Идея написания библиотеки работы с конечными автоматами на Nemerle пришла давно, вот здесь началась развиваться тема на форуме:  
[http://rsdn.ru/forum/nemerle/3716028.aspx](http://rsdn.org/forum/nemerle/3716028.aspx)  
  
Все это время я потихоньку развивал проект, сейчас библиотека вошла в рабочую фазу, реализованы основные функции семантики автоматов UML. С выходом Nemerle.Peg добавились новые возможности создания языка DSL. Подобные библиотеки есть во всех языках, например boost::statechart в C++, Sanford.StateMachineToolkit, SimpleStateMachine — C# State Machine Library — Boo and Rhino DSL, StaMa, Stateless. Хотелось бы иметь и такую же в Nemerle, плюсы от использования Nemerle очевидны, это удобный разбор dsl языка описания автомата (в аналогичных библиотеках как правило неудобно создавать сами диаграммы), анализ на этапе компиляции, генерация высокоэффективного кода, практически полное отсутствие динамических конструкций ускоряющие код. Идея хорошо себя оправдала, я тестировал аналогичные автоматы Sanford.StateMachineToolkit и моей библиотеки, разница в скорости в плюс библиотеки на Nemerle была в десятки раз. Все это говорит о том что аналогов подобной библиотеки в мире нет, остальные библиотеки проигрывают в скорости и удобстве описания автомата, все это дает причину опубликовать библиотеку для широкого использования. Хотел спросить есть ли у сообщества согласие на хостинг библиотеки в snippets, я хотел назвать Nemerle.Statechart.  
Текущие возможности библиотеки таковы:  
— язык описания диаграммы состояний представляет собой текстовый dsl вида

state Parked

{

EngineStarted => IdleInNeutral;

}

state IdleInNeutral

{

EngineKilled => Parked;

GearEngaged => IdleInGear;

}

state IdleInGear

{

GearDisengaged => IdleInNeutral;

GasApplied => RacingAlong;

}

state RacingAlong

{

BrakeApplied => IdleInGear;

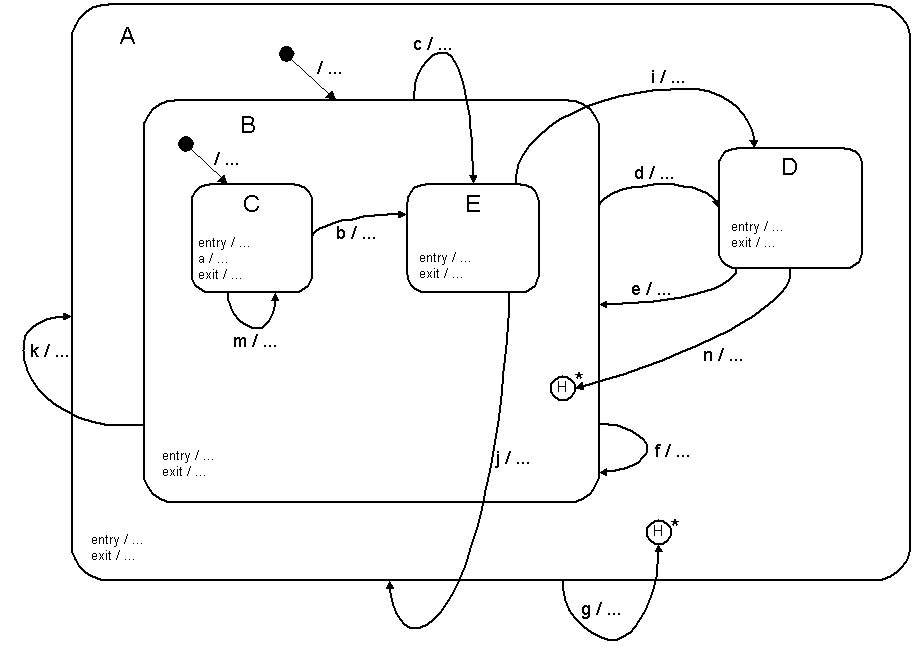
CarCrashedIntoTree => CarDestroyed;

}

state CarDestroyed

{

}

Здесь изображен плоский автомат, библиотека включает большинство из возможностей uml автоматов, например такой автомат:  
  
  
можно дополнить сторожевыми условиями, входными выходными действиями и получить в конечном счете что то типа:

[statechart(<#

flags : auto\_initial, transition\_completed\_events debug;

state A

{

(H) => D;

$>;

$<;

b => E;

f => D; // cross

state B

{

0 / init\_action => C;

(H\*) => E;

$>;

$<;

b => D; // cross

d => D;

f [test\_guard1] / f\_action =>@;

k => A;

c => E;

\_ => D;

state C

{

$>;

a / A;

$<;

b [test\_guard2] => E;

m =>@;

}

state E

{

$>;

$<;

i => D;

j => A;

o /final\_action1 => final;

}

}

state D

{

$>;

$<;

e => B;

n => B.H;

}

g => H;

}

#>

)]

class PathCoverFsm

{

//[GuardFor(test\_guard1)]

test\_guard1() : bool// {get;set;}

{

true

}

test\_guard2 : bool// {get;set;}

{

get

{

true

}

}

test\_guard3 : bool// {get;set;}

{

get

{

true

}

}

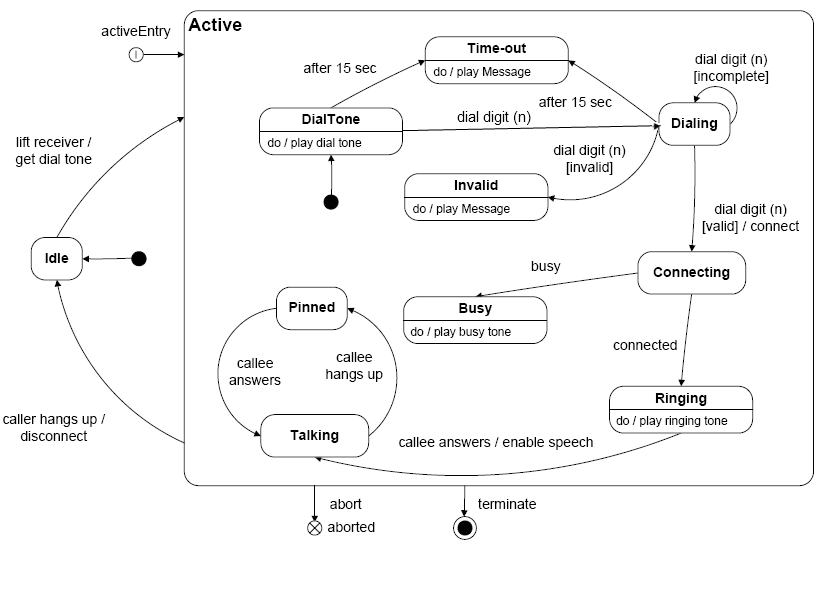
}

}

вся эта штука потом парсится, анализируется и в конечном счете получается код класса содержащие определения состояний, события обработчики для действий, входных, выходных и при переходах и это можно использовать в другом коде. Сама обработка событий основывается на виртуальных методах.  
  
весь список возможностей на текущий день:  
  
— парсинг метаттрибута текстового описания автомата  
— анализ путей переходов  
— генерация представления автомата в класс  
— анализ и генерация истории shallow и deep history  
— переходы по умолчанию из исторических состояний  
— сторожевые условия  
— встраиваемые члены сторожевых условий  
— конечные состояния (final)  
— начальные состояния (initial)  
— действия при входе и выходе  
— сброс истории при переходе в конечное состояние (check spec)  
— приоритет переходов вложенных классов  
— внутренние переходы  
  
еще не достает до полноты:  
-terminal pseudostate  
-do activity  
-choice  
-junction  
-fork and join pseudostates  
-orthogonal regions  
-local and external transitions  
-deffered events   
  
Что скажет сообщество на сей творческий порыв

Здравствуйте, VladD2, Вы писали:  
  
VD>Здравствуйте, CodingUnit, Вы писали:  
  
CU>>Хотел спросить есть ли у сообщества согласие на хостинг библиотеки в snippets, я хотел назвать Nemerle.Statechart.  
  
VD>Все будут только "за".  
  
VD>Только делать теперь это нужно через создание fork-а и размещения пул-реквеста на github. Тут рядом это все описано. Почитай, плиз.  
VD>[http://www.rsdn.ru/forum/nemerle/4336310.1.aspx](http://rsdn.org/forum/nemerle/4336310.1.aspx)  
  
CU>>еще не достает до полноты:  
CU>> -terminal pseudostate  
CU>> -do activity  
CU>> -choice  
CU>> -junction  
CU>> -fork and join pseudostates  
CU>> -orthogonal regions  
CU>> -local and external transitions  
CU>> -deffered events   
  
VD>Еще бы понимать что все это значит http://rsdn.org/Forum/Images/smile.gif.  
  
Это элементы семантики UML автомата, все это я разъясню в процессе описания библиотеки, скажу вкратце,  
что будет в библиотеке:  
текущие возможности:  
— создание иерархических автоматов UML, с поддержкой входных, выходных действий, начальные, конечные, исторические состояния  
  
будущие возможности из описанных выше:  
— terminal pseudo state — терминальное состояние при котором автомат должен завершатся  
— do activity это деятельность которую автомат должен осуществлять в состоянии, то есть некоторый поток который должен работать  
— orthogonal regions — ортогональные области, это области в которых автомат пребывает одновременно, то есть параллельно в двух и более состояниях, ака параллелизм, моделируется с помощью многопоточности  
— junction — сложный переход, в котором может быть одно событие но на пути много разных разветвлений в разные состояния по разным сторожевыми условиям  
— fork and join это псевдосостояния помогающие разделить поток на несколько параллельных, join синхронизирует потоки  
— local and external это разные виды переходов, local не вызывает выход из состояния при переходе, подразумевается переход в подсостояние  
— deffered events — отложенные события, события которые должны откладываться до лучших времен  
  
VD>Скажет — давай! Только к этому делу еще нужно качественное описание и побольше примеров.  
  
Описание постараюсь набросать, начну с некоторого начального описания библиотеки на примерах, которое потом внесу в документацию библиотеки.  
  
Библиотека Nemerle.Statechart это ответ сообщества Nemerle на запрос автоматного, реактивного и событийного программирования, в ней предполагается внести перспективные функции, возможные с помощью системы метапрограммирования Nemerle. Сама библиотека представлена в виде макроса, который делится на три части:  
— парсер (основанный на Nemerle.Peg) который обеспечивает преобразование текстового описания автомата в дерево  
— анализ дерева автомата, это статический анализ, переходов, действий, проверка на правильность описания, обработка и создания структур для генерации  
— генерация автомата в запускаемый код, в виде класса, в котором представлены подклассы состояний, переменные текущего состояния, историй, событий действий которые срабатывают при входах, выходах из состояний и переходах  
  
Для чего все это нужно?   
  
Автоматы состояний — это поведенческая абстракция, которая помогает разделить общее поведение системы или класса на некоторое множество разных состояний, соединенных переходами и реагирующими на события. Как известно новейшие языки и платформы делают шаги в сторону реактивности и событийности, введение в язык конструкции делегатов, событий, помогает строить поведение классов с помощью концепции событийного программирования.  
  
Цитата из википедии:

Событи́йно-ориенти́рованное программи́рование (англ. event-driven programming; в дальнейшем СОП) — парадигма программирования, в которой выполнение программы определяется событиями — действиями пользователя (клавиатура, мышь), сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета).  
  
Как правило, в реальных задачах оказывается недопустимым длительное выполнение обработчика события, поскольку при этом программа не может реагировать на другие события. В связи с этим при написании событийно-ориентированных программ часто применяют автоматное программирование.  
  
Событийно-ориентированное программирование, как правило, применяется в трех случаях:  
  
при построении пользовательских интерфейсов (в том числе графических);  
при создании серверных приложений в случае, если по тем или иным причинам нежелательно порождение обслуживающих процессов;  
при программировании игр, в которых осуществляется управление множеством объектов.

Применение и плюсы автоматов состояний широко известны, такой подход позволяет строить поведение системы визуально, на практике я столкнулся с тем что гору if else конструкций можно преобразовать в несколько легко представляемых состояний, разделенных переходами, которые визуально легко представляются и переносятся в код, при этом код понятен, и чист, легко расширяется, обдумывается и отлаживается из за того что описание автомата легко представляется в виде диаграммы.   
В автоматном программировании, поведение системы или класса может легко расширяться, за счет добавления состояний или переходов, а не модификации логики, в какой то мере это часто лучше рекурсии и даже удобных match конструкций.  
Вот здесь интересные статьи по системам работающим по событиям:   
<http://www.excode.ru/art6033p2.html>  
<http://www.ict.edu.ru/vconf/files/3195.rtf>  
[здесь](http://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)  
  
В процессе жизненного цикла система или класс системы, может менять свое состояние, как реакция на внешние события, в каждом состоянии объект себя ведет по разному. Автоматы состояний можно использовать при моделировании поведения графического интерфейса, как реакции на действия пользователя, различные приложения с множеством разных режимов работы в которых система ведет себя по разному, моделирование объектов реального мира, например в играх солдат может быть в нескольких состояниях: стоит, бежит, лежит, стреляет, оружие достано, говорит, спит, целится в каждом из этих состояний он меняет свой внешний вид (что может осуществлятся при входе в это состояние), и по другому реагирует на внешние события, например может споткнутся если бежит, на какие то события он может не реагировать в определенных состояниях, а в каких то наоборот может, например стреляет в ответ на выстрел, что не может делать если оружие не достано или если он спит.   
Разные потоки workflow, цикл жизни объекта, разные протоколы легко реализуются на автоматах, причем эта библиотека с легкостью даст возможность автоматизировать классы, и использовать в системах реального времени, где требуется высокая вычислительная скорость, поскольку за счет статического анализа, изменения состояний и переходы осуществляются очень быстро, как правило это сводится к присваиванию полю класса, нескольких вызовов и запуска событий.  
Практически любая сложная логика может быть описана автоматами, автоматы начали применятся и исследоваться с середины прошлого века, в разных научных кругах, отечественных и зарубежныхх разработках.   
Сначала применялись различные подходы такие как детерменированные и недетерменированные автоматы, автоматы Мили и Мура, затем Д.Харел разработал более сложную и полную семантику автоматов, иерархических автоматов, в которых состояния группировались на основе принципа Или и И, Или когда автомат находится в одном из состояний, И когда в нескольких (параллельных). Введены разные теории семантики автоматов, которые были объеденены в стандарте UML.   
Я вкратце опишу диаграммы и их составляющие, для тех кто не знаком с диаграммами состояний UML, а рассмотрю их интерпретацию в языке описания автоматов в библиотеке и представлю несколько примеров.  
  
Вот статьи по теории диаграмм состояний:  
<http://www.business-process.ru/designing/methodology/uml/theory/statechat_diagram_theory.html>  
<http://khpi-iip.mipk.kharkiv.edu/library/case/leon/gl6/gl6.html>  
<http://ertostik.net/uml/diagramma-sostoyaniy-state-diagram.html>  
  
вся диаграмма представляет собой ориентированный граф, вершинами которого являются состояния, а переходы дугами.  
Диаграмма состоит из композитных состояний, которые включают другие подсостояния, с одним возможно неявным верхним состоянием.  
Когда состояние находится во внутреннем состоянии, то активны и все его родительские состояния.  
  
— черный кружок со стрелкой есть начальное состояние, когда автомат начинает работу он входит в верхнее начальное состояние, переход из которого осуществляется в одно из подсостояний, когда автомат входит в композитное состояние через переход, то он входит в его начальное состояние, в которое осуществляется переход из начального состояния  
  
Вот диаграмма состояний телефона:  
  
Система телефона находится в одном из следующих состояний:  
1) Сначала система входит в состояние Idle, то есть телефон ожидает и пребывает в нем  
2) Когда наступает событие lift receiver то есть поднята трубка, осуществляется переход из состояния Idle в состояние Active при этом сначала вызывается действие get dial tone — получить сигнал выдачи гудка (?),  
3) При входе в состояние Active, переход осуществлятся сразу в его начальное псевдосостояние, из которого сразу же система переходит в состояние Dial Tone то есть проигрывание гудка  
4) do / play dial tone — обозначает деятельность внутри состояния, то есть состояние осуществляет активность проигрывает гудок  
5) при наступлении события after 15 sec, то есть после 15 секунд, телефон переходит в состояние Time-out в котором постоянно проигрывает сообщение, из этого состояния он может выйти только если положена трубка caller hangs up переход из родительского состояния Active, то есть принадлежащий ко всем внутренним состояниям, если автомат находится в любом из подсостояний Active то событие caller hangs up выводит из него автомат и переводит в состояние Idle то есть если кладем трубку то телефон ожидает  
6) Далее последовательно автомат переходит по соотв. событиям состояния Connecting (соединения), Ringing (звонит), Busy, Talking  
Дальнейшее обсуждение диаграммы состояний я думаю лишне, должна быть уже понятна концепция, весь этот автомат текстово можно представить в следующем виде (в будущем надеюсь сделать перевод графического представления из редактора автоматически в автомат):

// макроаттрибут statechart

[statechart(<#

flags : auto\_initial; // начальные флаги автомата

// определяем состояние Idle

state Idle

{

lift\_receiver / get\_dial\_tone => Active; // переход: событие / действие => конечное состояние

}

// определяем состояние Active

state Active

{

caller\_hangs\_up / disconnect => Idle; //

terminate => $0; // переход в конечное состояние

state DialTone

{

do / play\_dial\_tone; // внутренняя активность (пока не поддерживается)

after15\_sec => TimeOut;

dial\_digit => Dialing;

}

state TimeOut

{

}

state Dialing

{

dial\_digit [incomplete] => @; // переход со сторожевым условием в себя

after15\_sec => TimeOut;

dial\_digit [invalid] => Invalid;

dial\_digit [valid] / connect => Connecting;

}

state Invalid

{

do / playMessage;

}

state Connecting

{

connected => Ringing;

busy => Busy;

}

state Busy

{

do / play\_busy\_tone;

}

state Ringing

{

do / play\_ringing\_tone;

callee\_answers / enable\_speech => Talking;

}

state Talking

{

callee\_hangs\_up => Pinned;

}

state Pinned

{

callee\_answers => Talking;

}

}

#>)

class PhoneFsm

{

}

Далее после компиляции макрос должен проанализировать автомат и создать методы, он создаст действия при переходах, do хотя пока и не поддерживается, его можно заменить входными действиям, можно описать ключевые слова используемые в языке:

state Имя // состояние

{

entry / action; // действие при входе

exit / action2; // действие при выходе

\_ =>

(H\*) => Имя2;// историческое состояние, автомат запоминает последнее состояние при выходе, переход в это состояние возвращает запомненное состояние

// \* означает Deep History - глубокая история которая запоминает подсостояние на любой глубине вложенности

// стрелочка означает переход по умолчанию из исторического состояния, если нет истории осуществляется он

state Имя2

{

$>; // входное действие (альтернативный синтаксис) для него создается событие внутри класса,

// к которому можно присоединить обработчик запускающийся при входе в состояние

$<; // выходное действие

trigger [guard] / action => State3; // событие [сторожевое условия] / действие при переходе => Состояние

// сторожевые условия вычисляются при наступлении события, если оно истинно значит переход сработает

(H) // историческое состояние Shallow - запоминает верхнее подсостояние этого состояния

evt => final; // переход в конечное состояние, при этом композитное состояние завершается

}

}

state State3

{

exit

{

Something2 // действие при входе 1

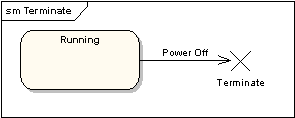
Something3 // действие при входе 2

} // другой синтаксис для входного действия

A => Имя.H; // переход в историческое состояние

}

Тут я описал основной синтаксис, оставшуюся семантику UML думаю со временем включу, парсер пока не распознает сложные и ошибки и выдает обычные сообщения которые в него заложены, синтаксис расширяем если у кого есть мысли как его улучшить, анализатор и генератор тоже если будут найдены баги или если будут советы по улучшению архитектуры можно поправить. На этом сообщение завершаю, после рассмотрю остальные аспект и дам больше примеров. Я сделал пулл реквест библиотеки, когда она будет добавлена в git ее можно скачать и попробовать. Буду рад любым комментариям и вопросам.

Создал пул реквест с новыми фичами, реализовал:  
  
— поддержку do активности в состоянии на базе Task из .net 4  
— разные флаги к этой фиче чтобы можно было настраивать   
— terminate узел (см. описание Terminate)  
— провел некоторые оптимизации в анализаторе и генераторе, убрал обработчики действий в соотв. им вх/вых действия  
— сделал пример с gui для работы с файлами  
  
Do активность позволяет состоянию внутри себя производить некоторую деятельность, когда состояние активируется оно запускает свою деятельность помеченную как: do / имя  
внутри описания состояния и эта активность работает в параллельном процессе независимо от поступающих событий. Если состояние изменяется на другое или автомат останавливается то активность завершается принудительно. Это сделано на базе Task и токенов отмены как посоветовал Влад. Теперь эту возможность можно использовать, я пока еще не встречал библиотек которые бы включали эту возможность языка UML. Ниже я опишу как эту возможность можно использовать и ее нотацию.   
Еще одна из задач библиотеки наиболее более полно приблизиться к стандарту UML, и передать его семантику диаграмм состояний, что реализовано не во всех библиотеках. Для этого я перелопачиваю сейчас стандарт, и справочник по UML, и разные описания семантики диаграмм состояний, беседую с западными авторами статей по поводу тонких моментов семантики.   
Надеюсь эта цель будет достигнута, от сообщества же требуется хорошенько оттестить библиотеку и выловить баги, разные случаи и запрещенные комбинации, я выловил уже большую часть, но могут быть разные тонкие моменты, ведь язык это вещь сложная, библиотека по сути включает в себя маленький компилятор (парсер, анализатор, генерация).  
  
Terminate узел это псевдосостояние (псевдосостояние означает что это не настоящее состояние, в нем автомат не может пребывать, я постепенно опишу все псевдосостояния, они дают разные управляющие конструкции для работы с автоматом) при входе в которое автомат завершается, то есть уничтожается. Я реализовал этот узел в виде вызова метода Terminate в определенной точке истории автомата, то есть при переходе по определенному событию в узел Terminate. Прошлый пример с системой сохранения файлов, я дополнил узлом terminate, логика дополнена так что если пользователь запускает выход, то запрашивается вопрос о сохранении если файл не был сохранен.  
  
Диаграмма дополняется узлом terminate, нотация которого может быть:  
state A  
{  
terminate;  
x;  
X;  
}  
  
  
Он отображается как большой крестик поэтому и выбран синтаксис Х,   
в ниже описанном автомате взятом из предыдущего примера узел объявлен как X; в верхнем состоянии. Код дополняется следующим:

[statechart(<#

flags : auto\_initial transition\_completed\_events;

0 => Ожидание;

state НенужноСохранение

{

$> / СохранениеВыкл;

НовыйФайл => НужноСохранение;

ОткрытФайл =>@;

**Выход => X;**

state Сохранен

{

$> / НадписьИмяФайла ПоследнийФайлТек;

Сохранение =>@;

Изменение => Измененный;

}

state Ожидание

{

$> / СохранениеВсеВыкл НадписьНазваниеПрограммы;

$< / СохранениеКакВкл;

}

}

state НужноСохранение

{

$> / СохранениеВкл;

ОткрытФайл, Сохранение => НенужноСохранение;

НовыйФайл =>@;

**Выход => X;**

state Новый

{

$> / НадписьФайл ПоследнийФайлПустой;

}

state Измененный

{

$> / НадписьИзменен;

}

}

**X;**

#>

В тело класса можно встраивать методы для работы, в примере работы с файлами я встраивал методы запуска событий, которые дают сообщение о сохранении перед запуском события, на этапе теста я не переопределял методы уже существующих событий (хотя надо бы сделать именно так, чтобы нельзя было вызвать события без проверок), но для теста оставил чтобы тестировать автомат без проверок и диалогов. Простое определение метода дает встроить логику в автомат состояний, можно делать переопределение, это можно сделать двумя путями, назначив члену тоже имя что и в определении автомата, например:  
  
если в классе объявить метод НовыйФайл то он переопределит запуск события Новый Файл которые срабатывают также через методы:

[statechart()]

class FileFsm

{

public НовыйФайл() : void

{

when (НовыйДиалог()) НовыйФайл();

}

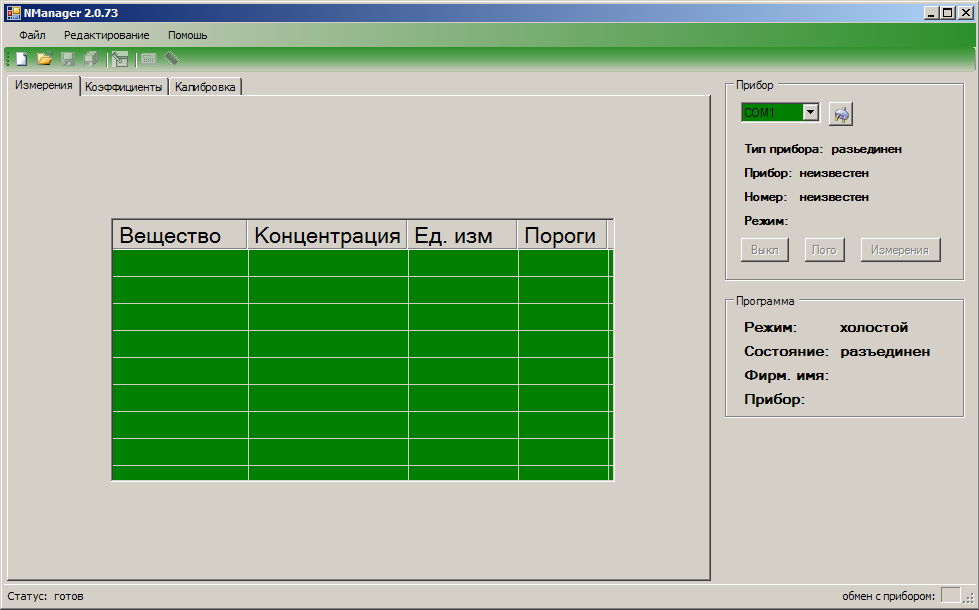
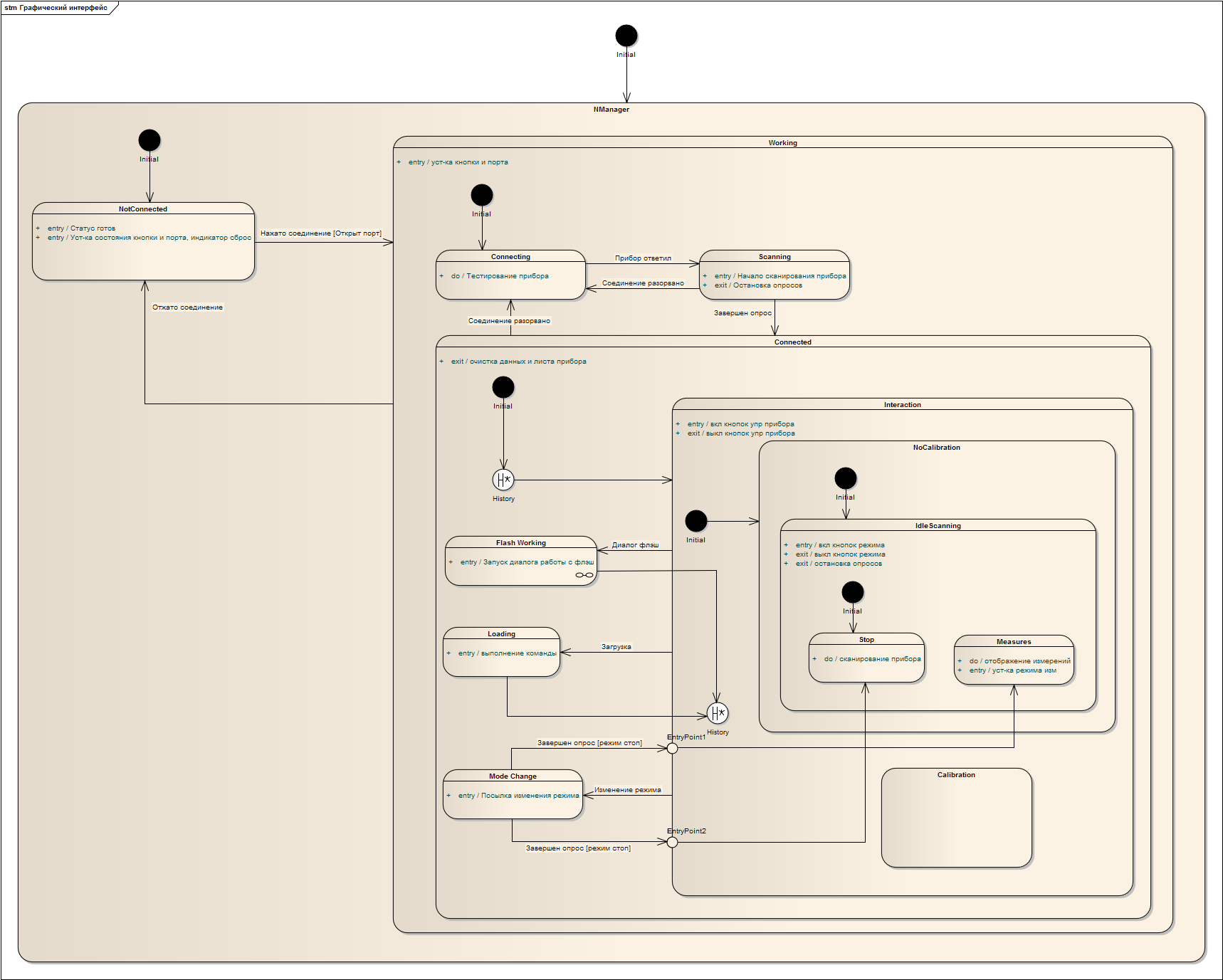
[EventFor(Сохранение)] // переопределяет событие Сохранение

public Сохранить() : void

{

}

}

Чтобы задать явно что метод переопределяет какое то событие это можно сделать через аттрибут EventFor(имясобытия), тогда он переопределит его и будет везде использоваться этот метод. Помимо запуска событий через вызовы методов, которые генерируются, есть еще метод RunEvent который генерируется автоматически и делает тоже самое но принимает перечисление в качестве параметра и запускает обработчики внутри себя на основе перечисления.  
Переопределять можно не только события, но также действия и сторожевые условия. Семантика при этом подобна описанной выше для событий за исключением того, что  
аттрибуты называются ActionFor и GuardFor соответственно. Сторожевые условия несколько отличаются от действий тем что они представляются не событиями а свойством,  
с выходным типом void -> bool то есть функцией которая должна возвратить true или false для соответствующего сторожевого условия.  
  
Сегодня опишу еще один вариант автомата, с новыми возможностями историей и внутренней активностью, это реальный случай из моей практики. Мне пришлось писать приложение для взаимодействия с прибором, который присоединялся к компьютеру через COM или USB порт. Прибор надо было калибровать, считывать, записывать данные, измерять с помощью него. Логику приложения и gui я вынес полностью в автомат, который я и представлю, без него я бы долго и муторно выслеживал ошибки и пытался бы наладить правильную логику, что я уже перенес в своей практике, с помощью автоматов этого можно легко избежать.  
  
На диаграмме представлен автомат работы программы, программа связывается с прибором и имеет в своем интерфейсе кнопку соединения, которая запоминает свое состояние (залипает), выбор COM порта, сообщения о состоянии программы и прибора.  
  
  
  
Вот так выглядит диаграмма состояний.  
  
  
Программа в процессе своей жизненной истории переживает несколько состояний или режимов, основными двумя ее состояниями являются NotConnected и Working.   
В первом оно пребывает по умолчанию при входе, во второе переходит когда пользователь нажимает связь с прибором. Все состояния обрамляются одним большим композитным  
состоянием, которое просто обобщает весь автомат.  
  
1) При входе в состояние NotConnected запускаются действия при входе в это состояние, отключаются элементы работы с прибором, которые должны быть активны только если  
он подсоединен, статус сбрасывается на обычный, контролы для запуска соединения включены  
  
2) При нажатии на кнопку соединения в правом верхнем углу, запускается событие Нажато Соединение, при этом программа пытается открыть выбранный порт и проверяет открывается ли он.  
Сторожевое условие проверяется однократно при наступлении события, если оно истинно то переход запускается если ложно то событие игнорируется и переход не запускается.  
Здесь это помогает тем что кнопка не залипает лишний раз если порт не открылся  
  
3) Если порт открылся и запускается переход, автомат начинает вход в состояние Working, оно композитное то есть содержит вложенные подсостояния, одно из них может быть активно,  
но если активно вложенное то и все его родители. Это дает например если в композитном состоянии Working запускается событие Отжато соединение то автомат выходит из  
всех (любого) активных состояний внутри него и входит в состояние NotConnected, то есть все вложенные состояния реагируют на событие в родительском состоянии.  
  
4) Итак автомат начинает вход в состояние Working, сначала он запускает действие при входе залипает кнопка и выключаются кнопки работы с портом, чтобы нельзя было   
выбрать порт, поскольку программа работает только с одним портом. Переход внутри композитного состояния начинается с начального псевдосостояния (initial transition).  
В нашем случае из начального псевдосостояния идет переход в состояние Connecting.   
  
5) Сначала необходимо подать тестовый запрос в порт и определить есть ли прибор на линии, прибор при этом должен ответить. Здесь показан пример использования внутренней  
активности в состоянии Connecting при этом программа последовательно начинает подавать запросы в линию, прибора может и не быть подключено тогда программа  
будет просто висет и пытаться снова.   
  
6) Если произошло соединение и прибор ответил срабатывает переход в состояние Scanning, в этом состоянии надо опросить состояние прибора снять с него его имена,  
идентификатор и опросить остальные параметры. При входе в это состояние Начинается подаваться очередь запросов, на каждый из которых должен быть получен ответ,  
если ответ не был получен или получен некорректно то запускается событие Соединение разорвано и автомат переходит в состояние Соединения снова, и пытается подать  
последовательно тестовый запрос и все остальные чтобы получены были адекватные ответы  
  
7) Если очередь запросов была пройдена и получены ответы (не случилось сбоя), то прибор входит в состояние Connected, из его начального состояния переход идет  
прямо в историческое состояние. Что это такое? Историческое состояние запоминает "историю", то есть последнее состояние в котором находилось это родительское состояние.   
Существует два типа исторического псевдосостояния, shallow history (обозначается H в кружке) и deep history (H\* в кружке), первая запоминает состояние на первом уровне вложенности, то есть последнее верхнее подсостояние,  
в состоянии Connected ими являются Interaction, Flash Working, Loading и Mode Change. Второй тип истории запоминает любое подсостояние на любом уровне вложенности,  
им могут быть все вышеописанные включая их подсостояния как Stop, Measures в Interaction.   
  
8) Переход в историческое состояние активирует его историю, то есть автомат сразу входит в то состояние в котором находился (по правилам истории), в нашем случае   
deep history (со звездочкой), но какое состояние будет если он еще не был в этом состоянии? Ответ прост, он переходит в состояние по умолчанию, для этого существует  
такое понятие как default transition, или переход по умолчанию из исторического состояния, он показывает состояние которое должно активироваться если истории  
еще нет. У нас он начинает свой путь с Interaction. Сохранение же состояния всегда осуществляется при выходе из того состояния в котором объявлено историческое  
псевдосостояние.  
  
9) Итак начинает осществляться вход в состояние Interaction, в этом состоянии пользователь должен общаться с прибором, автомат входит в Idle Scanning, сканирование  
прибора и Stop. При этом включатся кнопки управления режимами прибора, можно запустить его на измерение и остановить.  
  
10) При этом в состоянии Stop программа просто постоянно опрашивает данные с прибора, его калибровочные данные в нижнем правом углу бегает индикатор связи с прибором  
  
11) Если в этом состоянии нажать кнопку разъединения, то осуществляется выход из всех состояний в Working, в обратном порядке и осуществляется переход в NotConnected,  
при этом порядок таков: выход Stop, выход IdleScanning, NoCalibration, Interaction (сохраняется его история), Connected (сохраняется его история), Working. При этом запускаются  
все действия при выходе каждого из состояний из которых выходим и действия при входе в те состояние в которые входим. Входим в NotConnected.  
  
12) Далее если мы нажимаем кнопку соединения вновь, то программа уже знает что мы работали в том то режиме и восстанавливает с помощью истории предыдущий  
режим, визуально это выглядит интеллектуально, программа продумывает эти ходы (с помощью автомата), последовательно переходя через Connecting, Scanning поскольку  
обязательно надо соединиться и просканировать данные (может быть подключен другой прибор или отключен вообше), далее в истории Connected находится запомненное  
глубокое состояние Stop в которое и входит автомат последовательно запуская все входные действия на пути из Connected в Stop.  
  
13) Чтобы перевести прибор и программу в состояние измерения, мы запускаем кнопку управления, которые активны при этом надо послать сначала команду прибору  
на измерение, программа при этом через событие Изменение режима переходит из Interaction в Mode Change, в нем посылается запрос прибору и производится переход  
когда прибор ответил, но переход уже нужен не в Stop а в Measures (измерения), для этого используются сторожевые условия на переходах из состояния Mode Change,  
срабатывает один из этих переходов с правильным сторожевым условием  
  
14) В состоянии измерение программа посылает запросы и отображает даннные измерения  
  
15) Также есть возможность перевода в режим работы с флэш памятью прибора, Flash Working при этом все запросы прерываются при выходе из Idle Scanning, Interaction  
и в диалоге работы с флэш можно посмотреть флэш память, очистить, считать данные. Когда пользователь закрывает диалог программа входит в историческое состояние  
в состоянии Interaction то есть восстанавливает предыдущий режим работы  
  
К сожалению показать как это все работает в реальном времени не представляется возможнным (нужен прибор и программа), но поверьте на слово, логика  
пользовательского интерфейса программы выглядит очень интересно и умно за счет этих включений, выключений контролов в нужное время, индикаторов  
обмена, изменяющихся сообщений состояния, что называется реактивный пользовательский интерфейс. И на отладку этого дела я потратил очень малое время,  
вся отладка сводилась к правильному пониманию диаграммы и введения ее соответствия в коде, чтобы все обработчики были подключены и запускали нужные действия.  
  
При этом возможности приложения последовательно наращиваются, сначала не было истории, появилась, потом добавляется состояние FlashWorking которого не было, и так  
далее, все это довольно легко расширяемо и быстро перенастраивается. В диаграмме это только надо добавить новое подсостояние и переход к нему, или объединить   
состояния в один родитель, текст переписать в описании несложно, но при этом изменяется логика во всем приложении и после перекомпиляции будет новой. Макроаттрибут  
перекомпилируется довольно быстро, сама библиотека только долго собирается (400 кб уже весит dll ка), но для тех кто ее не собирает это не существенно.

[statechart(<#

flags : auto\_initial;

state NotConnected

{

entry / status\_ready;

entry / buttons\_init;

connecting\_clicked [PortOpened] => Working;

}

state Working

{

state Connecting

{

do / testing\_connection; // внутренняя деятельность в состоянии (крутится в нем)

}

state Scanning

{

$> / begin\_scan; // альтернативный синтаксис входного и выходного действия

$< / stop\_reqs;

}

state Connected

{

0 => H; // начальный переход в историческое состояние

(H\*) => Interaction; // историческое состояние и переход по умолчанию

state Interaction

{

$> / pribor\_controls\_ena;

$< / pribor\_controls\_dis;

(H\*) // другое историческое состояние

state NoCalibration

{

state IdleScanning

{

entry / mode\_buttons\_ena;

exit / mode\_buttons\_dis;

state Stop

{

do / scan;

}

state Measures

{

do / show\_meas;

}

}

}

state Calibration

{

}

mode\_change => ModeChange;

flash\_dialog => FlashWorking;

}

state FlashWorking

{

\_ => Interaction.H;

}

state ModeChange

{

reqs\_complete [mode\_is\_stop] => Measures;

reqs\_complete [mode\_is\_measure] => Stop;

}

state Loading

{

}

}

}

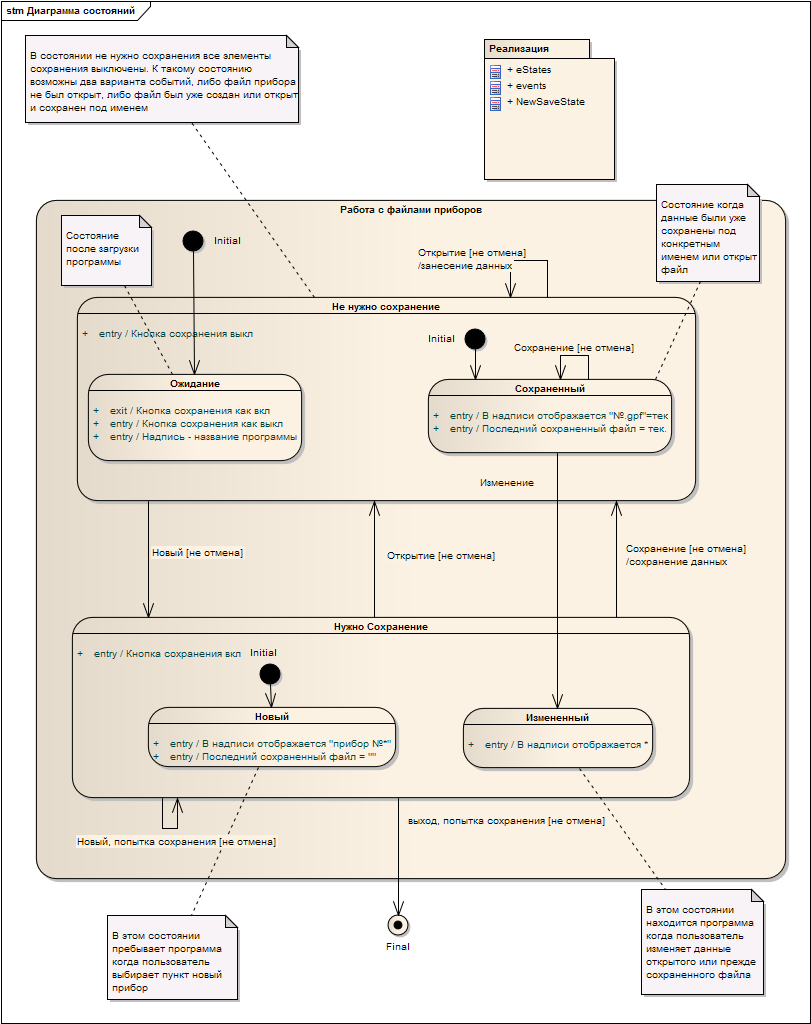
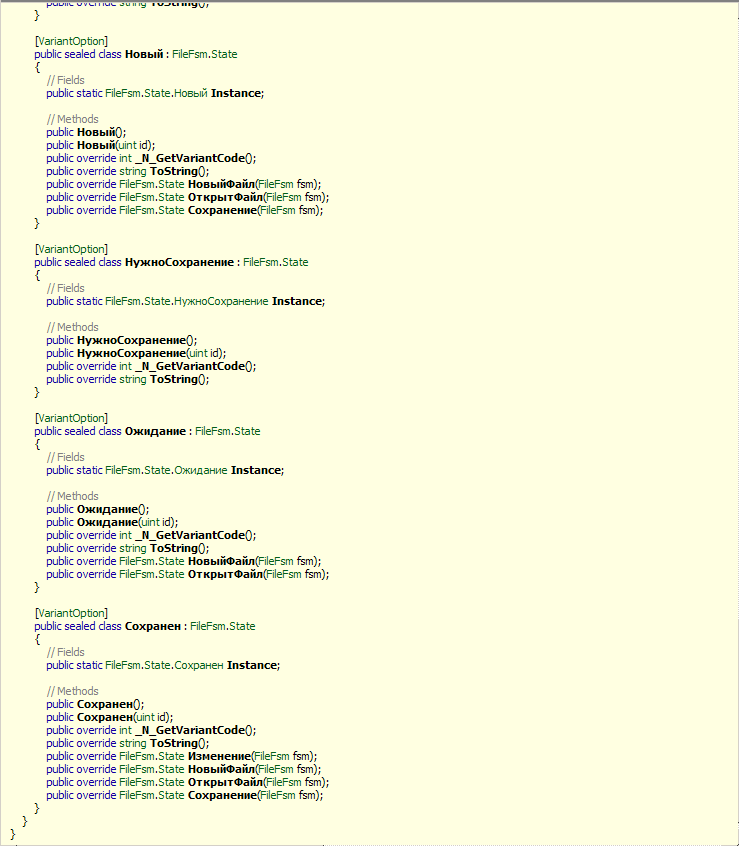
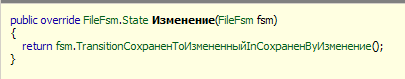
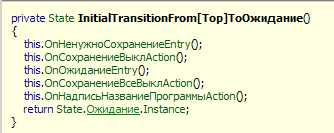
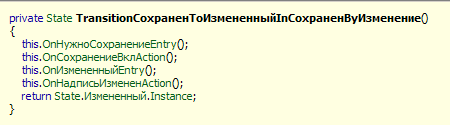
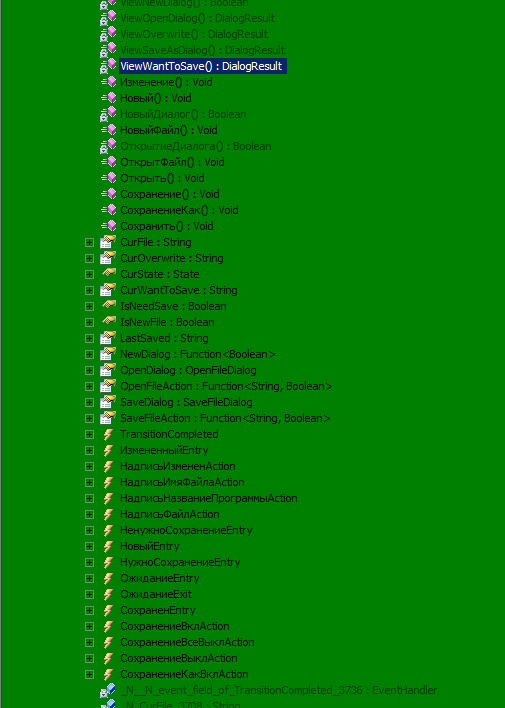
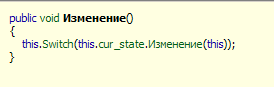
#>)]

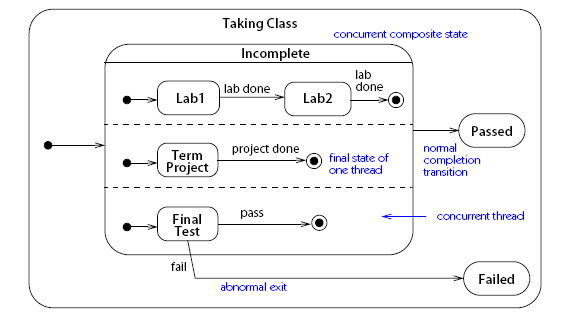
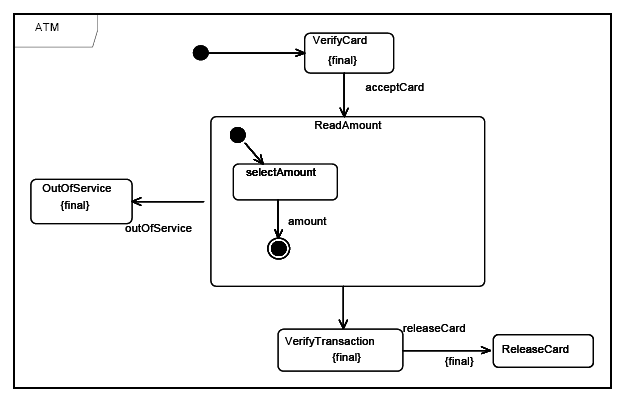
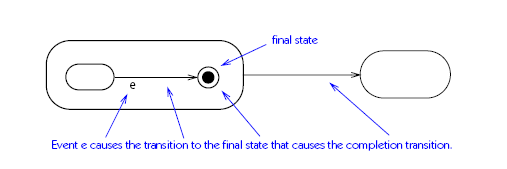
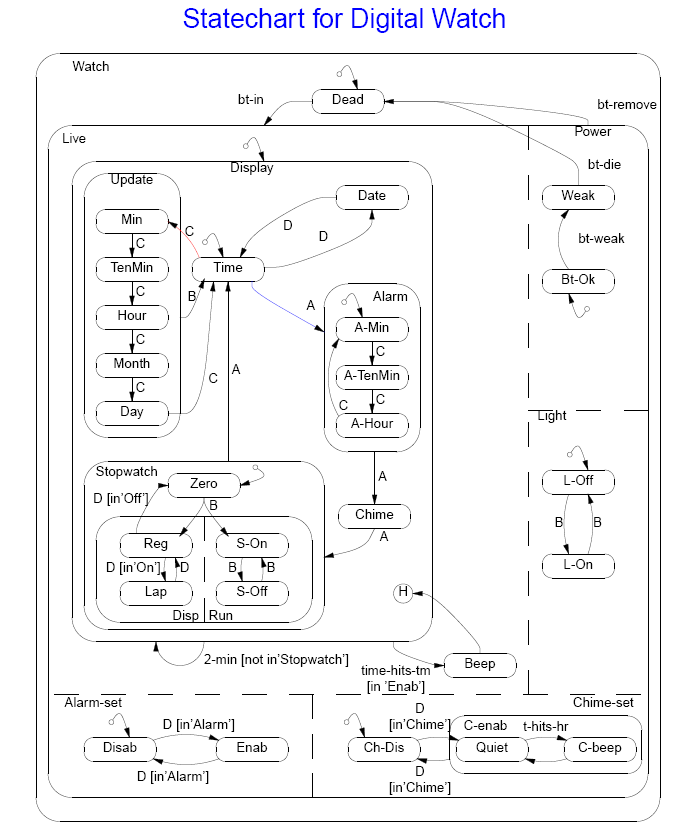
class GuiFsm

{

}

Дальше после компиляции, надо будет правильно подключить обработчики к действиям, для входных, выходных действий, связать событие в пользовательском интерфейсе с   
запуском событий автомата, привязать активности и сторожевые условия к объектам программы и пользовательскому интерфейсу и все. Вся эта сложная логика   
начинает работать, не знаю сколько бы ушло время на отладку без конечного автомата, но уж точно в несколько раз больше чем создание диаграммы состояний и перенос  
ее в описание автомата библиотеки. Как видите достаточно один раз переосмыслить логику на диаграмме и приложение может сразу перестроено на новый лад и расширять  
его будет удобно, можно при этом манипулировать большими сущностями поведения системы (состояниями и переходами), что дает сразу кумулятивный эффект в наращивании  
сложной функциональности.   
В будущем расскажу о других возможностях библиотеки, далее планируется работа с параллельными состояниями, которые дают еще один пласт мощи для написания сложных систем,  
но реализация их в библиотеке не так проста как обычных последовательных состояний, поэтому я трачу на них больше времени и отложил на реализацию после основных функций.  
  
ЗЫ: Если вы вклинитесь в теорию диаграмм состояний, то вам будет очень просто решать сложные задачи. Надеюсь эта библиотека поможет эффективно вам решить их.

Я пока не хочу особо широко распространяться о библиотеке сейчас хорошо бы протестировать ее и дать Nemerle сообществу как инструмент, может быть в будущем, посмотрим... Размещения в сниппеты будет более чем достаточно.  
  
Сегодня продолжу обсуждение возможностей библиотеки и автоматного подхода. Хочу заметить что используя автоматы можно моделировать объекты реального мира, в прошлый раз обсуждали модель телефона, таким образом можно моделировать что угодно, например: чайник, светофор, лифт, кодовая панель, телевизор, автомат продажи еды, банкомат, собака, вплоть до создания робота выполняющего разные действия. Программа тоже может являтся автоматом, gui реагирует на воздействия пользователя, кнопки должны включаться выключаться в определенной последовательности. Сегодня я покажу как можно автоматизировать часть логики приложения и gui на простом примере, этот пример есть в тестах библиотеки.  
  
Задача такова, всем известны приложения типа word, которые отслеживают изменения в файлах и не дают выйти из приложения если файл не был сохранен, gui при этом тоже реагирует и если файл открыт то его название пишется в заголовке окна, если изменен часто выводится \* после имени файла, такую логику можно пытаться писать влоб, но отлаживать такой код будет сложно, долго придется выслеживать ошибки, чтобы логика соответствовала, которые еще могут в будущем проявится неожиданно, можно нанять штат тестеров и исправлять логику, а можно пойти автоматным подходом и нарисовать диаграмму состояний, она должна выглядеть примерно так:  
  
  
Логика такова:  
  
1) При запуске приложения оно входит в состояние Ожидание через переход из начального псевдосостояния при входе в состояние выключаются кнопки сохранения, в заголовке отображается название программы  
  
2) Далее пользователь может либо открыть файл либо создать новый, если он открывает файл, то срабатывает переход из состояния Не Нужно Сохранение, по открытию и если файл открыт успешно, выходит из состояния Ожидание при этом запускается выходное действие включается кнопка сохранения, заходит в состояние Сохраненный через начальное состояние в Не Нужно Сохранение, при этом при входе в состояние Сохраненный в надписи начинает отображаться имя файла и запоминается последний сохраненный файл  
  
3) Далее пользователь может открывать новые файлы и логика будет повторяться если программа была в состоянии Сохраненный никаких предупреждений писаться не будет  
  
4) Далее пользователь производит некоторые действия в программе и изменяет данные так что они разняться с файлом это событие изменение, которое запускает переход из Сохраненный в Измененный при этом автомат выходит из состояния Сохраненный входит в Нужно Сохранение запускает действие при входе кнопка сохранения включается всегда в этом состоянии, поскольку файл нуждается в сохранении, при этом в надписи отображается звездочка (\*) рядом с именем файла что означает что данные изменены.  
  
5) В этом состоянии пользователь может принудительно сохранить файл тогда автомат перейдет в состояние Сохраненный, если он выберет Новый файл или Открыть файл, то программа ему сообщит что файл нуждается в сохранении и даст вопрос о сохранении, при этом пользователь может отвергнуть или согласится или отменить процедуру, если он выберет Новый Файл и соглашается на сохранение то данные сохраняются и создается новый файл, при этом автомат перейдет в состояние Новый Файл который также требует чтобы данные были сохранены, в надписи при этом отображается не имя файла а некое имя представляющее новые данные  
  
6) Если пользователь запросит выход из приложения то оно будет вести себя в зависимости от текущего состояния, если оно было сохранено то ничего не будет сказано, если было в состоянии Нужно Сохранение то будет запрошено сохранение  
  
7) Далее все повторяется в зависимости от того что делает пользователь в приложении  
  
Как видите логика легко описывается в рантайме если диаграмма продумана логика начинает работать сразу не требует никакой отладки и доводки, поиска ошибок, если что то не так работает то надо переосмыслить диаграмму и может стоить добавить действие при входе или перенести переход во внешнее состояние и все магически начинает работать  
  
На языке библиотеки подобный автомат можно представить так, более подробный пример смотри в тестах (когда будет принят пуллреквест):  
  
[statechart(<#  
  
// флаги auto\_initial автоматически добавляет в композитное состояние начальный переход в первое подсостояние  
// transition\_completed\_events — генерирует обработчик который оповещает о том что произошло событие (нужно при отладке)  
flags : auto\_initial transition\_completed\_events;  
  
0 => Ожидание; // начальный переход  
  
state НенужноСохранение  
{  
  
$> / СохранениеВыкл; // действие при входе  
  
НовыйФайл => НужноСохранение;  
ОткрытФайл =>@; // переход в себя  
  
state Сохранен  
{  
$> / НадписьИмяФайла ПоследнийФайлТек; // действие при входе 2 действия  
Сохранение =>@; // переход в себя  
Изменение => Измененный;  
}  
  
state Ожидание  
{  
$> / СохранениеВсеВыкл НадписьНазваниеПрограммы;  
$< / СохранениеКакВкл;  
}  
  
}  
  
state НужноСохранение  
{  
$> / СохранениеВкл;  
ОткрытФайл, Сохранение => НенужноСохранение;  
НовыйФайл =>@;  
  
state Новый  
{  
$> / НадписьФайл ПоследнийФайлПустой;  
}  
  
state Измененный  
{  
$> / НадписьИзменен;  
}  
  
}  
  
#>  
)]  
public class FileFsm  
{  
}  
  
Далее генерируется на основе описания класс автомата, который можно использовать во внешнем коде. Вот как выглядит класс в рефлекторе:  
  
  
  
  
  
Здесь показано тело класса Transition — это методы переходов они выполняют действия и выдают результирующее состояние,  
также там видны события для действий, чтобы автомат делал что либо полезное к его событиям подключают внешние действия, типа как вывод в строку заголовка окна,   
также есть стандартные методы определения состояния IsInState(st), Initiate() — запуск автомата  
  
  
  
  
здесь показаны вложенные классы состояний, они представляют из себя варианты, в них генерируются виртуальные методы обработки событий  
вот так выглядит тело одного из методов   
  
  
  
  
вот так выглядит функция перехода  
  
начальный   
  
  
  
обычный  
  
  
  
тело класса, события  
  
  
  
  
это тело запуска события   
  
  
  
метод Switch просто проверяет результат на null и присваивает текущей переменной cur\_state если произошел переход  
  
Вот вообщем и весь автомат, осталось в библиотеке реализовать семантику параллельных состояний, кое какие псевдосостояния и полезные функции, аттрибуты, флаги, опции, сейчас уже можно строить неплохие автоматы на ней.  
  
На этом обсуждение закончу, потом дам больше примеров и описаний, до новых встреч.

Выложил видео с примером работы приложения, автомата описанного в прошлый раз, видео [здесь](http://www.vimeo.com/27909233)  
  
Продолжаю писать примеры использования и описание библиотеки автоматов. На этот раз добавил функцию работы с параллельными состояниями, и хочу ее описать. Этой функции нет почти не в одной библиотеке, а она очень необходимая. Были реализованы следующие функции:  
  
— работа автомата в параллельных независимых состояниях  
— примитивы для разделения потока и слияния fork и join  
— разные варианты переходов в/из параллельных состояний  
— конечные состояния и переходы по завершению в параллельных состояниях  
  
Что же дают параллельные состояния? В отличие от обычных состояниях, в одном из которых может пребывать автомат, при входе в параллельное композитное состояние автомат входит сразу в несколько его подсостояний и пребывает в них одновременно. При этом он может выполнять некую параллельную активность в каждом из состояний и независимо реагировать на события и осуществлять переходы независимые от других параллельных регионов. На диаграмме параллельные области в состоянии показываются пунктирной линией:  
  
  
  
Здесь показан процесс обучения студентов, не важно в каком порядке выполняются задания, главное они должны быть все пройдены. Здесь при входе в состояние Taking Class он заходит в его начальное состояние Incomplete, оно является композитным то есть содержит другие состояния и параллельным, имеет 3 независимые области, при входе в это состояние автомат входит сразу во все его подрегионы, и будет пребывать в 3х состояниях Lab1, Term Project, Final Test, после появления событий lab done, он переходит из Lab1 В Lab2 также и в других регионах когда появится событие project done, pass (завершен проект, сдал тест) каждое состояние в подрегионе перейдет в конечное состояние, которое обозначается как черный кружок в белом (бычий глаз). Когда все три региона окажутся в конечном состоянии сработает переход по умолчанию (стрелка из состояния Incomplete без события) в состояние Passed, то есть студент сдал тест.  
Если автомат будет находится в состоянии Final Test и появится событие fail, то есть не сдал тест, то автомат выходит из состояния Final Test и из всех состояний в других регионах принудительно и переходит в состояние Failed.  
  
всегда при переходе на границу композитного состояния осуществляется переход в начальные состояние его областей. Возможен переход в одно вложенное   
параллельное состояние, при этом будет неявный вход во все остальные параллельные состояния через начальные переходы.  
  
Конечное состояние (final state), означает что композитное состояние завершило свою деятельность и может осуществлять переход по завершении, в параллельном состоянии имеет иную семантику чем в последовательном. В последовательных состояниях переход в final означает что состояние завершилось и если есть переход по умолчанию из верхнего состояния (completion transition), то он срабатывает. В параллельных этот переход сработает если все параллельные регионы находились в конечном состоянии, иначе регион будет ожидать завершения других регионов находясь в конечном состоянии.  
  
  
  
  
В этом примере показана работа электронного терминала. В состоянии selectAmount при возникновении события amount автомат осуществляет переход в конечное состояние и сразу переходит через переход по завершению в состояние VerifyTransaction.  
  
Ниже представлен расширенный пример еще одного автомата, настоящих электронных часов:  
  
  
  
Часы могут одновременно пребывать в нескольких режимах параллельно, здесь имеются разные состояния отображения Display, подсветка Light вкл или выключена L-On, L-Off, Chime выбор звонка, при этом могут быть как подсветка, так и разные режимы отображения, установки будильника Alarm-set, то есть любые независимые комбинации режимов. Это невозможно реализовать на последовательных состояниях, поскольку здесь требуются именно несколько состояний одновременно. Я не буду вдаваться в подробности автомата, скажу только что при запуске становятся активными все состояния в параллельных областях и они могут менятся в каждой из областей как реакция на события. Когда часы ломаются, он выходит из всех состояний в состояние Dead. Остальные аспекты я думаю будут понятны, если что то неясно читайте предыдущие сообщения или задавайте вопросы.  
  
Синтаксис:

state TakingClass

{

flags : auto\_initial; // показывает что первое подсостояние есть начальное

Extern => Lab2 TermProject FinalTest; // разделение fork переход сразу в три состояния

state Incomplete

{

\_ => Passed; // переход по завершении (нет триггера)

state Lab1

{

lab\_done => Lab2;

}

state Lab2

{

lab\_done => $0;

}

[----------] // определение границы параллельной области

state TermProject

{

project\_done => $0; // переход в конечное состояние

}

[----------] // определение границы параллельной области

initial => FinalTest;

state FinalTest

{

pass => final; // переход в конечное состояние

fail => Failed; // переход во вне

}

}

state Passed

{

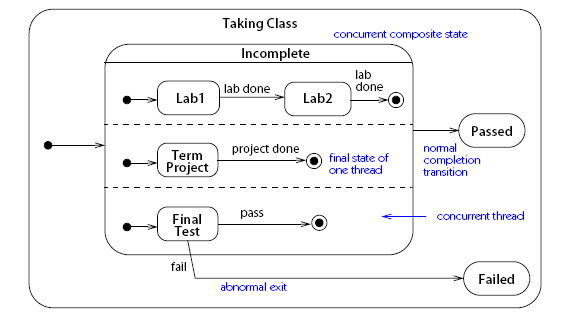
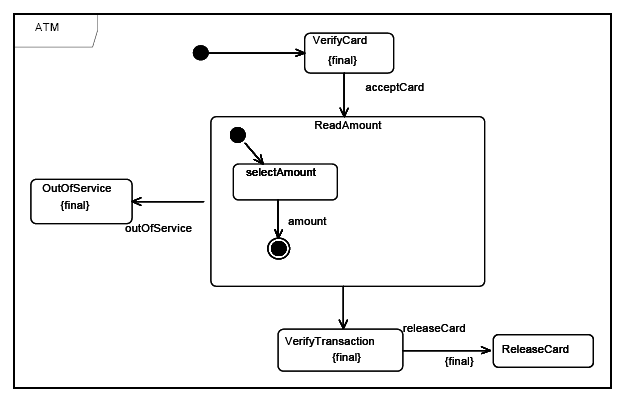
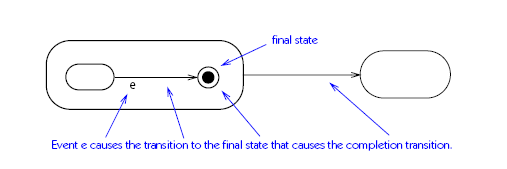
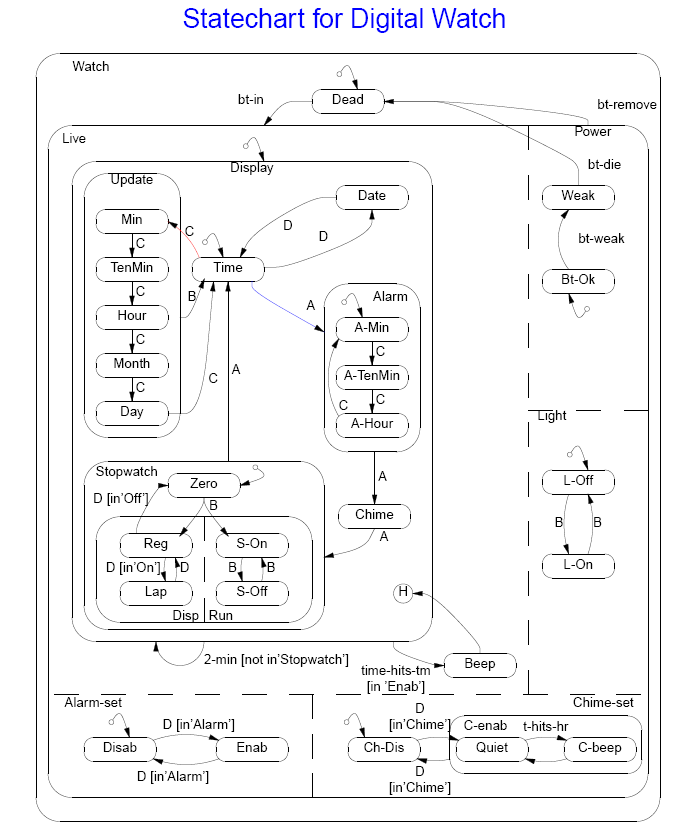
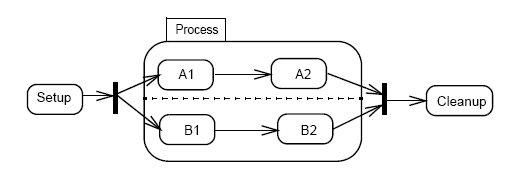
}

state Failed

{

}

}

[img]Продолжаю писать примеры использования и описание библиотеки автоматов. На этот раз добавил функцию работы с параллельными состояниями, и хочу ее описать. Этой функции нет почти не в одной библиотеке, а она очень необходимая. Были реализованы следующие функции:  
  
— работа автомата в параллельных независимых состояниях  
— примитивы для разделения потока и слияния fork и join  
— разные варианты переходов в/из параллельных состояний  
— конечные состояния и переходы по завершению в параллельных состояниях  
  
Что же дают параллельные состояния? В отличие от обычных состояниях, в одном из которых может пребывать автомат, при входе в параллельное композитное состояние автомат входит сразу в несколько его подсостояний и пребывает в них одновременно. При этом он может выполнять некую параллельную активность в каждом из состояний и независимо реагировать на события и осуществлять переходы независимые от других параллельных регионов. На диаграмме параллельные области в состоянии показываются пунктирной линией:  
  
  
  
Здесь показан процесс обучения студентов, не важно в каком порядке выполняются задания, главное они должны быть все пройдены. Здесь при входе в состояние Taking Class он заходит в его начальное состояние Incomplete, оно является композитным то есть содержит другие состояния и параллельным, имеет 3 независимые области, при входе в это состояние автомат входит сразу во все его подрегионы, и будет пребывать в 3х состояниях Lab1, Term Project, Final Test, после появления событий lab done, он переходит из Lab1 В Lab2 также и в других регионах когда появится событие project done, pass (завершен проект, сдал тест) каждое состояние в подрегионе перейдет в конечное состояние, которое обозначается как черный кружок в белом (бычий глаз). Когда все три региона окажутся в конечном состоянии сработает переход по умолчанию (стрелка из состояния Incomplete без события) в состояние Passed, то есть студент сдал тест.  
Если автомат будет находится в состоянии Final Test и появится событие fail, то есть не сдал тест, то автомат выходит из состояния Final Test и из всех состояний в других регионах принудительно и переходит в состояние Failed.  
  
всегда при переходе на границу композитного состояния осуществляется переход в начальные состояние его областей. Возможен переход в одно вложенное параллельное состояние, при этом будет неявный вход во все остальные параллельные состояния через начальные переходы.  
  
Конечное состояние (final state), означает что композитное состояние завершило свою деятельность и может осуществлять переход по завершении, в параллельном состоянии имеет иную семантику чем в последовательном. В последовательных состояниях переход в final означает что состояние завершилось и если есть переход по умолчанию из верхнего состояния (completion transition), то он срабатывает. В параллельных этот переход сработает если все параллельные регионы находились в конечном состоянии, иначе регион будет ожидать завершения других регионов находясь в конечном состоянии.  
Внутри одного региона параллельного состояния автомат живет в последовательных состояниях.   
  
  
  
  
В этом примере показана работа электронного терминала. В состоянии selectAmount при возникновении события amount автомат осуществляет переход в конечное состояние и сразу переходит через переход по завершению в состояние VerifyTransaction.  
  
Ниже представлен расширенный пример еще одного автомата, настоящих электронных часов:  
  
  
  
Часы могут одновременно пребывать в нескольких режимах параллельно, здесь имеются разные состояния отображения Display, подсветка Light вкл или выключена L-On, L-Off, Chime выбор звонка, при этом могут быть как подсветка, так и разные режимы отображения, установки будильника Alarm-set, то есть любые независимые комбинации режимов. Это невозможно реализовать на последовательных состояниях, поскольку здесь требуются именно несколько состояний одновременно. Я не буду вдаваться в подробности автомата, скажу только что при запуске становятся активными все состояния в параллельных областях и они могут менятся в каждой из областей как реакция на события. Когда часы ломаются, он выходит из всех состояний в состояние Dead. Остальные аспекты я думаю будут понятны, если что то неясно читайте предыдущие сообщения или задавайте вопросы.  
  
Также существуют псевдосостояния помогающие при работе с параллельными областями, поток перехода может разделится с помощью fork (разделение, слева от состояния Process) и объединятся при выходе из параллельных областей с помощью Join (два перехода объединяющиеся перед Cleanup) как показано на рисунке:  
  
  
  
При этом он может входить в любые подсостояния композитного параллельного состояния, переход запускающий переход в fork имеет одно событие запускающее переход, может иметь сторожевое условие и действие и идет стрелкой напрямую в псевдосостояние fork, (которое показывает жирной чертой,) из которого разветвляется на два и более перехода, отрезки не могут иметь сторожевых условий и идут напрямую в конечные состояния или псевдосостояния. Join антоним для fork, в него идут несколько стрелок переходов, которые срабатывают по одному событию, могут иметь сторожевые условия и действия, если оба перехода запущены, то есть появилось событие, и сторожевые условия истинны, то осуществляется переход в псевдосостояние join и далее через выходной переход в конечное состояние. При этом параллельное состояние завершается свою деятельность и становится неактивным.  
  
Я тут довавил fork переход в диаграмме обучения сразу в три состояния ко 2й лабораторке для определенного студента и выход из первый лабороторок, если студент выбыл.  
  
Синтаксис:

state TakingClass

{

flags : auto\_initial; // показывает что первое подсостояние есть начальное

Extern => Lab2 TermProject FinalTest; // разделение fork переход сразу в три состояния

join j1 => Failed; // создаем псевдосостояние join

state Incomplete

{

\_ => Passed; // переход по завершении (нет триггера)

state Lab1

{

lab\_done => Lab2;

leaved => (j1); // переход в псевдосостояние join j1 1 й отрезок

}

state Lab2

{

lab\_done => $0;

}

[----------] // определение границы параллельной области

state TermProject

{

project\_done => $0; // переход в конечное состояние

leaved => (j1); // переход в псевдосостояние join j1 2 й отрезок

}

[----------] // определение границы параллельной области

initial => FinalTest;

state FinalTest

{

pass => final; // переход в конечное состояние

fail => Failed; // переход во вне

leaved => (j1); // переход в псевдосостояние join j1 3 й отрезок

}

}

state Passed

{

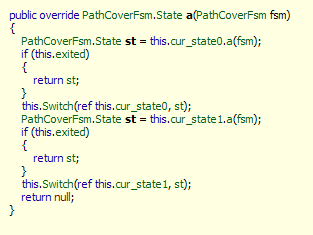
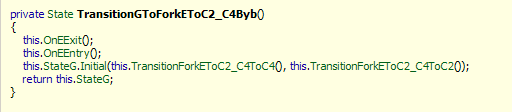
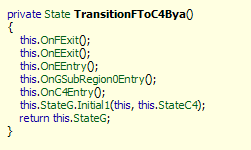
}

state Failed

{

}

}

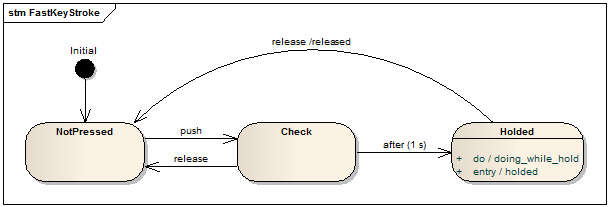
Сами же параллельные состояния реализованы в виде специальных классов, в иерархии дерева состояний, которые внутри себя содержат ссылки на несколько независимых переменных состояний:  
  
  
  
реакции выглядат так:  
  
  
состояния которые содержатся в параллельных ведут себя также как обычные  
  
Переходы выглядят несколько иначе чем для обычных состояний, входы и выходы из параллельных состояний задействуют другие состояния поэтому работают по другому:  
  
Переход в Fork:  
  
  
Переход сразу во внутреннее состояние:  
  
  
Но по сути и производительности они мало отличаются от обычных переходов.  
  
Осталось совсем немного доделать остальных возможностей, из них:  
  
— synch псевдосостояние (примитив синхронизации)  
— junction псевдосостояние, переход с ветвлением  
— choice/branch/merge узлы выбора  
— импорт и экспорт в диаграммы редактора  
— остальные доделки http://rsdn.org/Forum/Images/shuffle.gif   
  
Ждите статью и новых примеров. http://rsdn.org/Forum/Images/user.gif

Вышла новая версия библиотеки Nemerle.Statechart, залита в репозитарии Н, должна со следующей сборкой идти вместе с инсталлятором Н, расскажу об изменениях которые произошли. Была проделана большая работа по переписке движка, прочел кучу литературы по автоматам и диаграммам состояний UML, включая стандарт, все функции которые были исправлены или добавлены, привести не удастся, работа за два года, коммитов больше 500, скажу только об основных изменениях. Также для тех кто новичок и ничего не слышал о библиотеке или не знаком с областью конечных автоматов и диаграмм состояний UML я расскажу на примерах в нескольких постах о диаграммах состояний и их представлении в библиотеке и для чего она служит. Если что то упущу или не вспомню, в следующих примерах, и документации опишу последовательно все элементы и конструкции и их использование.  
  
Помимо функций которые уже были и работали, появились новые:

— Появилась полноценная система тестирования на основе Nemerle.TestFramework, написано больше сотни тестов как для обычных функций так и для тонких моментов и багов  
— добавлено много разных примеров автоматов и готовых графических приложений на их основе (Калькулятор, Кодовая панель, Часы с будильником)  
— сделана поддержка инлайна подавтоматов так называемые submachine, когда можно выделить шаблоны повторяющихся наборов состояний и вставить в определенном месте его копию  
— активные автоматы, основной тип, который используется, автомат прикрепляется к обьекту, у него есть свой поток управления, своя очередь событий  
— поддержка параллельных состояний претерпела изменения, сделана более эффективная работа с паралл.состояниями, введен синтаксис region A {} and {} для разных параллельных областей в состоянии A  
— поддержка fork и join псевдосостояний в переходах. В fork добавлен синтаксис A(B, C) предложенный одним товарищем, где A общий регион, B, C состояния в разных подрегионах  
— поддержка временных событий (timed events) через синтаксис after (n sec)  
— поддержка отложенных событий (deferred events)  
— события с параметрами, которые можно использовать в действиях тела перехода  
— в сторожевых условиях теперь можно использовать любые выражения Немерле  
— в действиях также можно вставлять любой код Немерле от небольших функций до большого кода  
— иерархия событий  
— автоматы могут обмениваться сообщениями между собой через единый канал передачи событий  
— сделан небольшой интерпретатор автоматов, помогающий при их тестировании, в нем можно устанавливать сторожевые условия, запускать события и смотреть за реакцией автомата, генерировать тест кейсы на основании входных и выходных данных автомата  
— поддержка входных и выходных точек (entry and exit points)  
— поддержка использования внешних классов событий  
— поддержка partial определений, когда автоматы из разных файлов с partial классами собираются в один  
— сделана привязка большинства ошибок к строкам, клик на сообщении об ошибке переходит к соответствующему элементу в описании автомата  
— сделано больше анализа на ошибочные ситуации и сообщений об ошибках и варнингов  
— компоненты отделены друг от друга, парсер, анализатор и генератор не зависят друг от друга  
— поддержка синстаксиса state A : parent когда любое состояние не обязательно вкладывать в родительское, можно определить где угодно и оно добавится в соответствующий родитель (parent)  
— сделан перемещаемый синтаксис для переходов, когда переход или группу можно описать в любом месте, с помощью синтакиса @A: a => B; и переход будет добавлен в состояние A  
— поддержка локальных переходов (local transitions), которые не выходят из начального состояния

На данный момент реализованы практически все функции из стандарта UML касательно автоматов состояний, остались некоторые редко используемые, но в будущем планируется полностью покрыть стандарт UML. Язык описания DSL и трансформация на этапе компиляции и генерация кода, позволяют сделать на основе описания поддержку любой сложной конструкции и сгенерировать эффективный код, выдавать сообщения об ошибках в случае ошибок в описании языка или семантике анализируемого автомата, такое на данный момент сложно реализуемо на других языках, сравнения по скорости с аналогичными автоматами сделанными на обычных библиотеках, которые строят автомат в рантайме, библиотека по скорости кода обходит как минимум в 7 раз, анализ и выдача ошибок тоже помогают качественному коду автомата. Сам язык автомата, декларативен, содержит лишь информацию касательно описания состояний, переходов событий, действий то есть того как описывается автомат из оригинальной диаграммы и легко перекладывается из диаграммы в код, не требуя усилий в виде сложных описаний в духе метапрограммирования шаблонов C++, или вызова методов для добавления переходов в других языках, с нулевой возможностью выдачи ошибок и очень низкой анализа. Текстовое описание возможно и без диаграммы, при этом создание не сложней чем рисовать диаграмму, никаких дополнительных конструкций кроме необходимой информации не нужно. На данный момент мало или практически нет нормальных генераторов кода для автоматов, поддерживающих все функции UML, потребность в них и заставила заняться этой задачей.   
  
Автоматами по стандарту UML могут снабжаться любые сущности, типа классов, описывая их поведение в четком детерминированном виде, когда все состояния объекта известны и скрыты внутри автомата, внутри каждого состояния объект выполняет какую то задачу (активность), переходы из состояния в другое состояние, наборы и последовательность эффектов при изменениях состояний определяются автоматом или диаграммой, по определенным событиям, при этом класс имеет свой поток управления, реагирует на внешние и внутренние события асинхронно, в каждом состоянии по разному, при этом каждое событие обрабатывается по одному в так называемом шаге до завершения (run-to-completion step) между двумя шагами автомат обрабатывает только одно событие, а после пребывает в стабильном состоянии, исключая порчу или рассинхронизацию своего внутреннего состояния, при работе из нескольких потоков. Все это есть реактивное программирование, когда программа не просто обрабатывает и трансформирует данные, но пребывает в нескольких состояниях, в течении разного времени, могут появляться различные события в любой момент времени, как то нажатие кнопок, временные интервалы, другие сигналы и обстоятельства которые случаются во времени и могут рассматриваться как события, на которые обьект должен реагировать, выполнять последовательность эффектов, изменять свое состояние управляемым способом и реагировать в каждом состоянии на разный набор событий, в таких случаях и стоит применять автоматы состояний и диаграммы, упрощающие их описание, надежность такого приложения возрастает, а отладка упрощается. Все описание поведения приложения, обьекта или какого то процесса может быть описано через состояния, набор эффектов при входах, выходах из состояний и при переходах. Примерами приложений в которых было бы полезно автоматное программирование можно назвать графический интерфейс, где он может пребывать в разных состояниях или режимах и по разному реагировать на действия пользователя, view-model в паттерне MVVM может описываться автоматом. Сама технология WPF может легко использоваться с автоматами, достаточно будет только послать события в автомат, подписаться на какие то действия, и автомат сам будет вызывать действия и эффекты в верной последовательности, все описание поведения может быть сконценрировано в автомате, чтобы изменить поведение приложения достаточно изменить описание или переосмыслить диаграмму. Также автоматы могут применяться в играх, где у обьектов мира бывает множество состояний, неожиданных событий в мире, которые могут появляться когда угодно, на которые они по разному реагируют и должны менять свое состояние, программирование таких сложных процессов через стандартную модель, даже если возможна событийность, без реактивного программирования, четкого описания в диаграммах может привести к трудности при отладке, куче багов и спагетти-кода. Также автоматы могут применяться в workflow процессах, робототехнике, других описаниях автоматных систем, типа разных приборов, имеющих взаимодействие с пользователем типа часов, телефонов, фотоаппаратов, автоматов выдачи валюты и тп. Моделирования обьектов реального мира, которые все изменяют свое состояние, описание протоколов связи и систем реального времени.   
Автоматы из библиотеки поддерживают работу со временными интервалами, через временные события, достаточно описать переход из одного состояния в другое через событие after (), можно использовать любые интервалы в секундах, миллисекундах, минутах, часах, днях, автомат сам будет следить за созданием, запуском таймеров, их распределением и оптимизировать чтобы таймеров использовалось по минимуму.  
  
В будущем я думаю сделать генератор не только для Немерле, но и для других языков, включая C/C++ это позволит использовать библиотеку для описания автоматов для систем реального времени и микроконтроллеров, всяких других устройств типа телефонов, планшетов и тп. Также хотелось сделать генератор в диаграммы и наоборот, простой рисовальщик диаграмм, который мог бы генерить код с помощью библиотеки. Также Найтра в будущем будет использоваться как основной генератор парсера, что улучшит текстовый парсинг и обработку ошибок синтаксиса.   
В следующих постах опишу на примерах работу с библиотекой, также вы можете сами посмотреть примеры которые есть в папке snippets\Nemerle.Statechart\Tests чтобы начать программировать с помощью библиотеки. Конечно хотелось бы чтобы появились люди которые могли бы потестить библиотеку в своих проектах и приложениях, дать комментарии, предложения и ошибки которые найдут. На английском форуме я также буду вести описание и составлять документацию, если кто то может помочь с переводом на английский тоже бы было неплохо.

|  |
| --- |
|  |

Сегодня я опишу несколько функций и начну с простых примеров, в старых постах я писал на обычных примерах, они давно уехали вниз и я хочу рассказать о новых функциях которых не было раньше.  
  
Начну с простого примера для тех кто не знаком с автоматами и их интерпретацией UML в частности. Опишу простой автомат, он понадобился мне когда надо было распознавать задержку кнопки и выполнять функцию в автомате если кнопка была нажата долгое время, в wpf есть RepeatButton но это совсем не то что мне было нужно и я написал простой автомат чтобы распознавать долгое нажатие кнопки.  
  
  
  
Суть его такова, у нас есть два состояния кнопки когда она не нажата и надо определить когда пользователь зажмет кнопку на какой то промежуток времени, с этого времени надо выполнять ускоренное действие, то есть не то которое используется по обычному клику, этот автомат используется в Alarm Clock примере для быстрого ввода времени при настройке, когда надо быстро прокрутить цифры до нужного числа, кликать постоянно сложно, а функцию кнопки можно использовать и для других целей. Итак у нас есть ненажатое состояние NotPressed, когда пользователь нажимает кнопку автомат переходит в следующее состояние проверки Check, если он быстро отжимает кнопку автомат переходит в состояние NotPressed и все возвращается к началу как будто был обычный клик. Если пользователь зажимает кнопку то после 1 секунды срабатывает временное событие и автомат переходит в состояние Задержан (Holded) что означает что клавиша задержана, и можно производить какое то действие если клавиша задержана. Далее если клавиша была отжата, автомат возвращается в начальное состояние и вызывает действие released что клавиша была отжата.   
Казалось бы простая задача, но ее решение может быть не столь простым, возиться с таймерами, вводить какие то флаги когда клавиша задержана и производить действия, при том надо будет распознавать когда отжали из состояния когда не была задержана и когда была, все это начало спагетти, а если логика бывает посложней тут и бывает лес флагов, if которые приходится муторно отлаживать и добиваться правильного функционирования, при том никто не гарантирует что ошибка не вскроется потом, а здесь мы имеем тривиальную диаграмму, которая переводится в такой же код на библиотеке и все поведение описывается в пределах диаграммы и терминах автоматов состояний, никакой ошибки быть не должно, поскольку все продумано по наглядной диаграмме изначально.  
На диаграмме имеются три **состояния (state)** в овалах с именами слева направа NotPressed, Check, Holded, состояния бывает простые и **композитные**, простые не имеют внутренних подсостояний, а композитные имеют, о них расскажу позднее, здесь у нас простые состояния которые достаточны для решения задачи. Начальное состояние выбирается с помощью **Начального псевдосостояния (Initial)**, который обозначен черным кружочком из которого идет переход в состояние, которое будет считаться начальным, с него автомат начинает выполнение. Псевдосостояние это фиктивное ненастоящее состояние, поскольку автомат не может в нем находиться в течении времени и должен сразу перейти из него дальше. Состояния соединены между собой **переходами (transition)**, переходы осуществляются по **событию (event)**, события обозначаются словом рядом с переходом. В стандарт UML помимо простых событий входят **временные события (timed events)**, самое распространенное это after, событие которое отсчитывает время от входа в состояние из которого осуществляется переход, если автомат успевает перейти в другое состояние, временное событие не сработает, в данном случае у нас после 1 секунды сработает событие after и автомат перейдет из Check в Holded. У состояний есть внутренние действия и активности, есть действия которые вызываются при входе в состояние, **входные действия (entry)**, **выходные действия (exit)**, обозначаемые соответствующими ключевыми словами с косой чертой / за которой идет имя действия или выражение на языке, что делает автомат при входе или выходе. В данном случае у нас есть действие holded которое вызывается когда автомат входит в состояние Holded. Входные действия гарантированно вызываются при входе в состояние до всех остальных действий и активностей внутри состояния, поэтому подходят для вызова каких то инициализирующих процедур, перед работой самого состояния, при выходе могут вызываться действия, они вызываются после всех других действий и активностей, и могут использоваться для выполнения очистки ресурсов и других действий при выходе из состояния. В состоянии Holded также есть **внутренняя активность (activity)**, которая обозначается ключевым словом do с косой чертой и именем действия. Активности от действий отличаются тем что могут выполняться долго в течении большого промежутка времени, они выполняются в своем потоке параллельно другим действиям, они начинают работать после входных действий которые выполняются до завершения, то есть синхронно, и автоматически завершаются если состояние перестает быть активным, если активность завершилась то порождается **завершающее событие (completion event)** и **могут выполняться переходы по завершению (completion transition)** но об этом позднее. В данном случае у нас просто активность которая выполняется все время пока автомат в состоянии Holded.   
Также из состояния Holded идет переход по событию release, когда пользователь отжимает кнопку, к нему также прикреплено действие released, обозначается через косую черту после события прикрепленного к переходу, действия при переходах выполняются после выходных действий состояния из которого выходит переход. Это действие должно вызываться когда у нас произошло отжатие и до этого кнопка была зажата то есть автомат находился в состоянии Holded.   
У меня к действиям автомата holded и released прикреплены посылка событий hour\_holded и hour\_released и такие же для минут, они посылаются в автомат Будильника, где он обрабатывает их дает быстро изменять время.  
Из диаграммы переведем автомат в код, этот пример есть в примерах под названием FastKeystroke:

[statechart(

<#

state NotPressed

{

push => Check;

}

state Check

{

after (1 s) => Holded;

release => NotPressed;

}

state Holded

{

entry / holded;

release / released => NotPressed;

do / doing\_while\_hold;

}

#>)]

public class FastKeystroke

{

public NeedCancel : bool

{

get

{

IsInState(StateHolded)

}

}

}

Как видите все наглядно, состояния описываются начиная с ключевого слова **state Name {}** с именем состояния далее в фигурных скобках идет описание тела состояния, переходы, действия и активности. Переход обозначается как **событие => состояние;** также вписываются действия при переходах **событие / действие => состояние;**  
все остальные элементы описываются аналогично как в UML, entry / holded; входное действие, **do / doing\_while\_hold;** активность внутри состояния.  
Когда автомат компилируется он при разрешении имен действий и активностей наблюдает за методами, если есть метод с таким именем, автомат использует его и привязывает к соотв. действию, если метода с таким именем не нашлось библиотека генерирует для него событие, которое запускается когда выполняется соотв. действие внутри автомата, к действию можно позднее прикрепиться из внешних классов. В данном случае я оставил действие holded на него идет подписка извне, также и для активности, для них можно создать методы внутри класса при этом активность будет иметь такую сигнатуру:

doing\_while\_hold(tok : CancellationToken) : void

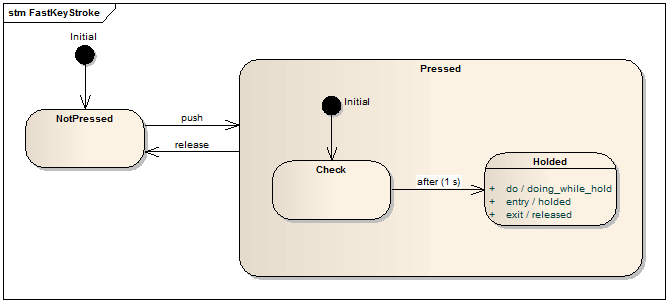
{

while (!tok.IsCancellationRequested)

{

}

}

в качестве параметра передается CancellationToken он нужен для автоматического завершения активности, если состояние перестанет быть активным и в методе надо обрабатывать флаг IsCancellationRequested, сами же do активности реализованы с помощью Task ов, облегченный вариант потоков, они идеально подходят для активностей. Также я добавил свойство IsCancel с методом IsInState(StateHolded), оно нужно для того чтобы знать что нужно отменить событие click при отжатии, чтобы у кнопки не сработал обычный клик, обрабатывается уже в wpf, IsInState это метод который генерируется для каждого автомата, проверяет что автомат находится в соотв.состоянии, StateHolded это публичное статичное поле, в котором лежит экземпляр класса состояния, к ним можно обращаться извне, IsInState используется также в сторожевых условиях, но об этом позднее.  
  
Я хочу провести некоторый рефакторинг автомата, у автомата наблюдается некоторая повторяемость, которую можно упростить переход по событию release идет как из Check так и из Holded почему бы нам не обобщить их, назовем общее состояние для них обоих как Pressed, и из него один переход release:  
  
  
  
Теперь у нас иерархичный автомат, появилось состояние Pressed у которого два подсостояния Check и Holded, при этом Check начальное состояние Pressed в него он переходит сразу после события push. Когда активно внутреннее состояние, активно и родительское состояние, иерархичные автоматы это разновидность подхода придуманного Давидом Харелом и позднее обобщенного в стандарте UML, они позволяют описывать более сложное поведение и структурировать автомат, при этом соблюдается LSP Liskov Substitution Principle, внутреннее состояние обладает чертами родителя и реагирует на события родителя, в данном случае событие release запускает переход как из Check так и из Holded, два перехода мы описали через один, иерархичные состояния позволяют и обобщить повторяющиеся конструкции, что упрощает дизайн автомата. Я убрал действие из перехода по событию release в выходное действие Holded. Описание автомата в библиотеке осталось таким же декларативным:

[statechart(

<#

state NotPressed

{

push => Check;

}

state Pressed

{

release => NotPressed;

state Check

{

after (1 s) => Holded;

}

state Holded

{

entry / holded;

exit / released;

do / doing\_while\_hold;

}

}

#>)]

public class FastKeystroke

{

public NeedCancel : bool

{

get

{

IsInState(StateHolded)

}

}

}

появляется состояние Pressed, в которое вкладываются подсостояния Check и Holded. В родительском остается переход release => NotPressed;  
Также возможен синтаксис : Parent этот синтаксис применяется когда большая глубина вложенности и позволяет упростить описание до такого:

state Pressed

{

release => NotPressed;

}

state Check : Pressed

{

after (1 s) => Holded;

}

state Holded : Pressed

{

entry / holded;

exit / released;

do / doing\_while\_hold;

}

здесь появляется описание родительского состояния Pressed через : после имени состояния, такое описание похоже на описания классов и будут понятны всем кто пишет в стиле ООП.   
После компиялции автомата, будет сгенерирован класс содержащий все обьявления необходимые для его работы. Автомат после создания через конструктор необходимо запустить через метод Initiate() это переводит автомат в начальное состояние, далее события можно посылать через специально сгенерированные методы по именам событий например fsm.push(); fsm.release(); их можно прикрепить к кнопке для событий например PreviewMouseDown и PreviewMouseUp в WPF, также привязаться к событиям действий holded, released чтобы выполнять действия при начале когда кнопка задержана и при отпуске. Я использовал для них посылку событий в другой автомат Будильник. Также можно запускать события через метод PostEvent(), в него подается экземпляр Event, базового класса для событий внутри этого автомата, он генерируется внутри и его экземпляры можно создать, бывают события с параметрами о них расскажу позднее. Для временных событий генерируется таймер System.Timer он стартует при входе в состояние и останавливается при выходе, при переполнении запускает метод события. Методы событий складывают его экземпляр в пул событий который выполнен на ConcurrentQueue и сразу же выходят, пул обрабатывает события в отдельном потоке по одному, для обеспечения run-to-completion step, в него можно посылать события из любых потоков, пока автомат не выполнит одного события до завершения включая все действия при переходе в другое, он не может выполнять и реагировать на другие события.   
Ну вот и все что я хотел рассказать на сегодня, если что то непонятно пишите. В данном посте рассказал о простых автоматах, состояниях, переходах, действиях при входе, выходе, активностях, об этом я и раньше рассказывал, новыми были временные события, : parent синтаксис и реализация активностей. Дальше расскажу об остальных функциях.