils.Utils.Assert( !_isRunning, "could not run State Machine
when it is already running" );
```

```
                    _isRunning = true;

                    Utils.Utils.Assert( _initialState[0] != null, "initialState is
not set" );
                    SetState( InitialState );

                    if ( _eventHandlingMode == EventHandlingMode.Asynchronous )
                    {
                        _thread = new Thread( StateMachineLoop );
                        _thread.Start();
                    }

                    HandleEvent( "*" );
                }
            }

            private readonly object _handleEventLocker = new object();
            public void HandleEvent( string eventName )
            {
                lock ( _handleEventLocker )
                {
                    if ( !_isRunning )
                        return;

                    if ( _eventHandlingMode == EventHandlingMode.Asynchronous )
                    {
                        lock ( _eventQueue )
                        {
                            _eventQueue.Enqueue( eventName );
                            Monitor.PulseAll( _eventQueue );
                        }
                    }
                    else
                    {
                        InternalHandleEvent( eventName );
                    }
                }
            }

            private readonly object _terminateLocker = new object();
            public void Terminate()
            {
                lock ( _terminateLocker )
                {
                    if ( IsFinished )
                        return;

                    State state = _currentState as State;
                    if ( state != null )
                        TerminateAll( state.InnerStateMachines );

                    if ( _thread != null )
                    {
                        _thread.Abort();
                        _thread = null;
                    }

                    FinalState finalState = new FinalState();
                    finalState.Name = "State after terminate";

                    _currentState = finalState;
                    _isRunning = false;
                }
            }

            #endregion

            #region Private methods

            private readonly Queue<string> _eventQueue = new Queue<string>();
            private void StateMachineLoop()
```

```
        {
            while ( true )
            {
                lock ( _eventQueue )
                {
                    while ( _eventQueue.Count == 0 )
                        Monitor.Wait( _eventQueue );

                    InternalHandleEvent( _eventQueue.Dequeue() );

                    if ( IsFinished )
                        break;
                }
            }
        }

        private void InternalHandleEvent( string eventName )
        {
            if ( IsFinished )
                return;

            State state = _currentState as State;
            if ( state != null )
            {
                switch ( _eventHandlingOrder )
                {
                    case EventHandlingOrder.InnerStateMachineUntilFinished:
                        {
                            if          (          !AllStateMachinesAreFinished(
state.InnerStateMachines ) )
                            {
                                PropagateEvent(        state.InnerStateMachines,
eventName );
                                return;
                            }
                            break;
                        }
                    case EventHandlingOrder.CurrentStateMachineBeforeInner:
                        {
                            if ( !CanHandle( eventName ) )
                            {
                                PropagateEvent(        state.InnerStateMachines,
eventName );
                                return;
                            }
                            break;
                        }
                    case EventHandlingOrder.InnerStateMachineBeforeCurrent:
                        {
                            if ( CanHandle( state.InnerStateMachines, eventName
) )
                            {
                                PropagateEvent(        state.InnerStateMachines,
eventName );
                                return;
                            }
                            break;
                        }
                    default:
                        throw new InvalidOperationException();
                }
            }

            Transition nextTransition = GetTransition( eventName );
            if ( nextTransition != null )
            {
                if ( state != null )
                {
                    TerminateAll( state.InnerStateMachines );
                    state.Exit();
                }
```

```
                nextTransition.MakeAction();

                SetState( nextTransition.Target );
            }
        }

        private bool CanHandle( string eventName )
        {
            return GetTransition( eventName ) != null;
        }

        private static bool CanHandle( IEnumerable<StateMachine> stateMachines,
string eventName )
        {
            foreach ( StateMachine stateMachine in stateMachines )
            {
                if ( stateMachine.CanHandleRecursive( eventName ) )
                    return true;
            }
            return false;
        }

        private bool CanHandleRecursive( string eventName )
        {
            State state = _currentState as State;
            if ( state != null )
            {
                foreach ( StateMachine stateMachine in state.InnerStateMachines
)
                {
                    if ( stateMachine.CanHandleRecursive( eventName ) )
                        return true;
                }
            }
            return CanHandle( eventName );
        }

        private         static         bool         AllStateMachinesAreFinished(
IEnumerable<StateMachine> stateMachines )
        {
            foreach ( StateMachine stateMachine in stateMachines )
            {
                if ( !stateMachine.IsFinished )
                    return false;
            }
            return true;
        }

        private Transition GetTransition( string eventName )
        {
            if ( _currentState == null )
                return null;

            GroupedElement groupedElement = _currentState as GroupedElement;
            while ( groupedElement != null )
            {
                Transition          transition          =          GetTransition(
groupedElement.ParentGroupState, eventName );
                if ( transition != null )
                    return transition;
                groupedElement = groupedElement.ParentGroupState;
            }

            return GetTransition( _currentState, eventName );
        }

        private  static  Transition  GetTransition(  BaseState  fromState,  string
eventName )
        {
            Transition guardedTransition = null;
```

```
                Transition freeTransition = null;
                if ( fromState != null )
                {
                    foreach ( Transition transition in
fromState.OutcomingTransitions )
                    {
                        if ( ( ( transition.Event == eventName ) || (
transition.Event == Transition.AnyEvent ) ) && ( transition.IsConditionSatisfied ) )
                        {
                            if ( transition.Guard == null )
                                freeTransition = transition;
                            else
                                guardedTransition = transition;
                        }
                    }
                }
                return guardedTransition ?? freeTransition;
            }

            private static void PropagateEvent( IEnumerable<StateMachine>
stateMachines, string eventName )
            {
                foreach ( StateMachine stateMachine in stateMachines )
                    stateMachine.HandleEvent( eventName );
            }

            private void SetState( BaseState newState )
            {
                _currentState = newState;

                System.Diagnostics.Debug.WriteLine( string.Format( "\"{0}\" is now
current state", ( _currentState != null ) ? _currentState.Name : string.Empty ) );

                InitialState initialState = newState as InitialState;
                if ( initialState != null )
                    initialState.Initialize();

                State state = newState as State;
                if ( state != null )
                {
                    state.Enter();
                    RunAll( state.InnerStateMachines );
                }

                FinalState finalState = newState as FinalState;
                if ( finalState != null )
                {
                    finalState.Complete();
                    _isRunning = false;
                }
            }

            private void RunAll( IEnumerable<StateMachine> stateMachines )
            {
                foreach ( StateMachine stateMachine in stateMachines )
                    stateMachine.Run();
            }

            private void TerminateAll( IEnumerable<StateMachine> stateMachines )
            {
                foreach ( StateMachine stateMachine in stateMachines )
                    stateMachine.Terminate();
            }

            #endregion
        }
    }
```

Класс *StateMachineEntity*

```
using System;

namespace FSM.Core.Model
{
    public class StateMachineEntity
    {
        internal StateMachine StateMachine { get; set; }

        protected void MakeAction( Action action )
        {
            if ( action != null )
            {
                action();

                System.Diagnostics.Debug.WriteLine( string.Format( "    \"{0}\"
{1} action", action.Method.Name, GetType().Name ) );
            }
        }
    }
}
```

Класс *Transition*

```
using System;

namespace FSM.Core.Model
{
    public class Transition : StateMachineEntity
    {
        internal const string AnyEvent = "*";

        public Transition()
        {
        }

        public Transition( string @event, BaseState source, BaseState target )
        {
            Event = @event;
            Source = source;
            Target = target;
        }

        public BaseState Source { get; set; }

        public BaseState Target { get; set; }

        public string Event { get; set; }

        public Action Action;

        public Func<bool> Guard;

        internal void MakeAction()
        {
            MakeAction( Action );
        }

        internal bool IsConditionSatisfied
        {
            get { return ( Guard != null ) ? Guard() : true; }
        }
    }
}
```

Класс *BaseState*

```
using System.Collections.Generic;

namespace FSM.Core.Model
```

```
{
    public abstract class BaseState : StateMachineEntity
    {
        private readonly IList< Transition > _incomingTransitions = new List< Transition >();
        internal virtual IList< Transition > IncomingTransitions { get { return _incomingTransitions; } }

        private readonly IList<Transition> _outcomingTransitions = new List<Transition>();
        internal virtual IList<Transition> OutcomingTransitions { get { return _outcomingTransitions; } }

        private string _name = string.Empty;
        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }

        private string _description = string.Empty;
        public string Description
        {
            get { return _description; }
            set { _description = value; }
        }
    }
}
```

Класс *GroupedElement*

```
namespace FSM.Core.Model
{
    public abstract class GroupedElement : BaseState
    {
        internal GroupState ParentGroupState { get; set; }
    }
}
```

Класс *InitialState*

```
using System;
using System.Collections.Generic;

namespace FSM.Core.Model
{
    public class InitialState : BaseState
    {
        public Action InitialAction;

        internal void Initialize()
        {
            MakeAction( InitialAction );
        }

        internal override IList<Transition> IncomingTransitions { get { return new List<Transition>(); } }
    }
}
```

Класс *State*

```
using System;
using System.Collections.Generic;

namespace FSM.Core.Model
{
    public class State : GroupedElement
```

```
        {
            public Action EntryAction;

            public Action ExitAction;

            private readonly IList<StateMachine> _innerStateMachines = new
List<StateMachine>();
            public IList<StateMachine> InnerStateMachines { get { return
_innerStateMachines; } }

            internal void Enter()
            {
                MakeAction( EntryAction );
            }

            internal void Exit()
            {
                MakeAction( ExitAction );
            }
        }
    }
```

Класс *FinalState*

```
    using System;
    using System.Collections.Generic;

    namespace FSM.Core.Model
    {
        public class FinalState : BaseState
        {
            public Action FinalAction;

            internal void Complete()
            {
                MakeAction( FinalAction );
            }

            internal override IList<Transition> OutcomingTransitions { get { return
new List<Transition>(); } }
        }
    }
```

Класс *GroupState*

```
    using System.Collections.Generic;
    using FSM.Core.Model.Collections;

    namespace FSM.Core.Model
    {
        public class GroupState : GroupedElement
        {
            public GroupState()
            {
                _states = new GroupStateChildren { GroupState = this };
            }

            private readonly GroupStateChildren _states;
            public GroupStateChildren States { get { return _states; } }

            internal override IList<Transition> IncomingTransitions { get { return
new List<Transition>(); } }
        }
    }
```

Класс *StateMachineEntityList*

```
    using System.Collections;
```

```
using System.Collections.Generic;

namespace FSM.Core.Model.Collections
{
    public class StateMachineEntityList<T> : IEnumerable<T> where T : StateMachineEntity
    {
        private readonly IList<T> _entities = new List< T >();

        public StateMachine StateMachine { get; set; }

        #region Implementation of IEnumerable

        public IEnumerator< T > GetEnumerator()
        {
            return _entities.GetEnumerator();
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }

        #endregion

        public virtual void Add( T t )
        {
            Utils.Utils.Assert( (StateMachine == null) || (!StateMachine.IsRunning), "State Machine should not be changed during it is running" );

            if ( t == null )
                return;

            if ( !_entities.Contains( t ) )
            {
                t.StateMachine = StateMachine;
                _entities.Add( t );
            }
        }

        public virtual void Remove( T t )
        {
            Utils.Utils.Assert( ( StateMachine == null ) || ( !StateMachine.IsRunning ), "State Machine should not be changed during it is running" );

            if ( t == null )
                return;

            if ( _entities.Contains( t ) )
                _entities.Remove( t );
        }

        public bool Contains( T t )
        {
            if ( t == null )
                return false;

            return _entities.Contains( t );
        }

        public int Count
        {
            get { return _entities.Count; }
        }

        public T this[int index]
        {
            get { return ( ( index >= 0 ) && ( index < _entities.Count ) ) ? _entities[ index ] : null; }
```

```
            }
        }
    }
```

Класс *TransitionList*

```
    using System;

    namespace FSM.Core.Model.Collections
    {
        public class TransitionList : StateMachineEntityList<Transition>
        {
            public override void Add( Transition transition )
            {
                if ( transition == null )
                    return;

                if ( ( transition == null ) || ( transition.Target == null ) || (
transition.Source == null ) )
                    throw new NullReferenceException();

                transition.Source.OutcomingTransitions.Add( transition );
                transition.Target.IncomingTransitions.Add( transition );

                base.Add( transition );
            }

            public override void Remove( Transition transition )
            {
                if ( transition == null )
                    return;

                if ( ( transition == null ) || ( transition.Target == null ) || (
transition.Source == null ) )
                    throw new NullReferenceException();

                transition.Source.OutcomingTransitions.Remove( transition );
                transition.Target.IncomingTransitions.Remove( transition );

                base.Remove( transition );
            }
        }
    }
```

Класс *BaseStateList*

```
    using System;
    using System.Collections.Generic;

    namespace FSM.Core.Model.Collections
    {
        public class BaseStateList<T> : StateMachineEntityList<T> where T :
BaseState
        {
            private readonly Dictionary<string, T> _entityByName = new
Dictionary<string, T>();

            public override void Add( T t )
            {
                base.Add( t );

                if ( _entityByName.ContainsKey( t.Name ) && ( _entityByName[t.Name]
!= t ) )
                    throw new InvalidOperationException(
                        string.Format( "Entities in {0} should have unique names.",
GetType().FullName ) );

                _entityByName[t.Name] = t;
            }
```

```
            public override void Remove( T state )
            {
                if ( state == null )
                    return;

                if ( StateMachine == null )
                    throw new NullReferenceException();

                HashSet<Transition> deletingTransitions = new HashSet< Transition
>();
                foreach ( Transition transition in state.IncomingTransitions )
                    deletingTransitions.Add( transition );
                foreach ( Transition transition in state.OutcomingTransitions )
                    deletingTransitions.Add( transition );
                foreach ( Transition transition in deletingTransitions )
                    StateMachine.Transitions.Remove( transition );

                _entityByName.Remove( state.Name );

                base.Remove( state );
            }

            public T this[string name]
            {
                get { return ( _entityByName.ContainsKey( name ) ) ?
_entityByName[name] : null; }
            }
        }
    }
```

Класс *GroupStateChildren*

```
    using System;

    namespace FSM.Core.Model.Collections
    {
        public class GroupStateChildren : BaseStateList<GroupedElement>
        {
            public GroupState GroupState { get; set; }

            public override void Add( GroupedElement groupedElement )
            {
                if ( groupedElement.ParentGroupState != null )
                    throw new InvalidOperationException();

                groupedElement.ParentGroupState = GroupState;

                base.Add( groupedElement );
            }

            public override void Remove( GroupedElement groupedElement )
            {
                if ( !Contains( groupedElement ) )
                    throw new InvalidOperationException();

                groupedElement.ParentGroupState = null;

                base.Remove( groupedElement );
            }
        }
    }
```

Приложение 2. Шаблон генерации кода конечного автомата с использованием библиотеки FSMLib

```
    <#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" #>
```

```
<#@ output extension=".cs" #>
<#@    StateMachineLanguage    processor="StateMachineLanguageDirectiveProcessor"
requires="fileName='Sample.stateMachine'" #>
<#@ import namespace="System.Collections.Generic" #>
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

using FSM.Core.Model;

namespace <#= StateMachineRoot.Namespace #>
{
    partial class <#= StateMachineRoot.Classname #>
    {
<#
foreach ( BaseStateMachine baseStateMachine in StateMachineRoot.StateMachines )
{
    StateMachine stateMachine = baseStateMachine as StateMachine;
    if ( stateMachine == null )
        continue;
    string stateMachineMemberName = GetMemberName( stateMachine.Name );
#>
        #region <#= stateMachine.Name #>

        #region Initialization

        private StateMachine <#= stateMachineMemberName #>;
        public StateMachine <#= GetUniqueName( stateMachine.Name ) #>
        {
            get
            {
                if ( <#= stateMachineMemberName #> != null )
                    return <#= stateMachineMemberName #>;
<#
    ClearData();
    PushIndent( new string( ' ', 16 ) );
    PrintStatesDeclaration( stateMachine.States );
    PrintTransitionsDeclaration( stateMachine.States );
    PrintStateMachineDeclaration( stateMachine );
    PopIndent();
#>
                return <#= stateMachineMemberName #>;
            }
        }

        #endregion

<#
    if ( _actions.Count > 0 )
    {
#>
        #region Actions

<#
    PushIndent( new string( ' ', 8 ) );
    PrintActionDeclarations();
    PopIndent();
#>
        #endregion

<#
    }
    if ( _predicates.Count > 0 )
    {
#>
        #region Predicates
```

```
        <#
            PushIndent( new string( ' ', 8 ) );
            PrintPredicateDeclarations();
            PopIndent();
        #>
                #endregion

        <#
            }
        #>
                #endregion

        <#
        }

        _referencedStateMachines.Clear();
        foreach ( BaseStateMachine baseStateMachine in StateMachineRoot.StateMachines )
        {
            StateMachine stateMachine = baseStateMachine as StateMachine;
            if ( stateMachine == null )
                continue;
            CollectReferencedStateMachines( stateMachine.States );
        }
        if ( _referencedStateMachines.Count > 0 )
        {
        #>
                #region Referenced State Machines

        <#
            foreach      (      ReferencedStateMachine      referencedStateMachine      in
_referencedStateMachines.Keys )
            {
        #>
                private  StateMachine  <#=  GetMemberName(  referencedStateMachine.Name  )
#>;
                public StateMachine <#= GetUniqueName( referencedStateMachine.Name ) #>
                {
                    get
                    {
                        return <#= GetMemberName( referencedStateMachine.Name ) #>;
                    }
                    set
                    {
        <#
                foreach            (            BaseState            baseState            in
_referencedStateMachines[referencedStateMachine] )
                {
        #>
                        ((State)<#=       GetPropertyPathToState(       baseState       )
#>).InnerStateMachines.Remove( <#= GetMemberName( referencedStateMachine.Name ) #> );
        <#
                }
        #>
                        <#= GetMemberName( referencedStateMachine.Name ) #> = value;
        <#
                foreach            (            BaseState            baseState            in
_referencedStateMachines[referencedStateMachine] )
                {
        #>
                        ((State)<#=       GetPropertyPathToState(       baseState       )
#>).InnerStateMachines.Add( <#= GetMemberName( referencedStateMachine.Name ) #> );
        <#
                }
        #>
                    }
                }

        <#
            }
        #>
```

```
            #endregion
        <#
        }
        #>
            }
        }

        <#+
        private string _initialState;
        private readonly List<string> _states = new List<string>();
        private readonly List<string> _finalStates = new List<string>();
        private readonly List<string> _groups = new List<string>();
        private readonly List<string> _transitions = new List<string>();

        private int _number;
        private readonly Dictionary<BaseState, BaseState> _markedStates = new
Dictionary<BaseState, BaseState>();

        private readonly Dictionary<string, string> _actions = new Dictionary<string,
string>();
        private readonly Dictionary<string, string> _predicates = new Dictionary<string,
string>();

        private readonly Dictionary<ReferencedStateMachine, List<BaseState>>
_referencedStateMachines = new Dictionary<ReferencedStateMachine, List<BaseState>>();

        private void ClearData()
        {
            _initialState = string.Empty;
            _states.Clear();
            _finalStates.Clear();
            _groups.Clear();
            _transitions.Clear();

            _number = 0;
            _markedStates.Clear();

            _actions.Clear();
            _predicates.Clear();
        }

        private void PrintStatesDeclaration( IEnumerable<BaseState> states )
        {
            if ( states == null )
                return;
            foreach ( BaseState baseState in states )
            {
                InitialState initialState = baseState as InitialState;
                State state = baseState as State;
                FinalState finalState = baseState as FinalState;
                Group group = baseState as Group;

                string stateVariableName = GetUniqueName( baseState.Name );
                if ( initialState != null )
                {
                    _initialState = stateVariableName;
                    WriteLine( "//" );
                    WriteLine( "//" + initialState.Name );
                    WriteLine( "//" );
                    WriteLine( string.Format( "InitialState {0} = new InitialState();",
stateVariableName ) );
                    WriteLine( string.Format( "{0}.Name = \"{1}\";", stateVariableName,
initialState.Name ) );
                    if ( !string.IsNullOrEmpty( initialState.InitialActivity ) )
                    {
                        _actions[GetInitialActionHandlerName( initialState )] =
initialState.InitialActivity;
                        WriteLine( string.Format( "{0}.InitialAction = {1};",
stateVariableName, GetInitialActionHandlerName( initialState ) ) );
                    }
                }
```

```
            else if ( state != null )
            {
                _states.Add( stateVariableName );
                WriteLine( "//" );
                WriteLine( "//" + state.Name );
                WriteLine( "//" );
                WriteLine(   string.Format(   "State   {0}   =   new   State();",
stateVariableName ) );
                WriteLine( string.Format( "{0}.Name = \"{1}\";", stateVariableName,
state.Name ) );
                if ( !string.IsNullOrEmpty( state.EntryActivity ) )
                {
                    _actions[GetEntryActionHandlerName(       state       )]       =
state.EntryActivity;
                    WriteLine(    string.Format(    "{0}.EntryAction    =    {1};",
stateVariableName, GetEntryActionHandlerName( state ) ) );
                }
                if ( !string.IsNullOrEmpty( state.ExitActivity ) )
                {
                    _actions[GetExitActionHandlerName(       state       )]       =
state.ExitActivity;
                    WriteLine(    string.Format(    "{0}.ExitAction    =    {1};",
stateVariableName, GetExitActionHandlerName( state ) ) );
                }
                foreach ( BaseStateMachine baseStateMachine in state.StateMachines )
                {
                    StateMachine stateMachine = baseStateMachine as StateMachine;
                    if ( stateMachine != null )
                        WriteLine( string.Format( "{0}.InnerStateMachines.Add( {1}
);", stateVariableName, GetUniqueName( stateMachine.Name ) ) );
                }
            }
            else if ( finalState != null )
            {
                _finalStates.Add( stateVariableName );
                WriteLine( "//" );
                WriteLine( "//" + finalState.Name );
                WriteLine( "//" );
                WriteLine( string.Format( "FinalState {0} = new FinalState();",
stateVariableName ) );
                WriteLine( string.Format( "{0}.Name = \"{1}\";", stateVariableName,
finalState.Name ) );
                if ( !string.IsNullOrEmpty( finalState.FinalActivity ) )
                {
                    _actions[GetFinalActionHandlerName(     finalState     )]     =
finalState.FinalActivity;
                    WriteLine(    string.Format(    "{0}.FinalAction    =    {1};",
stateVariableName, GetFinalActionHandlerName( finalState ) ) );
                }
            }
            else if ( group != null )
            {
                _groups.Add( stateVariableName );
                PrintStatesDeclaration( group.States );
                WriteLine( "//" );
                WriteLine( "//" + group.Name );
                WriteLine( "//" );
                WriteLine( string.Format( "GroupState {0} = new GroupState();",
stateVariableName ) );
                WriteLine( string.Format( "{0}.Name = \"{1}\";", stateVariableName,
group.Name ) );
                foreach ( BaseState childBaseState in group.States )
                {
                    State childState = childBaseState as State;
                    Group childGroup = childBaseState as Group;
                    if ( childState != null )
                        WriteLine(   string.Format(   "{0}.States.Add(   {1}   );",
stateVariableName, GetUniqueName( childState.Name ) ) );
                    else if ( childGroup != null )
                        WriteLine(   string.Format(   "{0}.States.Add(   {1}   );",
stateVariableName, GetUniqueName( childGroup.Name ) ) );
```

```
                    else
                        throw new InvalidOperationException( "Group should contain
states and other groups only." );
                }
            }
            else
                throw new InvalidOperationException( "Code generation fails. Type of
some state is not supported." );
        }
    }

    private void PrintTransitionsDeclaration( IEnumerable<BaseState> states )
    {
        foreach ( BaseState state in states )
        {
            if ( _markedStates.ContainsKey( state ) )
                continue;
            _markedStates[state] = state;
            foreach ( Transition transition in Transition.GetLinksToTargets( state )
)
            {
                _number++;
                WriteLine( "//" );
                WriteLine(    "//"    +    transition.Source.Name    +    "->"    +
transition.Target.Name );
                WriteLine( "//" );
                string variableName = "transition" + _number.ToString();
                _transitions.Add( variableName );
                WriteLine(  string.Format(  "Transition  {0}  =  new  Transition();",
variableName ) );
                WriteLine(  string.Format(  "{0}.Source  =  {1};",  variableName,
GetUniqueName( transition.Source.Name ) ) );
                WriteLine(  string.Format(  "{0}.Target  =  {1};",  variableName,
GetUniqueName( transition.Target.Name ) ) );
                WriteLine(  string.Format(  "{0}.Event  =  \"{1}\";",  variableName,
transition.Trigger ) );
                if ( !string.IsNullOrEmpty( transition.Guard ) )
                {
                    _predicates[GetTransitionGuardHandlerName(  transition  )]  =
transition.Guard;
                    WriteLine(  string.Format(  "{0}.Guard  =  {1};",  variableName,
GetTransitionGuardHandlerName( transition ) ) );
                }
                if ( !string.IsNullOrEmpty( transition.Activity ) )
                {
                    _actions[GetTransitionActionHandlerName(  transition  )]  =
transition.Activity;
                    WriteLine(  string.Format(  "{0}.Action  =  {1};",  variableName,
GetTransitionActionHandlerName( transition ) ) );
                }
            }
            PrintTransitionsDeclaration( Transition.GetTargets( state ) );
        }
    }

    private void PrintStateMachineDeclaration( StateMachine stateMachine )
    {
        WriteLine( "//" );
        WriteLine( "//" + stateMachine.Name );
        WriteLine( "//" );
        string stateMachineMemberName = GetMemberName( stateMachine.Name );
        WriteLine(    string.Format(    "{0}    =    new    StateMachine();",
stateMachineMemberName ) );
        WriteLine(  string.Format(  "{0}.Name  =  \"{1}\";",  stateMachineMemberName,
stateMachine.Name ) );
        WriteLine( string.Format( "{0}.EventHandlingMode = EventHandlingMode.{1};",
stateMachineMemberName, stateMachine.EventHandlingMode.ToString() ) );
        WriteLine(        string.Format(        "{0}.EventHandlingOrder        =
EventHandlingOrder.{1};",                                    stateMachineMemberName,
stateMachine.EventHandlingOrder.ToString() ) );
```

```
            WriteLine( string.Format( "{0}.InitialState = {1};", stateMachineMemberName,
_initialState ) );
            foreach ( string stateName in _states )
                WriteLine(      string.Format(      "{0}.States.Add(      {1}      );",
stateMachineMemberName, stateName ) );
            foreach ( string finalStateName in _finalStates )
                WriteLine(     string.Format(     "{0}.FinalStates.Add(     {1}     );",
stateMachineMemberName, finalStateName ) );
            foreach ( string groupName in _groups )
                WriteLine(     string.Format(     "{0}.GroupStates.Add(     {1}     );",
stateMachineMemberName, groupName ) );
            foreach ( string transitionName in _transitions )
                WriteLine(     string.Format(     "{0}.Transitions.Add(     {1}     );",
stateMachineMemberName, transitionName ) );
        }

        private void PrintActionDeclarations()
        {
            foreach ( string handler in _actions.Keys )
            {
                WriteLine( string.Format( "protected virtual void {0}()", handler ) );
                WriteLine( "{" );
                WriteLine( string.Format( "    {0};", _actions[handler] ) );
                WriteLine( "}" );
                WriteLine( "" );
            }
        }

        private void PrintPredicateDeclarations()
        {
            foreach ( string handler in _predicates.Keys )
            {
                WriteLine( string.Format( "protected virtual bool {0}()", handler ) );
                WriteLine( "{" );
                WriteLine( string.Format( "    return {0};", _predicates[handler] ) );
                WriteLine( "}" );
                WriteLine( "" );
            }
        }

        private    readonly    Dictionary<string,    string>    _uniqueNames    =    new
Dictionary<string, string>();
        private  readonly  Dictionary<string,  int>  _postfixes  =  new  Dictionary<string,
int>();
        private string GetUniqueName( string name )
        {
            if ( _uniqueNames.ContainsKey( name ) )
                return _uniqueNames[name];
            string result = string.Empty;
            foreach ( char ch in name )
            {
                if ( char.IsLetterOrDigit( ch ) )
                    result += ch;
            }
            if ( string.IsNullOrEmpty( result ) )
                result = "name";
            if ( !_postfixes.ContainsKey( result ) )
            {
                _postfixes[result] = 0;
            }
            else
            {
                _postfixes[result]++;
                result += _postfixes[result].ToString();
            }
            _uniqueNames[name] = result;
            return result;
        }

        private string GetMemberName( string name )
        {
```

```
            string result = GetUniqueName( name );
            return    string.Format(    "_{0}{1}",    char.ToLower(    result[0]    ),
result.Substring( 1 ) );
        }

        private string GetInitialActionHandlerName( InitialState initialState )
        {
            return string.Format( "On{0}Initialize", GetUniqueName( initialState.Name )
);
        }

        private string GetEntryActionHandlerName( State state )
        {
            return string.Format( "On{0}Enter", GetUniqueName( state.Name ) );
        }

        private string GetExitActionHandlerName( State state )
        {
            return string.Format( "On{0}Exit", GetUniqueName( state.Name ) );
        }

        private string GetFinalActionHandlerName( FinalState finalState )
        {
            return string.Format( "On{0}Complete", GetUniqueName( finalState.Name ) );
        }

        private string GetTransitionGuardHandlerName( Transition transition )
        {
            return  string.Format(  "IsFrom{0}To{1}ConditionSatisfied",  GetUniqueName(
transition.Source.Name ), GetUniqueName( transition.Target.Name ) );
        }

        private string GetTransitionActionHandlerName( Transition transition )
        {
            return      string.Format(      "OnFrom{0}To{1}Action",      GetUniqueName(
transition.Source.Name ), GetUniqueName( transition.Target.Name ) );
        }

        private void CollectReferencedStateMachines( IEnumerable<BaseState> states )
        {
            foreach ( BaseState baseState in states )
            {
                Group group = baseState as Group;
                if ( group != null )
                {
                    CollectReferencedStateMachines( group.States );
                    continue;
                }
                State state = baseState as State;
                if ( state != null )
                {
                    foreach ( BaseStateMachine baseStateMachine in state.StateMachines )
                    {
                        ReferencedStateMachine referencedStateMachine = baseStateMachine
as ReferencedStateMachine;
                        if ( referencedStateMachine != null )
                        {
                            if          (          !_referencedStateMachines.ContainsKey(
referencedStateMachine ) )
                                _referencedStateMachines[referencedStateMachine]  =  new
List<BaseState>();
                            _referencedStateMachines[referencedStateMachine].Add(  state
);
                        }
                    }
                }
            }
        }

        private string GetPropertyPathToState( BaseState state )
        {
```

```
            List<BaseState> path = new List<BaseState>();
            BaseState tmp = state;
            while ( tmp.ParentGroup != null )
            {
                tmp = tmp.ParentGroup;
                path.Add( tmp );
            }
            string result = GetUniqueName( tmp.StateMachine.Name );
            for ( int i = path.Count - 1; i >= 0; i-- )
            {
                string childrenProperty = ( i + 1 == path.Count ) ? "GroupStates" :
"States";
                result = string.Format( "((GroupState){0}.{1}[\"{2}\"])", result,
childrenProperty, path[i].Name );
            }
            return string.Format( "{0}.States[\"{1}\"]", result, state.Name );
        }
        #>
```