**Да се обмисли издаването на поредица статии**

1. Статия за кодогенерация не една списъчна обработка (клас) без разклонения.
2. Статия за кодогенерация на списъчни обработки с разклонения – реализирани на предходното описание но разклонения описани посредством с автоматен граф.
3. Статия за реализация на точка 1. (ако решим да разпространяваме ноу хауто)
4. Статия за реализация на точка 2.
5. Статия за генерация на потребителски интерфейси, интегрирана с точка 2.
6. Статия за генерация на уеб услуги интегрирана с точка 2.
7. Статия за генерация на навигационно дърво и управление на права на потребителите, интегрирана с точка 2 и 6.

**Увод**

ООП позволява да се правят описания на обектите, при които поведението на обектите не е заложено в самите обекти, а е заложено в интерфейсите, които тези обекти имплементират. Имплементацията на тези интерфейси може да бъде извършена автоматизирано, когато интерфейса е достатъчно добре описан като поведение, достатъчно гранулиран (прост) и описан като шаблон. При подаване на метаданните на обекта в кодогенератора на интерфейса е възможно постигането на пълна автоматизация на програмирането на обектен език. За целите на реализацията е избрана .net платформата на Microsoft с език C#, тъй като това е езика който в най-голяма степен съответства на идеята за унаследяване на поведение на обекти посредством интерфейси, както и единствения от широко разпространените обектни езици, който не позволява полиморфизъм.

В тази статия ще бъде описан кодогенератор на .нет на базата на подготвени шаблони на интерфейси (методи) за обработка и описания на данните.

Графичното описание на сигналния граф ще представлява диаграма от правоъгълници (функционални модули) с набор от визуализирани входове и изходи (поименно) и връзки между тях, визуализирани със стрелки.

Обектите ще се описват с мета-данни в XML формат, или директно с код.

Всяка инстанция на сигнален граф ще представлява клас, който се генерира автоматично на база на описанията на обектите с които този клас ще работи. Обектите с които класа работи, ще бъдат неговите член-променливи, а на база на функционалните модули които го изграждат ще се генерира неговото поведение (методи). Предаването на данните между методите ще се определя от връзките в сигналния граф.

Шаблоните на кода ще бъдат представени като функционални модули. Всеки функционален модул ще взаимодейства с останалите функционални модули като разполага с набор от входове (динамично променящи стойността си по време на работа на системата), параметри (входове с предварително зададена начална стойност), вътрешни състояния и изходи.

Всеки вход ще приема референция към обект от стойностен или референтен тип. За всеки вход ще може да се задава дали ще работи с копие от получените данни или ще има право да променя оригиналните входни данни. Всеки вход ще има маркер за нуждата от активация. Активация на входа се получава тогава, когато на входа са постъпили валидни данни и предхождащ го блок от сигналния граф е изпълнен успешно в рамките на текущото обхождане на графа. Повече по тази тема в секцията обхождане на графа.

Параметрите на функционалния модул ще бъдат задавани от програмиста по време на създаване на конфигурацията. Техните начални стойности ще бъдат инициализирани по време на първоначално създаване на инстанцията на функционалния модул. Те могат да бъдат променяни по време на работа на системата, като към тях бъдат свързани изходи на други функционални модули. Параметрите не носят маркер за активация.

Вътрешните променливи на всеки един функционален модул ще се създават, тогава, когато е необходимо конкретния функционален модул да има вътрешна памет за това, какво се е случвало при предишни изпълнявания на модула. Вътрешните състояния на модулите ще бъдат създавани като статични променливи в рамките на метода.

Входовете и изходите на функционалните модули ще са свързани с връзки визуализирани със стрелки.

**Автоматични валидации и връзки между блоковете**

Входовете на всеки блок имат стандартно описание. Ако входовете са от стойностен тип, то при създаване на връзка се изисква да има съвпадение между типа на данните на свързваните вход и изход. Ако входа е от обектен тип той приема референция към обекта, а типа на обекта се определя по време на генерация на системата от получените мета-данни.

Връзки между блоковете могат да се правят, като визуално връзките започват от изход на функционален модул и достигат до вход на друг или на същият функционален модул. Ако се изгради връзка от изход на функционален модул до вход на същият, то не се допуска този вход да получава активиращ маркер.

**По отношение на разпространението на данните има два възможни подхода:**

1. Връзка 1 към 1 (изход към вход). При такъв тип връзка не се създава нова инстанция на обекта, а се предава референция към оригиналния обект и следващият функционален модул променя стойността на обекта. Допустимо е дори при връзка 1 към 1 входа на приемащият модул да бъде зададен като вход който копира данните за обекта, в този случай независимо че връзката е от тип 1 към 1 данните ще бъдат копирани и няма да се промени оригинално подаденият обект.
2. Връзка 1 към много. При такъв тип връзка оригиналния обект остава непроменен, а се предават само копия от данните.
3. Връзки между различни графи – възможно е изход на функционален модул на изпълнен вече граф да бъде използван за инициализиране на параметричен вход на функционален модул от друг граф. (Да се опише обща концепция за наличие на множество графи и връзките между тях) Необходимо е входа да е от параметричен тип, за да се гарантира инициализирането на данните на входа в случай че графа източник по някаква причина не е бил изпълнен.
4. Не се допуска връзка на един вход с повече от един изход на друг модул.

**Всеки вход/изход има т.нар. мета-описание на данните:**

Всеки вход ще има мета описание – стандартен обект който ще дава допълнителна информация за състоянието на данните които се пренасят по него – валидност, времеви маркер на последна промяна и др. Такъв тип допълнения към данните може да съдържа информация за реализация на одит лог на работата на системата. Допустимо е и с цел повишаване на производителността на генерираната система разпространението на мета-данни да бъде изключено. (Да се дефинира и опише точно какво представлява мета-описанието).

Тук да се опише точно какви ще бъдат метаданните, които ще вървят по всеки канал.

**Валидност на сигналния граф**

Сигналния граф подлежи на предварителна проверка за валидност и изпълнимост. За да бъде валиден е необходимо да се направи обхождане на графа, като се проверява разпространението на активиращите маркери на функционалните модули. Всеки един функционален модул от графа трябва да може да бъде изпълнен, за да се счита графа за валиден. Всеки граф трябва да има поне по един функционален модул който няма активиращи входове. Графа се обхожда посредством следния алгоритъм:

1. Намират се всички модули, които нямат нито един активиращ вход. Счита се че тези модули са предвидени за изпълнение при стартирането на изпълнението на графа.
2. Добавят се избраните модули в опашка предназначена за модулите които има готовност за изпълнение.
3. По реда в който са добавени в опашката всички модули се маркират като изпълнени. Изходите на изпълнените модули се маркират като инициализирани.
4. За всеки един вход на функционален модул който е свързан към вход който е бил инициализиран на предходната стъпка се включва активационния маркер.
5. Избират се всички модули които не са изпълнявани до сега и имат пълен комплект от активационни маркери на входовете си (всеки един вход маркиран като активиращ е получил активационен маркер).
6. Ако има модули, които са били избрани на предходната стъпка се преминава към стъпка 2.
7. Ако няма такива модули се проверява дали всеки един модул от графа е бил изпълнен.
8. Ако няма неизпълнени модули значи графа е изпълним и може да се премине към кодогенерация.
9. Ако има неизпълнени модули значи потребителя не е свързал всички активиращи входове и следва да получи съобщение за грешка, заедно с описание на всички открити грешки.

**Изпълнение на сигналния граф**

Сигналния граф се изпълнява визуално от ляво на дясно, като изпълнението започва от модули които нямат входове очакващи активация.

За всеки модул на базата на предварително зададен шаблон за кода ще получава инстанция на данните и допълнителни параметри, по които да прави обработките. Данните ще се генерират в отделен файл, като това ще позволи на кода генериран от генератора да се интегрира в по-сложни проекти.

Посредством използване на подхода за кодо-генерация се решават няколко сериозни проблема:

1. Синтактична валидация на кода – генерирания от системата код ще бъде импортиран в средата за разработка (Visual Studio). Там ще бъде извършена синтактичната валидация, преди фазата на компилиране.
2. Компилация – Директно ще се използва .нет компилатора.
3. Дебъг – тъй като генерирания код ще бъде създаден на базата на шаблони, той ще бъде много лесно четим от човек, което ще даде възможност да се използва целия наличен към момента инструментариум на Visual Studio за изпълнение на програмата стъпка по стъпка.
4. Автоматизирано тестване – на база на изградената платформа ще бъде изключително удобно да се генерират тестови сценарии и да се използват всички налични инструменти на Microsoft и както и на трети страни. Използването на една и съща среда за генерация на кода и на тестовете ще позволи да се осигури висока степен на стабилност, отказоустойчивост и обратна съвместимост на генерираните системи.

Изпълнител – да има готов проект (подготвен за наливане на файловете), който да може да се компилира през командния ред и да се изпълни директно.

**Описания на структурите от данни:** За сега в XML или директно в код, да се обмисли имплементация на UML клас диаграма.

**Описания на минималният набор от функционални модули за реализация на прототип на система за кодогенерация на програми за обработка на списъчни данни:**

**Генератор на инстанции** – генератор на инстанция на списъка – входен параметър ще му бъде файла с описанието на списъка – Блок който чете схемата на данните и на изхода си подава празен списък (инстанция), който може да се използва за база за попълване от парсер или друг вид входен модул.

**Парсер** – за сега от текстови файлове, в последствие може да се разшири и към бази данни, с подходящо описание. Парсера трябва да получава схемата на парсваните данни. Може да обработва за сега XML, csv, json. Като вход парсера трябва да получава инстанция на списъка който да се попълва (парсва) заедно с метаданните на списъка. На базата на тези метаданни ще се извършва самото парсване.

**Филтър** – по избрано поле или комбинация от полета. Отново на база на мета-данните ще се извършва генериране на код за извършване на филтрацията. Ще създава нови обекти.

**Групер** – Ще създава групи от списъци (също списъци сами по себе си), като ще групира по произволни критерии, т.е. по всичко което може да се групира. Ще създава реално нови обекти, т.е. нови списъци. Това да се обмисли дали е за първи етап – май не е.

**Сорт** – по избрано поле или поредица от полета. Отново на база на мета-данните ще се извършва генериране на код за извършване на сортирането. След сортиране ще се получава същият списък подреден.

**Брояч –** брояч на елементите в списъка (всички или отговарящи на определени условия).

**ФорИч** – Някаква предварително заложена обработка.

**Find** – търсене. Изходи буул и обект.

**Изход** – генератор на изходен файл – csv, XML, json, генератор на заявки към бази данни.

**Използване на вложени графи**

Тъй като всеки един граф ще представлява пълно описание на клас реализиращ определено поведение, е напълно възможно да се реализира клас който да бъде използван като функционален модул и добавен в библиотеката на функционалните модули. Такъв тип клас ще се нарича функционален блок. Когато един граф се превръща във функционален блок е необходимо да се добави нов конструктор на класа, който да описва какви са входните данни, които този клас ще получава, както и да се добавят свойства, които да експортират резултатите от изпълнението на функционалния блок като изходи. Когато в библиотеката на функционалните модули бъде добавен такъв функционален блок, той ще може да бъде маркиран като функционален модул от съставен тип (т.е. да се генерира всеки път когато се генерира графа който е включил този блок) или генерираният код ще бъде запазен като шаблон, потенциално достъпен за ръчна редакция на кода. Допустимо е един функционален блок да няма явни входове или явни изходи – в този случай резултата от неговото изпълнение ще бъде достъпен за други модули, посредством база данни, файл, споделена памет, комуникационен протокол или по друг начин за предаване и обмен на данни.

**Реализация на шаблоните за функционалните модули.**

Всички шаблони на функционални модули ще бъдат изграждани по стандартен шаблон за изграждане на метод. Всички входове и параметри от стойностен тип ще бъдат предавани през името на модула.

1. Входове и параметри– ако един вход е маркиран като вход от референтен тип стойността му се получава и използва директно в кода на модула. Ако е изрично отбелязано че е необходимо да се копира модула с цел непроменена на оригиналния вход се извършва автоматична генерация на код за копиране на обект от съответния тип.
2. Вътрешни променливи – те се създават при първото изпълнение на графа и се запазват до края на работата на системата. Тяхната промяна е възможна само от кода на конкретния модул.
3. Изходи – предават се през името с допълнителен параметър out. По време на изпълнение на модула всеки един изход трябва да бъде инициализиран.
4. Грешки – всеки един функционален модул ще връща стандартизиран тип съобщение, което ще бъде празно в случай че модула е изпълнен безпроблемно и ще има съдържание носещо информативно описание на грешката, както и изключението което я е породило, ако има такова. Тък като по време на кодогенерация всеки един модул ще връща резултат от изпълнението си, ако по време на рабата на системата бъде засечена грешка ще се активира подсистемата за обработка на грешки.

**Пълна кодогенерация Да се опише алгоритъма.**

Може би е време да обърна малко внимание на разработката и да опиша алгоритъма след като успея да го реализирам.

**Описание на системата**

Всяка генерирана система се състои от два вида графи

1. Трансформационен сигнален граф – описва трансформациите на данните, служи за описание и взаимодействие с физическите или логическите входове и изходи на системата.
2. Автоматен граф – работи само с логическите входове/изходи на системата, като в него се залага логиката, последователността от действията, разклоненията в зависимост от условията и обработката на грешки.

Типове входове на системата:

1. Текстови вход – директно се парсва на базата на посочен тип.
2. Вход тип връзка ( – получава се

**Реализация на програмен генератор**

За реализация на описаният програмен генератор се използва контролата Telerik Diagrams, а като помощен пакет за реализация на спомагателните потребителски интерфейси е използвана и останалата част от WPF пакета от контроли на Telerik (Progress). Избраната контрола е част от платен пакет, който дава възможност за безплатно използване в пробен период за неограничено време, като в случайни моменти генерира съобщение че се използва без да е лицензиран.

Предимствата на тази контрола са:

* Добрата и изчерпателна документация;
* Практически неограничените възможности за реализиране на промени в поведението на елементите на използваната диаграма;
* Възможността за създаване на библиотеки от графични елементи които могат да запазват вътрешно своите спомагателни атрибути необходими за работата на генератора;
* Възможността за запазване на диаграми в отвореният XML формат, както и последващото им зареждане;
* Възможност за създаване на всички необходими за работата на генератора връзки, и графични елементи.

За реализация на функционалните модули се използват следните спомагателни контроли:

* Shape – графичен елемент с който се визуализира функционален модул. Размера и формата на графичния елемент могат да бъдат динамично управлявани. Всеки графичен елемент подлежи на автоматично оразмеряване така че да бъдат визуализирани кратки описания на всички негови входно-изходни конектори.
* Connector – Графичен елемент който позволява създаване на връзка. За да може да бъде изградена връзка към обект от тип Shape, този обект трябва да има поне един конектор. Ако бъде създаден модул без конектори, то към него не може да се изгради връзка (все пак създаването на такъв тип модули има смисъл).
* Connection – графичен елемент свързващ два конектора. Всяка връзка завършва със стрелка сочеща посоката на предаване на данните.

Всеки функционален модул има четири области за визуализация на връзките си – лява за входовете, дясна за изходите, горна за параметрите и долна за вътрешните състояния. Тези области се оразмеряват автоматично щом бъдат попълнени със съдържание и това позволява автоматичното оразмеряване на целият елемент отговарящ за функционалния модул. При създаването на нов функционален модул автоматично се отваря спомагателен прозорец в който потребителя да зададе основните му параметри – име на шаблона на база на който се генерира модула, брой входове, параметри, изходи, вътрешни състояния, както и да именува и при нужда добави допълнителни описания за всяка връзка.

Характеристики на входно-изходните точки.

Всички от по-долу описаните характеристики биват могат да имат следните атрибути:

* Уникален – тази характеристика трябва задължително да фигурира в описанието точно веднъж.
* Задължителен – тази характеристика трябва да фигурира поне веднъж.
* Незадължителен – тази характеристика може и да не фигурира.
* Опционален – този тип характеристика не може да бъде описан като задължителен или не предварително. Тя може да стане задължителна в случай, че някоя от другите описващи характеристики има определена стойност.

Всички входно-изходни точки на функционалния модул имат следните общи характеристики:

* Тип на данните (някой от основните типове данни или съставен) (Уникален)
* Връзка – описание с коя инстанция на кой модул на кой изход е вързан входа. (Незадължителен – конфигурацията генерирана от потребителя може да бъде приета и запазена без да са изградени всички връзки. При липса на критични връзки потребителя ще получава съобщение при заявка за кодо-генерация).
* Размерност – в случай че предаваната величина е двумерна (двумерен масив) и размера ѝ е статичен може да се опише размерността, така че тя да подлежи на автоматична валидация при свързването на входа (Задължителен – по подразбиране с размерност 1 на 1 – множество с един елемент).
* Име на входно-изходната точка – описателно име което да подсказва на потребителя на модула какви данни се очаква да бъдат използвани. (Уникален)
* Получени метаданни – различни метаданни носещи допълнително описание на получения обект (подават се по време на генерация на системата и служат за офлайн валидация на генерираната конфигурация. Не се задават ръчно от потребителя) (Незадължителен).
* Изисквани метаданни – метаданни описващи изискванията към получавания обект с цел автоматична валидация по време на създаване на генерираната система (Опционален – типа на данните е стойностен може да бъдат или да не бъдат описани допълнителни изисквания към получаваната стойност. Ако типа на данните е обектен, е задължително да се направи валидация че получавани тип обект има минимално необходими атрибути за правилното изпълнение на модула).

В спомагателния прозорец ще се визуализират пет секции:

* Основна секция
  + име на библиотечния модул – с възможност за избор от йерархична структура на библиотеката на функционалните модули
  + версия на библиотечния модул – списък с наличните версии на конкретния модул (особено важен при нужда от осигуряване на съвместимост с по-стари версии на генерираните системи)
  + име на инстанцията на функционалния модул – свободен текст
  + Описание – място за описание на модула от библиотеката – текстiz
  + Допълнително описание – място за допълнителни бележки от страна на създателя на конфигурацията.
* Секция входове – списък с подробно описание на всички входове на модула. Входовете могат да имат следните специфични атрибути:
  + Активиращ/пасивен – атрибут на входа описващ ролята му по отношение на определяне на реда на изпълнение на сигналния граф.
  + Стойност/референция – атрибут показващ дали данните получени на входа ще се използват директно или ще се работи с тяхно копие.
* Секция параметри – списък с подробно описание на всички параметри на модула
* Секция вътрешни променливи – списък с подробно описание на вътрешните променливи
* Секция изходи – списък с подробно описание на изходите на модула. Изходите могат да имат следните специфични атрибути:
  + Свързан/несвързан – показва дали конкретния изход може да бъде свързан към повече от един обект или не. Ако изхода е свързан само към един вход, то евентуална промяна на данните извършена от модула към който изхода е свързан би могла да бъде видима и използвана от текущия ФМ при следващото му изпълнение.

Запис на съществуваща диаграма (конфигурация). За запис на изградена от потребителя диаграма се използва възможността за сериализиране и десериализиране (определение за сериализиране?) на диаграмата предоставена от използваната контрола. Съхраняването и възстановяването на графичната част на диаграмата се извършва посредством директно предоставените от производителя на контролата възможности. За съхраняване и възстановяване на данните на генератора е необходимо да се реализират алгоритми за съхраняване и зареждане, които да дадат възможност за пълно записване и възстановяване на данните въведени от потребителя.

За съхранение на данните на модулите е използван формат XML, като е дефинирана следната схема:

<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:complexType name="Method">

<xs:sequence>

<xs:element name="MethodName" type="xs:string" minOccurs="1" maxOccurs="1" />

<xs:element name="returnValue" type="DataType" minOccurs="1" maxOccurs="1"/>

<!-- Used only in case the upped data type is of type object-->

<xs:element name="ClassName" minOccurs="0"/>

<xs:choice maxOccurs="unbounded">

<xs:element name="Member" type="Member" minOccurs="0"/>

</xs:choice>

</xs:sequence>

</xs:complexType>

<xs:complexType name="Member">

<xs:sequence>

<xs:element name="MemberName" type="xs:string" minOccurs="1" maxOccurs="1" />

<xs:element name="DataType" type="xs:string" minOccurs="1" maxOccurs="1" />

<!-- Used only in case the upped data type is of type object-->

<xs:element name="ClassName" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="ClassDescription">

<xs:sequence>

<xs:element name="ClassName" type="xs:string" minOccurs="1" maxOccurs="1" />

<xs:choice maxOccurs="unbounded">

<xs:element name="Member" type="Member"/>

<xs:element name="Method" type="Method"/>

</xs:choice>

</xs:sequence>

</xs:complexType>

<xs:complexType name="DataType">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:enumeration value="char"/>

<xs:enumeration value="short"/>

<xs:enumeration value="int"/>

<xs:enumeration value="float"/>

<xs:enumeration value="double"/>

<xs:enumeration value="string"/>

<xs:enumeration value="object"/>

</xs:restriction>

</xs:simpleType>

</xs:complexType>

<xs:complexType name="Port">

<xs:sequence>

<xs:element name="PortName" type="xs:string" minOccurs="1" maxOccurs="1"/>

<xs:element name="PortDescription" type="xs:string" minOccurs="0"/>

<xs:element name="PortUserTextDescription" type="xs:string" minOccurs="0"/>

<xs:element name="DataType" type="DataType" minOccurs="1" maxOccurs="1"/>

<xs:element name="Link" type="xs:string" minOccurs="1" maxOccurs="1" />

<xs:element name="ObjectRequirements" type="ClassDescription" minOccurs="0" maxOccurs="1" />

</xs:sequence>

</xs:complexType>

<xs:complexContent name="In">

<xs:extension base="Port">

<xs:sequence>

<xs:element name="ActivationType">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:enumeration value="Activating"/>

<xs:enumeration value="Passive"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

<xs:element name="ReferenceType">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:enumeration value="Reference"/>

<xs:enumeration value="Value"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

</xs:sequence>

</xs:extension>

</xs:complexContent>

<xs:complexContent name="Parameter">

<xs:extension base="Port">

</xs:extension>

</xs:complexContent>

<xs:complexContent name="State">

<xs:extension base="Port">

</xs:extension>

</xs:complexContent>

<xs:complexContent name="Out">

<xs:extension base="Port">

<xs:sequence>

<xs:element name="ConnectionType">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:enumeration value="Connected"/>

<xs:enumeration value="Disconnected"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

</xs:sequence>

</xs:extension>

</xs:complexContent>

<xs:complexType name="FMData">

<xs:sequence>

<xs:element name="FMName" type="xs:string" minOccurs="1" maxOccurs="1" />

<xs:element name="FMInstanceName" type="xs:string" minOccurs="1" maxOccurs="1"/>

<xs:element name="FMTextDescription" type="xs:string" minOccurs="0"/>

<xs:element name="FMUserTextDescription" type="xs:string" minOccurs="0"/>

<xs:element name="FMVersion" type="xs:string" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

<xs:element name="FMDescriptor">

<xs:complexType>

<xs:sequence>

<xs:element name="FMData" type="FMData" minOccurs="1" maxOccurs="1"/>

<xs:element name="Inputs">

<xs:element name="In" type="In" minOccurs="0"/>

</xs:element>

<xs:element name="Parameters">

<xs:element name="Parameter" type = "Parameter" minOccurs="0"/>

</xs:element>

<xs:element name="States">

<xs:element name="State" type="State" minOccurs="0"/>

</xs:element>

<xs:element name="Outputs">

<xs:element name="Out" type="Out" minOccurs="0"/>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

Родителски елемент в схемата е елемента FMDescriptor, в който са включени базовото му описание (FMData), както и описание на всички налични входове (In), параметри (Parameter), вътрешни състояния (State).

За да се осигури възможност за сериализация, както и за да се осигури връзката с потребителския интерфейс са дефинирани следните класове:

[Serializable]

[XmlRoot("FMDescriptor")]

public class FMDescriptor

{

[XmlElement("FMData")]

public FMData fmData;

[XmlArray("Inputs"), XmlArrayItem(typeof(In), ElementName = "In")]

public List<In> lstInputs;

[XmlArray("Parameters"), XmlArrayItem(typeof(Parameter), ElementName = "Parameter")]

public List<Parameter> lstParameters;

[XmlArray("States"), XmlArrayItem(typeof(State), ElementName = "State")]

public List<State> lstStates;

[XmlArray("Outputs"), XmlArrayItem(typeof(Out), ElementName = "Out")]

public List<Out> lstOutputs;

}

public class FMData

{

public string FMName;

public string FMInstanceName;

public string FMTextDescription;

public string FMUserTextDescription;

public string FMVersion;

}

public class Port

{

public string PortName;

public string PortDescription;

public string PortUserDescription;

public string DataType;

public string Link;

public string Objectrequirements;

}

public class In:Port

{

public bool Activating;

public bool Rference;

}

public class Parameter:Port

{

}

public class State:Port

{

}

public class Out:Port

{

public bool Connected;

}

Алгоритъма за сериализация обхожда всеки един функционален модул, като събира данните за всяка от входно-изходните точки и ги съхранява в XML формат съгласно гореописаната схема. Всеки така формиран XML се съхранява в свободното поле Tag на контролата Shape. Тази процедура дава възможност след това да се използва вградената в диаграмата функция за сериализация, като практически цялата необходима информация за възстановяването на диаграмата ще бъде запазена. При запазването на диаграмата се запазват формите и разположението на всички елементи на диаграмата, както и връзките между тях. С помощта на допълнително съхранената информация при последващо зареждане диаграмата ще бъде възстановена в оригиналния си вид.