ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ



ФАКУЛТЕТ АВТОМАТИКА



Катедра „Системи и Управление”

**ДИПЛОМНА РАБОТА**

**Тема:** Разширение на ПГ PROCONOS чрез система за генериране, експорт и параметризиране на метамодели

**Дипломант: Ръководител:**

**/**Николай Евгениев Демирев/ / доц. д-р Иван Евг. Иванов /

**Ръководител на кат. СУ:**

/ проф. д-р Емил Гарипов /

СОФИЯ 2013

Съдържание:

[1. Описание на задачата и очаквани крайни резултати 3](#_Toc367692614)

[2. Изходни данни 3](#_Toc367692615)

[3. Изходни литературни източници 3](#_Toc367692616)

[4. Съдържание 3](#_Toc367692617)

[4.1. Увод 4](#_Toc367692618)

[4.2. Глава 1. Преглед на методи за програмна генерация. Метаописания на системи. 5](#_Toc367692619)

[4.2.1. Езиците от стандарта IEC 61131-3 5](#_Toc367692620)

[4.2.1.1. Езици за програмиране 6](#_Toc367692621)

[4.2.1.2. Програмни организационни единици 7](#_Toc367692622)

[4.2.1.3. Задачи (Tasks) 8](#_Toc367692623)

[4.2.1.4. Sequential function charts 9](#_Toc367692624)

[4.2.1.5. Ред на изпълнение 11](#_Toc367692625)

[4.2.1.5.1. Sequential function charts 12](#_Toc367692626)

[4.2.1.5.2. Функционални блокове 13](#_Toc367692627)

[4.2.2. Програмен генераотр PROCONOS 17](#_Toc367692628)

[4.2.3. Matlab/Simulink – Real-Time Workshop 23](#_Toc367692629)

[4.3. Глава 2. Описание на методите за мигриране на системи, създадени чрез програмна генерация 26](#_Toc367692630)

[4.4. Глава 3. Разработка на метамодел на система 28](#_Toc367692631)

[4.4.1. XML семантика за съхраняване на моделите на моделираните системи 29](#_Toc367692632)

[4.4.2. Формат за експортиране на модела на описващата система, подходящ за интерпретатора 35](#_Toc367692633)

[4.5. Глава 4. Система за параметризиране и експорт на системата за управление на конкретна платформа 38](#_Toc367692634)

[4.6. Глава 5. Оценка на резултатите 48](#_Toc367692635)

[4.7. Глава 6. Заключение 49](#_Toc367692636)

[5. Графична част и приложения 49](#_Toc367692637)

# Описание на задачата и очаквани крайни резултати

# Изходни данни

# Изходни литературни източници

[1] G. Frey, L. Litz, Formal methods in PLC programming, in: Proceedings of the IEEE Conference on Systems Man and Cybernetics SMC 2000, Nashville, USA, (2000).

[2] J. Yoo, S. Cha, H.S. Son, C.H. Kim, J.-S. Lee, PLC-based safety critical software development for nuclear power plants, in: Proceedings of the SAFECOMP—International Conference on Computer Safety, Reliability, and Security, Potsdam, Germany, (2004).

[3] C.M. Davidson, J. McWhinnie, M. Mannion, Introducing object oriented methods to PLC software design, in: Proceedings of the International Conference and Workshop: Engineering of Computer-Based Systems (ECBS ‘98), Jerusalem, Israel, (1998).

[4] ISA (ANSI/ISA-S88.01.1995), Standard Batch Control. Part 1. Models and Terminology, Instrument Society of America, 1995.

[5] David, R., Alla, H., 1992. Petri Nets and Grafcet: Tools for modelling discrete events systems. Prentice-Hall.

[6] IEC, 1995a. Guidelines for the application and implementation of programming languages for programmable controllers. Technical Report. IEC.

[7] IEC, 1995b IEC 1131-3. Technical Report. IEC. First edition.

[8] Implementation aspects of the PLC standard IEC 1131-3 Martin Öhman*\**, Stefan Johansson, Karl-Erik Årzén

[9] Windows Presentation Foundation, Wikipedia, <http://en.wikipedia.org/wiki/Windows_Presentation_Foundation>

[10] C Sharp, Wikipedia, <http://bg.wikipedia.org/wiki/C_Sharp>

# Съдържание

## Увод

Софтуерът за индустриалните системи за управление без съмнение е много сложен и труден за разработка. Причината за това лежи в природата на системите за управление, които са предназначени да управляват машини, устройства и процеси. Ако софтуерът не е безгрешен, машините, устройствата или процесите могат да излязат извън контрол и да причинят сериозни щети на производството или дори да отнемат човешки живот. Надежността, сигурността и бързите реакции са сред най-важните особености на този тип софтуер [1, 2].

От друга страна модерните системи за управление обхващат голямо количество функции и работят с широк асортимент от различни хардуерни устройства като сензори, задвижващи механизми, контролери и дори сложни компютърни системи. Затова управляващият софтуер става все по-сложен и заради това труден за разработка и поддръжка. Не е очудващо, че качеството на софтуера и продуктивността на процеса на разработка на софтуера е основен проблем в сферата на управляващите системи [3].

Широко разпространен подход за справяне с някои от сложните проблеми споменати по-горе, използван от компаниите за разработка на управляващ софтуер е изпозването на концепцията за „reusability“. С прости думи, изолзването на концепцията за „reusability“ значи да се разчита на система от добре тествани и практически доказани основни блокове софтуер, които могат да бъдат конфигурирани и свързани в софтуерна система. Този подход е доста подходящ за реализиране на основни контролни функции (според стандарта ISA S88 [4]), които са предимно предназначени за установяване и поддържане на определено състояние на управляващия хардуер, както и променливите на управлявания процес, които са същите или сходни с тези на съвсем различен процес.

За разлика от основните управляващи функции, процедурното управление, което се състои от последователност от събития, преходи между състояния, като старт и стоп на дадена система, са много по-проблемни от гледна точка на разработването на софтуера. Процедурната част от управлението е изцяло зависеща от конкретния процес и изисква разработката на специфичен софтуер. Управлението на разработката на този софтуер е доста важна задача. Имплементирането на функциите за управление днес е основно базирано на Programmable Logic Controllers (PLCs), но постепенно в индустрията навлизат и Embedded Systems. Основният проблем е как да се подобри процесът по разработката на софтуер за различния хардуер.

Решение на посочените по-горе проблеми свързани със сигурността, надеждността, бързото и качествено разработване на управляващ софтуер е използването на програмен генератор. За да се реализира управляващата система, използвайки програмен генератор, е необходимо описание на системата под формата на метамодел. Различните функционални блокове от метамодела са предварително разработени, тествани и платформено независими.

Целта на дипломната работа е да се представи улеснен подход за разработка на управляващ софтуер чрез ПГ PROCONOS. За целта на дипломната работа е създадено разширение на ПГ PROCONOS, което представлява система за генериране, експорт и параметризиране на метамодели.

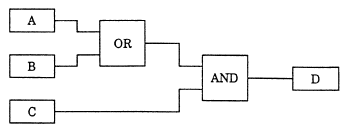
## Глава 1. Преглед на методи за програмна генерация. Метаописания на системи.

### Езиците от стандарта IEC 61131-3

IEC 61131-3 е стандарт, дефиниран от International Electrotechnical Commission (IEC), който дефинира четири порграмни езика, които се използват за програмиране на PLC контролери. Ще бъдат разгледани два от тях – Function Block Diagram (FBD) и Sequential Function Charts (SFC). Програмите за автоматично управление, имплементирани чрез гореспоменатите езици, представляват метаописания на системата, реализираща управлението. След като софтуерът бъде имплементиран чрез FBD или SFC следва превеждането на програмата към ANSI-C код, което става автоматично от средите за програмиране, които предлагат проиводителите на PLC контролерите. След като програмата е преведена към ANSI-C код следва компилирането към изпълняем машинен код за съответния хардуер. Преминаването (превода) от FBD или SFC към ANSI-C се нарича програмна генерация.

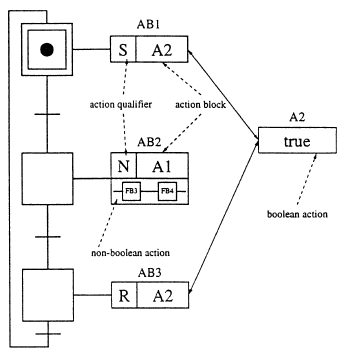
### Езици за програмиране

Function block diagram - в този език всички функции, входове и изходи са представени като графични блокове. Те са свързани един с друг чрез линии, които показват потока на данните. Посоката е винаги от ляво надясно с изключение на обратните връзки. Пример на FBD може да се види на Фигура 4.1.



*Фигура 4.1. Function Block Diagram.*

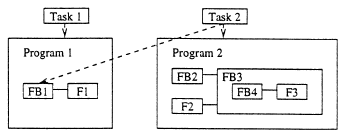
Sequential function charts – този език представлява разширен автомат. Според стандарта SFC е начин за структуриране на програми и функционални блокове. Част от програмата, която не е структурирана, се смята за самостоятелно действие, което се изпълнява непрекъснато. Пример на SFC може да се види на Фигура 4.2.



*Фигура 4.2. Sequential function charts.*

### Програмни организационни единици

Независимо от избора на език, PLC промирането по стандарта IEC 61131-3 използва три програмни организационни единици: функции, функционални блокове и програми. Те са предоставени от производителя или дефинирани от потребителя. Програмна единица не може да бъде рекурсивна т.е. не може да вика себе си. Фигура 4.3 показва как отделните програмни организационни единици си взаимодействат.



*Фигура 4.3. Програмни организационни единици.*

*Функции (Functions).* Функциите имат един или повече входове, но само един изход. Функцията не може да пази информация за състоянието т.е. изпълнението на функция с определени стойности на входовете ѝ винаги дават един и същ резултат на изхода. Функциите могат да бъдат йерархично дефинирани, като се използват вече дефинирани функции. Функцията може да бъде разширявана, което значи, че потребителят може да прецени броя входни параметри. Стандартът не оказва, дали потребителят трябва да зададе броя входни параметри, когато функцията се създава, или броят може да бъде променен по време на живота на функцията. Функциите биват typed (оперират върху определен тип данни) или overloaded (оперират върху различни типове).

*Функционални блокове (Functional blocks).* Функционалните блокове могат да имат няколко входа и няколко изхода. Те могат да пазят информация за състоянието, така че стойностите на изходите зависят от предишните стойности.

Потребителят трябвада именува всяка инстанция на функционален блок с уникално име. Функционалните блокове могат да бъдат йерархично дефинирани, като се използват вече съществуващи функции или функционални блокове.

Според стандарта може да има много инстанции на функционален блок, също така вероятно и функциите могат да имат много инстанции.

*Програми (Programs).* Програмите са изградени от функции и функционални блокове, освен това програмите нямат входове и изходи.

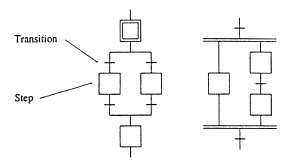
### Задачи (Tasks)

Изпълнението на програми и функционални блокове се управлява от задачи. Стандартът не специфицира дали графикът за изпълнение на задачи може да бъде с приоритетно прекъсване, или без. Задача може да управлява повече от една програмна организационна единица и една програмна организационна единица може да бъде индиректно управлявана от повече от една задача, както функционалният блок FB1 от Фигура 4.3. Ако се опитат да се изпълнят по едно и също време и се използва многозадачност с приоритетно прекъсване на активната задача, имплементацията трябва да подсигури коя програмно организационна единица ще бъде изпълнена.

Задачата има три параметъра: single, interval и priority. Тя може да се изпълни веднъж при зададен параметър single или периодично на определен интервал от време, който се задава чрез параметъра interval. Приоритета на задачата може да се зададе чрез параметъра priority.

### Sequential function charts

Sequential function chart (SFC) е разширен автомат, също познат като Grafecet (David and Alla, 1992). Според стандарта SFC представлява начин за структуриране на програми и функционални блокове. Част от програмата, която не е структурирана се смята за самостоятелно действие, което се изпълнява непрекъснато. SFC се състои от два основни елемента, стъпки (steps) и преходи (transitions), показани на Фигура 4.4. Стъпките могат да са активни или неактивни. Състоянието на SFC се определя от това, кои стъпки са активни. Преходът има булев параметър (условие за преход), което може да бъде описано с всеки от езиците, дефинирани в стандарта IEC 61131-3 или от булева променлива. Преходът ще бъде осъществен когато стъпката преди него е активна и условието за преход е изпълнено. Sequential function charts могат да съдържат паралелни или алтернативни пътища на изпълнение, които са показани на Фигура 4.4.



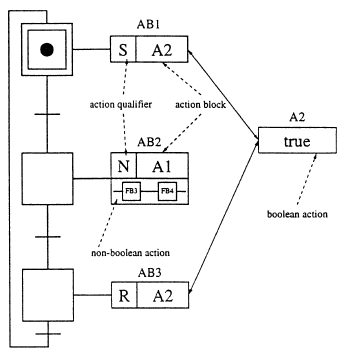
*Фигура 4.4. SFC с алтернативни (ляво) и паралелни (дясно) пътища.*

*Действия (Actions).* Възможно е да се асоциира действие към стъпка. Действието може да е булева променлива, също може да бъде описано с някой от езиците от стандарта включително SFC. Всички действия трябва да имат уникални имена.

*Блокове от действия (Action blocks).* Блоковете от действия се използват, за да се асоциират действия със стъпки. Блокът от действия има булев параметър, който индикира дали асоциираната стъпка е активна. Всеки блок от действия е асоцииран с действие. Стъпките, действията и блоковете от действия са свързани по следния начин:

* Стъпката може да бъде свързана с един, николко или повече блокове от действия.
* Всеки блок от действия е свързанс една стъпка
* Всеки блок от действия е асоцииран към едно действие.
* Всяко действие е асоциирано към един или повече блок от действия.

На Фигура 4.5 се вижда, че булево действие A2 е асоциирано с блок от действия AB1 и AB3. Небулевото действие А1 е конструирано директно в блок от действия AB2, и асоциирано само с един блок от действия.



*Фигура 4.5. Sequential function chart съдържащ блок от действия, описания на действия и действия.*

|  |  |
| --- | --- |
| **N** | **Non-stored** Action е активаен докато е активен action block. |
| **S** | **Stored** Action е активен, когато action block стане активен. |
| **R** | **Reset** Action е деактивиран, когато action block стане активен. |
| **L** | **Limited** Action е активиран, когато action block стане ативен и деактивиран, след определен период от време, когато action block стане неактивен. |
| **D** | **Delayed** Action е активиран определен период от време, след като action block стане активен и е активен докато action block е активен. |
| **P** | **Pulse** Action е активен за един цикъл на системния таймер, когато action block е активиран. |
| **SD** | **Stored and delayed** Action е активен опреден период от време, след като action block стане активен. |
| **DS** | **Delayed and stored** Action е активен определен период от време, след като action block стане активен, ако action block е още активен. |
| **SL** | **Stored and limited** Action е активен, когато action block стане активен и е деактивиран, след определен период от време. |

*Таблица 1. Описания на действия.*

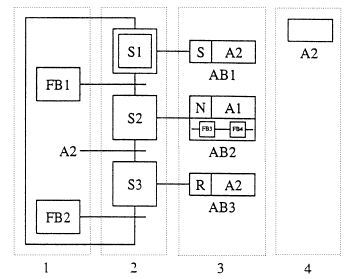
*Описание на действие (Action qualifier).* Блок от действия използва описание на действие за контрол на действието. Действие може да бъде изпълнено в заисимост от описанието на действието за асоциираните блокове от действия и статута на асоциираните стъпки. Различните описания на действия са показани на Таблица 1.

Фигура 4.5 показва блокове от действия, описания на действия и действия.

### Ред на изпълнение

Тук ще бъде разгледан редът на изпълнение между отделните функции, функционални блокове, SFC елементи, блокове от действия и действия. Това не е добре специфицирано в стандарта. Твърди се, че нито един елемент няма да бъде изчислен, преди състоянията на всички входове да бъдат изчислени. Авторите смятат, че това води до следния ред на изпълнение, показан на Фигура 4.6:

1. Функционален блок свързан към преход
2. Стъпки и преходи
3. Блокове от действия и небулеви действия
4. Булеви действия.

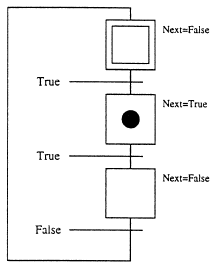


*Фигура 4.6. Редът за изпълнениe на функции, функционални блокове, стъпки, преходи, блокове от действия и действия.*

Друг вариант може да бъде да се изпълнят стъпките и блоковете от действия в една и съща фаза, като се изпълняват само блоковете от действия, които са свързани към активни стъпки. Този метод може все пак да причини проблеми например с описанието на действие “stored”. Тогава блокът от действия ще се изпълни също, след като стъпката е била деактивирана.

### Sequential function charts

Стандартът гласи, че всички стъпки трябва да бъдат обновени синхронно. Това може да бъде постигнато чрез преминаване през всички стъпки два пъти. Първо се отбелязва кои стъпки могат да бъдат активирани и се маркират с флаг “next” (вижте Фигура 4.7). След това всички стъпки се активират. Редът под който се обработват стъпките не е от значение. Този метод подсигурява, че може да стане само един преход наведнъж.

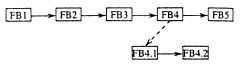


*Фигура 4.7. Изпълнение на SFC.*

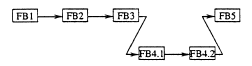
### Функционални блокове

Функции и функционални блокове се изпълняват, така че потокът на данни се движи от ляво надясно. Един начин за запазване на определения ред на изпълнение е функциите и функционалните блокове да се добавят в динамичен списък. Стандартът гласи, че функциите и функционалните блокове могат да бъдат йерархично дефинирани. Не е специфицирано дали такава функция, или функционален блок трябва да бъдат разглеждани като фиксирана единица или вътрешната им структура може да бъде променяна. Ако частите на програмата имат йерархична структура, редът на изпълнение може да се разреши по два начина: локално и глобално сортиране. Не е обявено, кое от тях трябва да се използва.

*Локално сортиране.* В локалното сортиране всеки йерархичен функционален блок има свой списък, който съдържа функциите и функционалните блокове от вътрешната му структура. Фигура 4.8 показва главния списък с функционални блокове на FBD от Фигура 4.10, както и локалния списък от FB4.



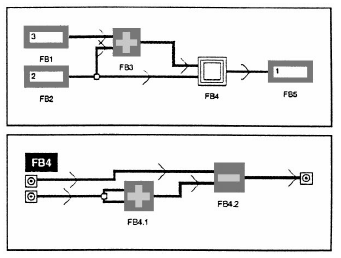
*Фигура 4.8. Списъкът от функционални блокове на FBD от Фигура 4.10.*



*Фигура 4.9. Глобалният списък от функционални блокове от FBD от Фигура 4.10.*

Основното предимство на локалните списъци е, че ако се направят промени в йерархичен функционален блок, само локалните списъци за този функционален блок трябва да бъдат генерирани отново. Това е свързано с разделното компилиране на програмните единици.

*Глобално сортиране.* В глобалното сортиране се използва глобален списък с функционални блокове, който съдържа всички функционални блокове, които са от най-високо ниво и съдържат функционални блокове от всички нива. В този списък не присъстват йерархични функционални блокове. Фигура 4.9 показва глобалния списък на FBD от Фигура 4.10. Забележете, че FB4 не е в списъка.



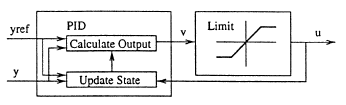
*Фигура 4.10. Йерархичен FBD.*

Основно предимство на глобалните списъци е, че имагинерните цикли, които изглежда, че съществуват локално, се премахват. Ако FBD от Фигура 4.10 трябваше да са сортирани локално, потребителят трябваше да въведе цикъл. Глобалният списък показва, че това не е нужно с глобално сортиране. Премахването на циклите позволява бърза имплементация, като спестява тактове на процесора.

Недостатък на глобалните списъци е, че имплементацията им е по-сложна. Когато йерархичен функционален блок е изключен, всички функционални блокове от вътрешната му структура също трябва да бъдат изключени. Ако те са пръснати в глобален списък, това ще бъде трудно.

*Пример.* ПИД регулаторите са често използвани в индустриалните системи за управление. В IEC 61131-3 обикновено ПИД регулатор се имплементира като функционален блок, който има за вход желания сигнал yref, измеваната величина y и контролния сигнал *v* като изход. Въпреки това, ако насищанията в отработването на заданието не се вземат в предвид, резултатът е windup на интегратора, който се отразява негативно на резултата от управлението. Често използвано решение на проблема е познато като „следене“. Подходът е базиран на връщане на наситения контролен сигнал обратно в ПИД регулатора и промяна на вътрешните състояния, така че контролният сигнал, генериран от контролера, да бъде равен на изходното насищане.

Следователно следенето изисква допълнителен вход на ПИД регулатора – наситеният изход, често означен като *u*. Следенето също значи, че не е възможно да се обнови състоянието на регулатора, докато не стане известен наситеният изход, т.е. да се изчисли. Понякога лимитите на насищане са известни в ПИД регулатора, но невинаги. Например ПИД регулаторът може да е част от каскадна структура или насищанията в отработването се изчисляват в специален ограничаващ блок, както е показано на Фигура 4.11.



*Фигура 4.11. ПИД регулатор.*

Функционалният блок за ПИД регулатор е вътрешно разделен на два блока: CalculateOutput и UpdateState. Единствената стратегия за сортиране, която ще даде правилно поведение от гледна точка на управлението е глобалното сортиране. Изплзвайки тази стратегия, редът на изпълнение ще бъде ПИД: CalculateOutput, Limit, ПИД: UpdateState. Използвайки локално сортиране, старата стойност на *u* може да бъде използвана за изчисляване на UpdateState, т.е. ще бъде въведено ненужно закъснение на един измервателен интервал.

Съгласно дискутираните по-горе подробности се получава следният алгоритъм на изпълнение:

* За всяка функция или функционален блок (сортирани локално или глобално):
  + Прочитане на входовете.
  + Изпълняване на функционалния блок.
  + Записване на изходите.
  + Подаване на изходите към свързаните функционални блокове.
* За всяка стъпка:
  + Проверка дали стъпката може да бъде обновена.
* За всяка стъпка:
  + Ако стъпката може да бъде обновена.
    - Обновяване на стъпката.
  + Записване към блок от действия.
* За всеки блок от действия:
  + Установяване дали асоциираните действия трябва да бъдат активни.
  + Вероятно изпълнение на небулеви действия.
  + Записване на булеви действия.
* aЗа всяко булево действие:
  + Обновяване на действието.

### Програмен генераотр PROCONOS

Програмният генератор PROCONOS е среда за проектиране и реализация на компютърни системи за управление, реализирана на базата на т. нар. разширен автомат на Мур. Практическата реализация на този разширен автомат е чрез графи. Използват се два вида:

* Граф за сигнални трансформации (Signal Flow Graph)
* Граф за преходи на състоянието (State Transition Graph)

Графите за сигнални трансформации (Фигура 4.12) се използват за създаване и промяна на сигналите. Тези графи са съставени от функционални модули, свързани по определен начин. Функционалните модули имат характер на подпрограма. Те използват предварително зададени настройваеми параметри; имат променливи, свързани с техните наблюдаеми състояния и имат изходи от определен тип. Типът на предаваните променливи може да бъде int – целочислена променлива, double – променлива с плаваща запетая и bool – булева променлива. Характерно свойство, което използваме при изработването на модела е, че входовете могат да бъдат променени, така че да бъдат интерпретирани като константи – имат точно определена стойност, която не се променя от друг модул.

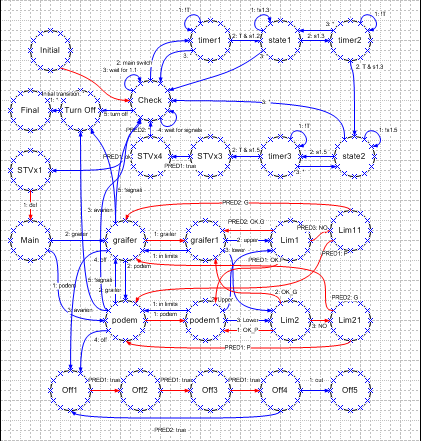


*Фигура 4.12. Структура на сигнална трансформация.*

Част от вградените функционални модули, които предлага програмният генератор са:

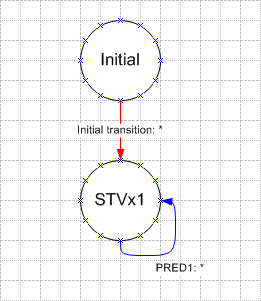
* Timer – Таймер с възможности за отброяване на определен брой тактове, многократно или еднократно стартиране, използване на стойности от вграден Windows таймер.
* Digital In – Модул за прочитане на сигнали чрез серийната комуникация на компютъра
* Digital Out – Модул за изпращане на сигнали по серийната комуникация на компютъра
* Message Reader – Модул за прочитане на сигнали от комуникационния канал на системата за работа в реално време
* Const Repeater – Модул, реализиращ стъпаловидно входно въздействие със зададена амплитуда
* Msg Writer – Модул, изпращащ сигнал към комуникационния канал на системата за работа в реално време
* Linear Constraint – Модул, извършващ ограничение по ниво на входния си сигнал и даващ знак за премината дадена граница
* PID – Модул, реализиращ стандартен ПИД регулатор с вътрешно ограничение на максимален и минимален управляващ сигнал
* SUM – Суматор на два сигнала

Графите за преходи на състоянието (Фигура 4.13) реализират логиката, вложена в модела и са включени като задачи, които се изпълняват при изпълнение на модела от програмата за работа в реално време. Те се състоят от върхове (състояния) и преходи. Всеки връх може да бъде свързан с един граф за сигнална трансформация, който се изпълнява, докато съответният връх е активен. Всеки преход може да зависи от едно или повече условия. За да се осъществи преход, трябва всичките условия за него да са удовлетворени. Важно свойство на преходите е, че могат да бъдат синхронни или асинхронни. Синхронен е преход, който се извършва когато е удовлетворено условието му и когато е постъпил сигнал от синхроизточника на системата (tick). Асинхронният преход се осъществява веднага, след като е изпълнено условието за прехода. Това свойство се използва, за да бъдат реализирани някои от по сложните функции на обекта в модела.



*Фигура 4.13. Граф за преходи на състоянието (STG).*

Една разновидност на графите за сигнални преходи е драйверният граф (Фигура 4.14). Той се използва за прочитане на данни от входни платки от комуникационната подсистема на управляващия компютър. Съставянето му е като на всеки друг граф, но се включва в задачите, които изпълнява моделът с тип драйвер. Независимо какво извършва моделът, на всеки такт на системата за работа в реално време ще се изпълнява драйверният граф. Тази функция ще се използва по-късно за да се презаписват стойности на входните сигнали, различни от вече получените.



*Фигура 4.14. Драйверен STG граф.*

Програмата за работа в реално време се състои от интерпретатор, визуализатор и потребителско управление.

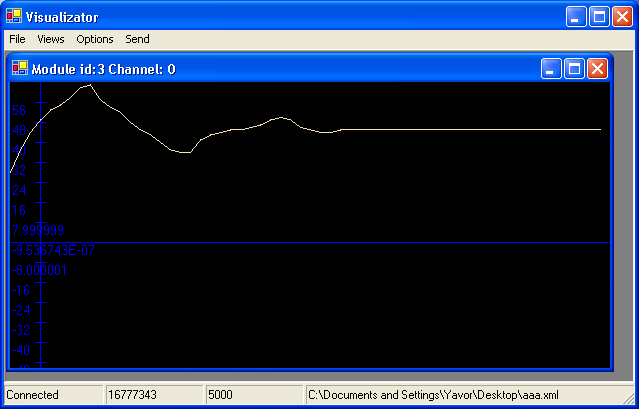
Интерпретаторът използва набор от системни таблици, описващи изцяло модела (Фигура 4.15). Той също така подържа комуникационния канал и управлява изходите и входовете към модела.



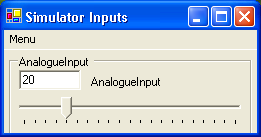
*Фигура 4.15. Интерпретиране на модел.*

Визуализаторът (Фигура 4.16) се използва за графичен изход, чрез който могат да се наблюдават всички сигнали, които са изпратени в комуникационния канал. Освен това чрез него можем да изпращаме сигнали към даден канал, да прочетем моментна стойност и дори да извлечем запис (log file) от даден канал.

Програмата за потребителско управление (Фигура 4.17) служи за създаване на управляващи сигнали и смущения. Тя използва предварително направен конфигурационен файл, в зависимост от който предлага различни по брой и тип входове към обекта. Типовете сигнали, които можем да създаваме са Int – целочислен, Bool – булеви и Double – с плаваща запетая. Методът, по който се генерира сигнала, може да бъде чрез checkbox, slider, switch или joystick.



*Фигура 4.16. Визуализатор.*



*Фигура 4.17. Програмата за потребителско управление.*

### Matlab/Simulink – Real-Time Workshop

Matlab е софтуерен пакет, който предоставя високопроизводителен език за програмиране за технически изчисления. Той се използва за изчисления, визуализации и програмиране в лесна за използване среда, където проблемите и решенията са описани в познат математически стил. Типичните приложения са:

* Математика и изчисления
* Разработка на алгоритми
* Моделиране, симулация и разработка на прототипи
* Анализ на данни, изследвания и визуализация
* Научни и инженерни графики
* Разработка на приложения, включително графичен потребителски интерфейс

За целите на дипломната работа ще разгледаме средата за моделиране и симулиране Simulink. Simulink е софтуерен пакет за моделиране, симулиране и анализиране на динамични системи. Той поддържа линейни и нелинейни системи, моделирани като аналогови, дискретни или хибридни. Също така системите могат да бъдат различнотактови, т.е. различни части от тях използват различна тактова честота.

За моделиране, Simulink предоставя графичен потребителски интерфейс за изграждане на блок диаграми. Моделите могат да бъдат йерархични. Моделираната система може да се разглежда от високо ниво, също така всеки блок може да се разгледа отделно в детайли. Този подход помага, за да се получи по-добра информация, за това, как е органииран моделът и как взаимодействат отделните части. След като се дефинира модел, може да се направи симулация на моделираната система.

Real-Time Workshop, който се използва с MATLAB и Simulink, генерира код директно от моделите на Simulink и автоматично компилира програми, които могат да работят на много различни среди, включително системи за реално време и самостоятелни симулации. Също така могат да се стартират модели на Simulink в реално време на външен процесор. Real-Time Workshop помага да се изпълняват бързи самостоятелни симулации на компютъра, на който се разработват или друг компютър. Тук са изброени някои от приложенията на Real-Time Workshop:

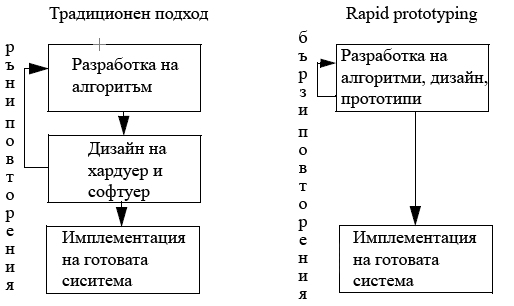
* Управление в реално време – могат да се разработват системи за управление използвайки MATLAB и Simulink и да се генерира код от блок диаграмата, която представлява модел на разработваната система. След това може да се компилира и запише директно на управляващото устройство.
* Обработване на сигнали в реално време – могат да се разработват алгоритми, а обработка на сигнали в MATLAB и Simulink . Кодът, който се генерира от блок диаграмата, може да се компилира и запише директно на управляващото устройство.
* Високоскоростни самостоятелни симулации
* Генериране на портативен C код с цел експортиране към други програми за симулация

Генерираният код е високо оптимизиран и съдържащ много коментари.Той може да бъде генериран от всеки модел на Simulink, включително линейни, нелинейни, аналогови, дискретни или хибридни модели. Real-Time Workshop позволява да се изберат различни езици, в които да бъдат конвертирани желаните модели:

* C код – генерира код, който съдържа системните уравнения и инициализиращите функции за модела на Simulink. Този код може да се използва в нереалновремеви симулационни среди, а реалновремеви приложения.
* Ada код – генерира Ada код от модела на Simulink.
* Real-time program – трансформира генерирания код в реалновремева програма, подходяща за хардуер за реално време. Резултатният код е изграден така, че да взаимодейства с външен източник на тактов сигнал и по този начин да работи на фиксирана и избрана от потребителя тактова честота.
* Високопроизводителна самостоятелна симулация – използва генерирания код заедно, за да генерира изпълним файл за самостоятелни симулации. Накрая на симулацията изпълнимият файл генериа model.mat файл, който съдържа променливите, които са следени в Simulink. Този файл може да бъде използван и за анализи в MATLAB.

Real Time Workshop позволява rapid prototyping (скоростно разработване на прототипи). Това е процес, който позволява да се направи концептуално решение използвайки модели от блок диаграми. И така рано да се предвиди производителността на системата, преди да се избере хардуера, да се напише какъвто и да е софтуер или да се работи по фиксиран дизайн. Като се използва rapid prototyping, може да се прилага conscious integration (практика, при която работата на няколко разработчици се свързва с основната част на разработвания софтуер по няколко пъти на ден). Освен това параметрите на модела могат да се настройват по всяко време,като Simulink се използва за потребителски интерфейс на реалновремевия модел.

Ключът към rapid prototyping е автоматичната генерация на код. Това намалява писането на алгоритми и го превръща в автоматизиран процес. Той включва писането, компилирането, свързването и записването на софтуера на съответната хардуерна платформа. Тази автоматизация позволява да се правят промени по дизайна директно върху блок диаграмата. Фигура 4.18 показва процеса на rapid prototyping.



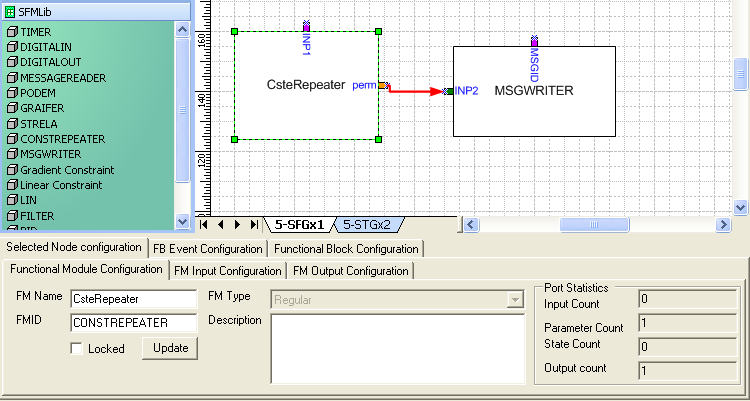
Фигура 4.18. Сравнение на традиционния процес на разработка на прототипи и Rapid prototyping.

Традиционният подход за дизайн на системи за реално време и имплементацията им обикновено включват няколко екипа от инженери, включилтено и екип, който се занимава с разработката на алгоритмите. Когато екипът, разработващ алгоритмите, завърши спецификациите, екипът от софтуерни разработчици имплементира алгоритмите в среда за симулации и разработва спецификация за необходимия хардуер. Екипът, който се занимава с разработката на хардуера, го произвежда. Накрая друг екип внедрява произведения продукт в по-голяма система. Този подход води до голям производствен процес, защото екипът, разработващ алгоритмите, не работи с реалния хардуер. Процесът rapid prototyping комбинира фазите за дизайн на алгоритмите, софтуера и хардуера и по този начин елиминира евентуалните затруднения. Процесът позволява на инженерите да видят резулататите и бързо да подобрят дизайна преди скъпият хардуер да бъде произведен.

След като са постигнати желаните резултати в средата на Simulink, чрез Real-Time Workshop се генерира C код, подходящ за съответния хардуер. Използвайки Simulink може да се настроят параметрите и да се довърши модела, като отново се използва rapid prototyping, за да се постигнат желаните резулатите. На това ниво rapid prototyping процесът е завършен. Може да се започне финалното имплементиране с убеждението, че всички използвани алгоритми, работят добре в системата за реално време.

## Глава 2. Описание на методите за мигриране на системи, създадени чрез програмна генерация

Описанието на системите, които се моделират в програмния генератор PROCONOS представлява метамодел съставен от блокове, наречени функционални модули. Всеки модул има входове и изходи, които се използват, за да се подават входните величини и след трансформациите им да се получават изходните. За да се предават данни от един функционален модул към друг, модулите се свързват посредством линии, по които преминава информацията. Повечето модули представляват атомарни действия (такива, които не могат да се разбият на по-прости). Сложните метамодели на системи могат да съдържат в себе си много на брой атомарни действия, чрез които се описва разработваната системата. Това затруднява създаването, поддържането на модела, откриването и отстраняването на грешки, промяната му. При разработването на модела много време се отделя за настройването на отделните функционални блокове, както и свързването им един с друг. Сложни конфигурации могат да съдържат хиляди функционални модули. Всеки от тях може да има различен брой входове и изходи, с което броят на връзките може да се сведе до десетки хиляди, които трябва да се свързват един по един с помощтта на мишката. Това отнема изключително много време на инженера и може да доведе до много грешки, както и затруднява проследяването на логиката, осъществена чрез връзките в модела. За конфигурирането на отделните модули се налага използването на сложен потребителски интерфейс, който значително забавя работата на инженера. На Фигура 4.19 може да се види потребителският интерфейс на програмния генератор PROCONOS, чрез който се конфигурират функционалните модули.

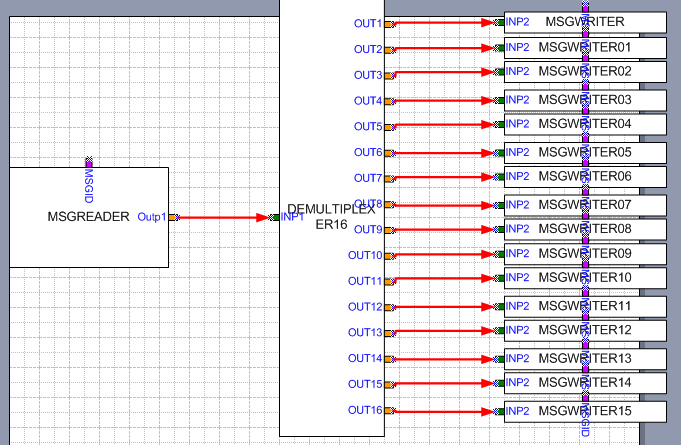


*Фигура 4.19. Потребителски интерфейс на програмен генератор PROCONOS*

Често се налага модели, разработени от един инженер, да се променят и допълват от друг. Наличието на толкова голям брой малки операции, дефинирани от функционалните модули, прави модела труден за разбиране от някого, който не е бил предварително добре запознат с него. Тези проблеми изискват нов подход в изграждането на метамоделите, който би улеснил разработката, валидацията и поддръжката на моделите на системите.

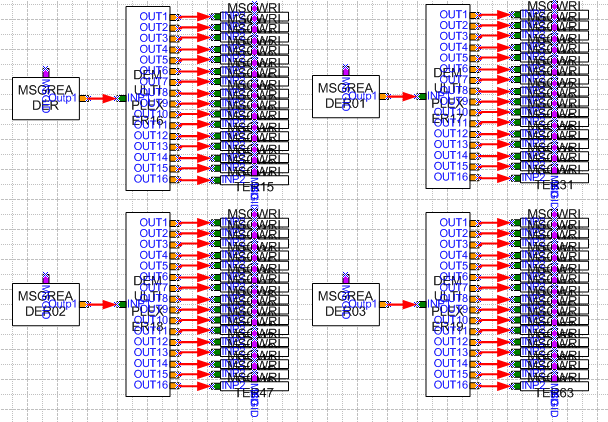
## Глава 3. Разработка на метамодел на система

Решение на проблемите, дискутирани в глава 4.3., би била нова система за конфигуриране на метамоделите, която използва вече съществуващи по-сложни модули, които ще наричаме шаблони, съставени от повече на брой атомарни действия. Един шаблон би могъл да съдържа повече от един функционален модул. Например за реализиране на шестнадесетканален демултиплексор се използват един Message Reader, шестнадесет Message Writer модула и един модул Demultiplexer16. Модулите се свързват, така че Message Reader да е вход за Demultiplexer16, и всеки един от Message Writer модулите се свързва към един от изходите на Demultiplexer16. По този начин се получава готов шаблон за Demultiplexer16, който може да се използва във всякаква конфигурация. Може да се види как изглежда шаблона за Demultiplexer16 на Фигура 4.20, изграден в графичната среда за създаване на метамодели на програмния генератор PROCONOS.



Фигура 4.20. Demultiplexer16 изграден в графичната среда за създаване на метамодели на програмния генератор PROCONOS.

За да може да се използват няколко подобни демултиплексора е необходимо цялата схема да се копира и така се получават много на брой елементарни модули, които създават объркване у потребителя и затрудняват разбирането на модела. На Фигура 4.21 може да се види конфигурация, съставена от четири шестнадесетканални демултиплексора.



Фигура 4.21. Конфигурация, съставена от четри шестнадесетканални демултиплексора.

## XML семантика за съхраняване на моделите на моделираните системи

Сложните системи за управление може да представляват съвкупност от хиляди подобни модули. Използвайки шаблоните, споменати по-горе, схемата би изглеждала много по-проста. Би съдържала четири на брой инстанции на един и същи шаблон, чиито входове и изходи се настройват според нуждите на потребителя. За да бъде възможно създаването на подобни шаблони, както и създаването на метамодели, използвайки тези шаблони, е необходимо всеки от тях да бъде реализиран в средата на програмния генератор PROCONOS и да бъде експортиран в подходящ формат, който да позволи лесно описание на всички входове и изходи и всички съдържащи се функционални модули, както и на тяхното взаимодействие.

За целта е избран езика XML (e**X**tensible **M**arkup **L**anguage). Причината за това е, че XML позволява лесно да се съхранява обектно-орентирани структури от данни поради същността си на метаезик.

XML е стандарт (метаезик), дефиниращ правила за създаване на специализирани маркиращи езици, както и синтаксисът на който тези езици трябва да се подчиняват. Сам по себе си той е безполезен, защото указва само как да бъде структуриран един документ. Това става чрез маркиране с етикети (тагове или tags), но не и какво означават оделните маркери (етикети). За да бъде маркирана дадена информация, тя се огражда в отварящ и затварящ таг. Таговете могат да бъдат вложени и всеки таг може да има неограничен брой атрибути. За да се спазват определени правила за именуване и структуриране на XML се създават така наречените схеми (schemas).

За да бъдат описани метамоделите, генерирани чрез разработваната система, е дефинирана следната семантика на форматиране на информацията използвайки XML:

Всеки файл, съдържащ описание на метамодел, започва с отварящ таг Configuration и винаги завършва със затравящ. Този таг се използва, за да се дефинира откъде започва описанието на метамодела и къде приключва. Също така се използва, за да дефинира схемата, по която е изграден XML документа. Всяка конфигурация може да съдържа един или много графи за сигнални трансформации (Signal Flow Graph или SFG). Те са използвани, за да реализират логиката, вложена в модела, и са включени като задачи, които се изпълняват при изпъленение на модела от програмата за работа в реално време.

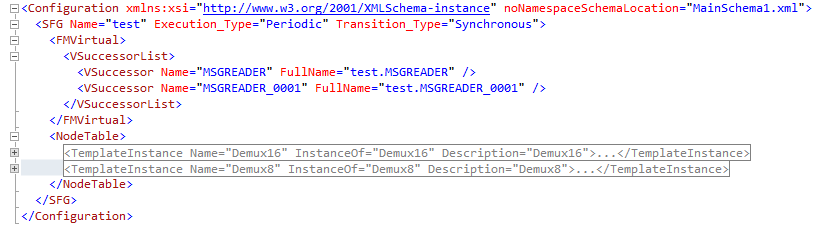
Всеки граф за сигнална трансформация се дефинира чрез тага SFG. Тагът притежава три атрибута – Name, Execution\_Type и Transition\_Type. Атрибутът Name се използва за да се зададе име на съответния SFG. Тъй като всеки SFG се изпълнява като задача (task), атрибутите Execution\_Type и Transition\_Type описват тяхнотот изпълнение. Атрибутът Execution\_Type може да приеме стойностите „Periodic“ – за периодично изпълнение на задачата или „OneShot“ – за еднократно. Периодичните обикновено се използват за извършване на работата на самата програма, а тези които се изпълняват еднократно – за инициализиране на комуникация или заделяне на памет или за връзка с бази данни и т.н. Атрибутът Transition\_Type може да приеме стойностите „Synchronous“ или „Asynchronous“. Асинхронните се използват за отработване на възникнало събитие, а синхронните - за нормалното изпълнение на програмата. SFG тагът може да съдържа вложени в себе си два други тага – FMVirtual и NodeTable.

Тагът FMVirtual се явява начална точка в програмата, съдържаща списък с функционални блокове, която посочва от кой от функционалните модули ще започне изпълнението на съответната програма и след като завърши изпълнението на съответния шаблон, чиито функционален модул започва изпълнението, се преминава на следващия в списъка от виртуални функционални модули. Списъкът с виртуални функционални блокове се задава чрез тага VSuccessorList. Тагът се влага директно в тага FMVirtual. Всеки виртуален функционален модул се дефинира с таг VSuccessor.

Всеки VSuccessor се влага директно във VSuccessorList и притежава два атрибута – Name и FullName. Атрибутът Name се използва, за да се даде уникален идентификатор на всеки функционален модул в контекста на съответния SFG. Атрибутът FullName се използва, за да се даде уникален идентификатор на всеки функционален модул в контекста на цялата конфигурация. Пълното име е съставено от името на съответния SFG, в който е включен функционалният модул, следван от „.“, следвана от името на функционалния модул.

Тагът NodeTable, споменат по-горе, е другият възможен директен наследник на тага SFG. Той се използва, за да опише списъка от използваните темплейти и няма атрибути. Всеки шаблон е негов директен наследник.

Описаните дотук тагове могат да се видят на Фигура 4.22.



*Фигура 4.22. XML описващ шаблоните*

Тагът, използван за дефиниране на инстанция на шаблон, е TemplateInstance. Тагът има три атрибута – Name, InstanceOf, Description. Атрибутът Name се използва, за да се даде уникален идентификатор на инстанцията на шаблона в контекста на целия метамодел на описваната система. Атрибутът InstanceOf дава информация за името на шаблона, чиято инстанция се описва. Атрибутът Description се използва за метаинформация, която не е нужна в логиката на системата, а само съдържа информация, която може да е полезна за потребителя. Всяка инстанция на шаблон съдържа списък на функционалните модули, от които е съставен.

Всеки функционален модул, съдържащ се в инстанцията на шаблона, се описва от тага FM, който има десет атрибута – Name, FullName, UserName, Description, Function, ActCount, InputCount, OutputCount, IsFmDataTypeEditable, MsgType. Първият атрибут – Name се използва, за да се даде на съответния функционален модул уникален идентификатор в контекста на инстанцията на шаблона. Следващият атрибут - FullName се използва, за да се даде уникален идентификатор на функционален модул в контекста на съответния SFG. FullName е съставен от името на SFG, последвано от точка, последвано от името на функционалния модул. UserName се използва, за да може потребителят да зададе име на функционалния модул през потребителския интерфейс. Атрибутът Description се използва за метаинформация, която не е нужна в логиката на системата, а само съдържа информация, която може да е полезна за потребителя. Атрибутът Function определя коя функция отговаря на съответния функционален модул. Следващият атрибут – ActCount се използва, за да покаже колко входа на модула са означени като активиращи. Целта е когато всеки един от тях получи стойността си да се променя брояча на активираните входове и когато всички са активирани да се изпълни съответният модул. Това гарантира, че всички необходими данни ще бъдат получени преди изпълнението му. Също така се гарантира, че за едно изпълнение на графа, функционалният модул ще бъде изпълнен само веднъж. Следващият атрибут – InputCount се използва, за да се опише колко входни портове съдържа съответният функционален модул. Другият атрибут – OutputCount се използва, за да се опише колко изходни портове съдържа съответният функционален модул. Следващият атрибут – IsFmDataTypeEditable се използва, за да се опише дали типа данни на съответния функционален модул, подлежи на промяна чрез потребителския интерфейс на конфигуратора.

Всеки функционален модул съдържа два списъка, единият от които описва наследниците на дадения модул. Другият описва портовете на модула. Под наследници се разбира модулите, които са свързани след текущия модул (неговите изходи представляват техни входове). Така след като даденият функционален модул е изпълнен, изпълнението продължава към неговите наследници. Портове са съответните входове и изходи. За да се опишат списъците се използват таговете FMSuccessorList и PortDescriptorTbl. Двата тага нямат атрибути.

В списъка с наследниците на всеки модул се описват един по един функционалните модули, които са преките наследници. Всеки наследник се описва с тага FMSuccessor. Той притежава два атрибута – Name и FullName. Първият атрибут – Name се използва, за да се даде на съответния функционален модул уникален идентификатор в контекста на инстанцията на шаблона. Следващият атрибут - FullName се използва, за да се даде уникален идентификатор на функционален модул в контекста на съответния SFG. FullName е съставен от името на SFG, последвано от точка, последвано от името на функционалния модул.

В списъка с портовете на функционалния модул се описват един по един входните и изходните портове на функционалния модул. Всеки порт се описва от тага Port. Той притежава дванадесет атрибута – Name, FullName, ActivationType, InputType, VarType, Storage, Rows, Cols, SourcePort, InitialValue, Description, Visible. Първият атрибут – Name се използва, за да се даде на съответния функционален модул уникален идентификатор в контекста на инстанцията на шаблона. Следващият татрибут - FullName се използва, за да се даде уникален идентификатор на функционален модул в контекста на съответния SFG. FullName е съставен от името на SFG, последвано от точка, последвано от името на функционалния модул. Атрибутът ActivationType може да приеме две стойности – Activating и Passive. Activating се използва, когато за да се продължи с изпълнението на даден модул, някои от портовете му се изисква да получат данни. Тези портове биват маркирани с ActivationType Activating. Следващият атрибут – InputType се използва, за да определи дали даденият порт е изход на функционалния модул или вход. Може да приема две стойности – Input и Output. Input означава, че даденият порт е вход за функционалния модул, а Output означава съответно, че е изход. Следващият атрибут – VarType се използва, за да се дефинира типа на данните, предавани през съответния порт. Възможни типове данни: Int, Bool, Double, Timestamp, Short, Byte, Float, UInt, UShort. Следващият атрибут – Storage, може да приеме две стойности - Link и Static. Когато стойността е Static, за този порт се заделя памет и се използва като променлива. Ако стойността е Link, за него не се заделя памет, а се реферира друг порт, за който е заделена памет (маркиран със Static). За да се подаде променлива от външна среда може да се използва MSGReader, като портът му е настроен на Link и така да се подаде външна стойост. Следващият атрибут – Rows, се използва, за да се зададе едната размерност на данните, които се подават през порта – броя редове. Това е необходимо, за да може да се предават матрици (многомерни данни). Атрибут Cols се използва, за да се зададе едната размерност на данните, които се предават през порта – броя колони. Това е необходимо, за да може да се предават матрици (многомерни данни). Пример за употребата на атрибутите Rows и Cols е предаването на символни низове. Тяхната размерност се задава като един ред и толкова колони, колкото символа се съдържат в низа. Следващият атрибут – SourcePort, се използва, за да се определи от кой порт в конфигурацията ще се вземат данните. Този атрибут се използва само, ако стойността на атрибута Storage е Link. Използва се пълното име на реферирания порт. Атрибутът InitialValue се използва за определяне на началната стойност на порта. В зависимост от типа на порта, данните поддържани от InitialValue варират. Следващият атрибут – Description, се използва за метаинформация, която не е нужна в логиката на системата, а само съдържа информация, която може да е полезна за потребителя. Другият атрибут – Visible, се използва за да се определи дали този атрибут ще е видим в системата за конфигуриране на метамодела.

На Фигура 4.23 може да се види структурата на XML използвана за описване на метамоделите.



*Фигура 4.23. XML описващ шаблоните*

## Формат за експортиране на модела на описващата система, подходящ за интерпретатора

След като моделът е описан в разработваната система е необходимо той да може да бъде експортиран във формат, който се разбира от интерпретатора на програмния генератор PROCONOS. За целта се използват няколко текстови файла, които описват модела.

* fbEventList.txt

На първия ред във файла е записан броя на MSGReader модулите. На следващия ред е записан броя на MSGWriter модулите. След това на всеки следващ ред е записана стойността на атрибута MSGID за всеки MSGReader. Следва на всеки ред записване стойността на атрибута MSGID за всеки MSGWriter.

* fbEventTable.txt

Файлът съдържа описание на събитията, активиращи преходите в системата. Всеки ред започва с уникален идентификатор за съответното събитие, следващата стойност е името на STV, който да бъде активиран, последната стойност е типа на събитието – стандартно или драйверно.

* FMDescriptor.txt

Всеки ред от файла съдържа информация за всеки един от функционалните модули от конфигурацията. Информацията съдържа пълното име на функционалния модул съставено от името на STG графа + „.“ + името на сътветния връх на графа + „.“ + името на SFG графа + „.“ + името на шаблона + „.“ + името на функционалния модул. Следва името на функцията на функционалния модул. След това е ред на стойността на атрибута ActCount на функционалния модул. След нея – стойността на атрибута InputCount, след това на атрибута OutputCount и на атрибута MSGType. За разделител между всеки две стойности се използва символът за табулация („\t“).

* FMPortDescriptor.txt

Всеки ред от файла съдържа информация за всеки един от портовете на функционалните модули, съдържащи се в дадена конфигурация. Информацията се състои от пълното име на порта съставено от името на STG графа + „.“ + името на сътветния връх на графа + „.“ + името на SFG графа + „.“ + името на шаблона + „.“ + името на функционалния модул + „.“ + името на съответния порт. Следва стойността на атрибута ActivationType, след това стойността на атрибута InputType, после на атрибута VarType, след нея на атрибута Storage, след това на атрибута Rows, на атрибута Cols, следвано от пълното име на порта, който е свързан с текущия (атрибута SourcePort). Последната стойност е стойността на атрибута InitialValue. Изброените стойности са разделени помежду си със символа за табулация (“\t”).

* sfgSuccessorTable.txt

Всеки ред от файла съдържа информация за наследниците на функционалните модули. На първите редове на файла се съдържат имената на всички наследници на виртуалния функционален модул на SFG графа. На следващите редове са изброени всички наследници на функционални модули, съдържащи се във всички шаблони в конфигурацията. Редът им във файла е същият, както в конфигурацията. Името на функционалните модули представлява името на STG графа + „.“ + името на сътветния връх на графа + „.“ + името на SFG графа + „.“ + името на шаблона + „.“ + името на функционалния модул.

* SFGTable.txt

Файлът съдържа информация за броя на наследниците на всеки функционален модул в конфигурацията. Всеки ред съдържа името на функционалния модул и броя на наследницете му. Първи функционален модул винаги е виртуалният модул на SFG графа. Името на всеки функционален модул е съставено от името на STG графа + „.“ + името на сътветния връх на графа + „.“ + името на SFG графа + „.“ + името на шаблона + „.“ + името на функционалния модул. Стойностите на всеки ред на файла са разделени посредством символа табулация („\t“).

* StateSpecification.txt

Този файл съдържа описание на състоянието на върховете на STG графа. Първата стойност от всеки ред на файла описва името на върха от STG графа. Следващата стойност описва броя на задачите в един връх на графа. Следващите два параметъра не се използват. Следва името на BDD върха. Последните два параметъра не се използват.

* STGraph.txt

Този файл описва графа на преходите. Всеки ред от него съдържа описание на един преход от графа. Всеки ред съдържа няколко стойности. Първата е името на BBD върха, който се използва за идентификация и навигация в случай на преход към следващ BDD връх. Следващият параметър е условието, при което се извършва прехода. Условието може да е „TRUE“ – безусловен преход, „\*“ – по подразбиране е „TRUE“, ако останалите условия не бъдат изпълнени. Използва се, когато се проверяват няколко условия. Също така условието може да бъде зададено като име на порт на функционален модул, който връща булева стойност, която да бъде проверена. Следващият параметър показва дали изчислението приключва с преход към определено състояние на графа на преходите - „T“ или продължава с преминаване към друг BDD връх - „N“. Поредният параметър е типът на прехода. Той може да бъде синхронен - „Sync“, асинхронен – „Async” или може да не съществува преход – „None“. Последният параметър показва накъде ще продължи изпълнението. Стойностите са „NOVAR“ – в случай, когато няма преход или името на BDD върха в случай, че преходът е означен с „N“ или името на състоянието в случай, че преходът е означен с „T“.

* TaskTable.txt

Файлът описва задачите, които са дефинирани в конфигурацията. Всеки ред от файла описва задача. Първият параметър във файла описва типа на задачата. Типовете могат да са обикновена – „0“ или външно събитие – „1“. Следващият параметър посочва името на задачата, което всъщност е името на един SFG.

## Глава 4. Система за параметризиране и експорт на системата за управление на конкретна платформа

В тази глава ще бъде представена системата за параметризиране и експорт на системата за управление. Системата е реализирана, използвайки технологията WPF за създаване на потребителски интерфейс за десктоп приложения и програмния език C# .Net с помощта на който е релизирана логиката на системата и функционалността за експорт.

Windows Presentation Foundation или WPF е система за визуализиране на потребителски интерфейс на Windows базирани приложения и е разработена от Microsoft. WPF е познат преди като Avalon. Първоначално е публикуван като част от .NET Framework 3.0. Вместо да разчита на системата GDI, WPF използва DirectX. Целта на WPF е да предостави консистентен програмен модел за разработване на приложения и разделянето на потребителския интерфейс от бизнес логиката. За описване на потребителския интерфейс се използва XAML, който е XML базиран език, който дефинира и свързва различни UI елементи [9]. Нуждата от разделение на потребителския интерфейс от бизнес логиката идва от това, че когато са имплементирани заедно, те трудно могат да бъдат сменени, защото са тясно свързани. Когато се използва разделение на потребителски интерфейс и бизнес логика, много лесно и бързо потребителският интерфейс може да бъде сменен. Също така това дава възожността потребителският интерфейс и бизнес логиката да се разработват от различни екипи и след като са готови да се свържат.

C# .NET е обектно ориентиран език за програмиране, разработен от Microsoft като част от платформата .NET. Още при създаването на езика C# стремежът е бил да се създаде един прост, модерен, обектно ориентиран език с общо предназначение. Основа за C# са C++, Java и донякъде езици като Delphi, VB .NET и C. Той е проектиран да балансира мощност (C++) с възможност за бързо разработване (Visual Basic и Java). Те представляват съвкупност от дефиниции на класове, които съдържат в себе си методи, а в методите е разположена програмната логика – инструкциите, които компютърът изпълнява. [10]. Като обектно ориентиран език C# поддържа концепциите за енкапсулация, наследяване и полиморфизъм. Всички променливи и методи, включително Main метода, който е входната точка за приложението, са енкапсулирани в дефиниция на клас. Клас може да наследява директно един бащин клас, но може да имплементира неограничен брой интерфейси. Методите му могат да заместват виртуалните методи от бащиния клас, което изисква употребата на запазената дума на езика override като начин да се заобиколи случайно предефиниране на даден метод. В C# структурите са като прости класове.Те са типове, които могат да се съхраняват в стека и могат да имплементират интерфейси, но не поддържат наследяване.

В добавка към тези основни обектно ориентирани принципи, C# прави лесна разработката на компоненти чрез няколко иновативни езикови конструкции, включително изброените:

* Енкапсулирани сигнатури на методи, които се наричат делегати (delegates) и позволяват типово безопасни известявания за събития
* Публични ствойства (Properties), които се използват за достъп до частни (private) променливи.
* Атрибути, които предоставят декларативна метаинформация относно типове по време на изпълнението на кода.
* XML документиране на кода, съдържащо се в самия код
* Language-Integrated Query (LINQ), което предоставя вградени възможности за създаване на заявки към различни източници на данни.

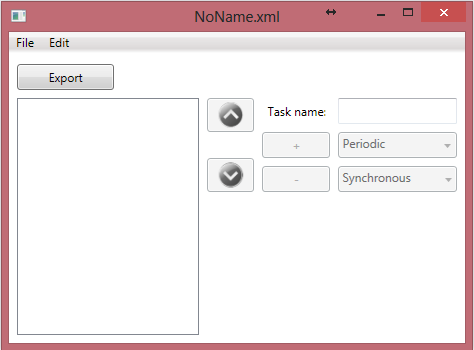
Програмите написани на C# се изпълняват от .NET Framework, който е интегриран компонент от операционната система Windows и включва в себе си виртуална система за изпълнение, която се казва Common language runtime (CLR) и унифициран набор от библиотеки с класове. CLR е комерсиална имплементация на Microsoft на Сommon language infrastructure (CLI), което е международен стандарт, който е база за създаване, изпълнение и разработване на среди за програмиране, в които езиците и библиотеките работят заедно безпроблемно.

Код написан на C# се компилира в intermediate language (IL), който отговаря на CLI спецификацията. IL кода и ресурсите, като изображения и символни низове, се съхраняват на диска заедно с изпълним файл, който се казва assembly, който типично има разширение .exe или .dll. Едно assembly съдържа така наречения манифест (manifest), който дава информация за типа на това assembly, версията, културата и изискванията за сигурност.

Когато програма на C# се изпълнява, assembly файлът се зарежда в CLR, който може да предприеме различни действия в зависимост от информацията, която се съдържа в манифеста. Когато изискванията на съответнотот assembly са изпълнени, CLR изпълнява just in time (JIT) компилация, за да конвертира IL кода до машинни инструкции. CLR също предоставя услуги, свързани с автоматично освобождаване на заделената динамична памет (garbage collection), управление на изпълнението и мениджмънт на ресурсите. Код, който се изпълнява от CLR понякога се нарича менажиран код (managed code). За разлика от неменажирания код (unmanaged code), който се компилира в машинен език, който е специфичен за системата, на която се изпълява.

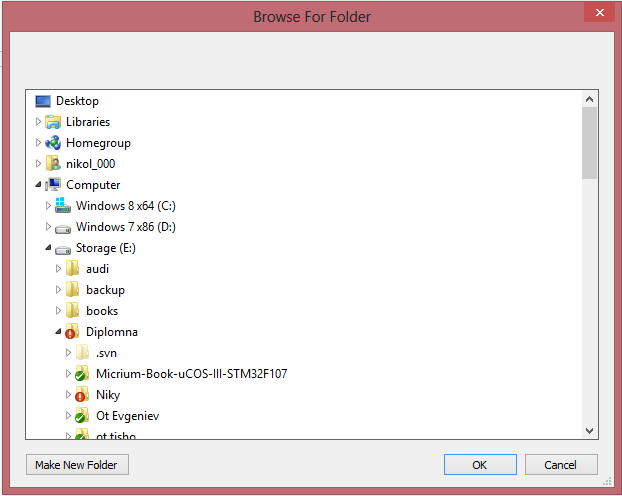
.NET платформата помага на програмиста, като предсотавя широк набор от имплементирани структури от данни и алгоритми, както и мощни разширения на езика като LINQ. LINQ или Language-Integrated Query е набор от функции, които разширяват синтаксиса на езика C# с мощни възможности за съставяне на заявки. LINQ предоставя стандартни, лесни за учене практики за създаване на заявки и обновяване на данни. Технологията може да бъде разширена, за да поддържа всички варианти за съхранение на данни. Visual Studio (средата за разработка) съдържа вградени библиотеки, които позволяват употребата на LINQ с колекциите на .NET, SQL бази данни, ADO .NET Datasets, както и XML документи. Тази възможност на езика е широко използвана в тази дипломна работа, за обработка на XML документите, с които се описват метамоделите на системите.

Преди да бъде разгледано реализирането на логиката и потребителския интерфейс на системата, ще бъде разгледан потребителският интерфейс и какви фунцкии може да изпълнява той. На Фигура 4.24 се вижда началният екран на системата.



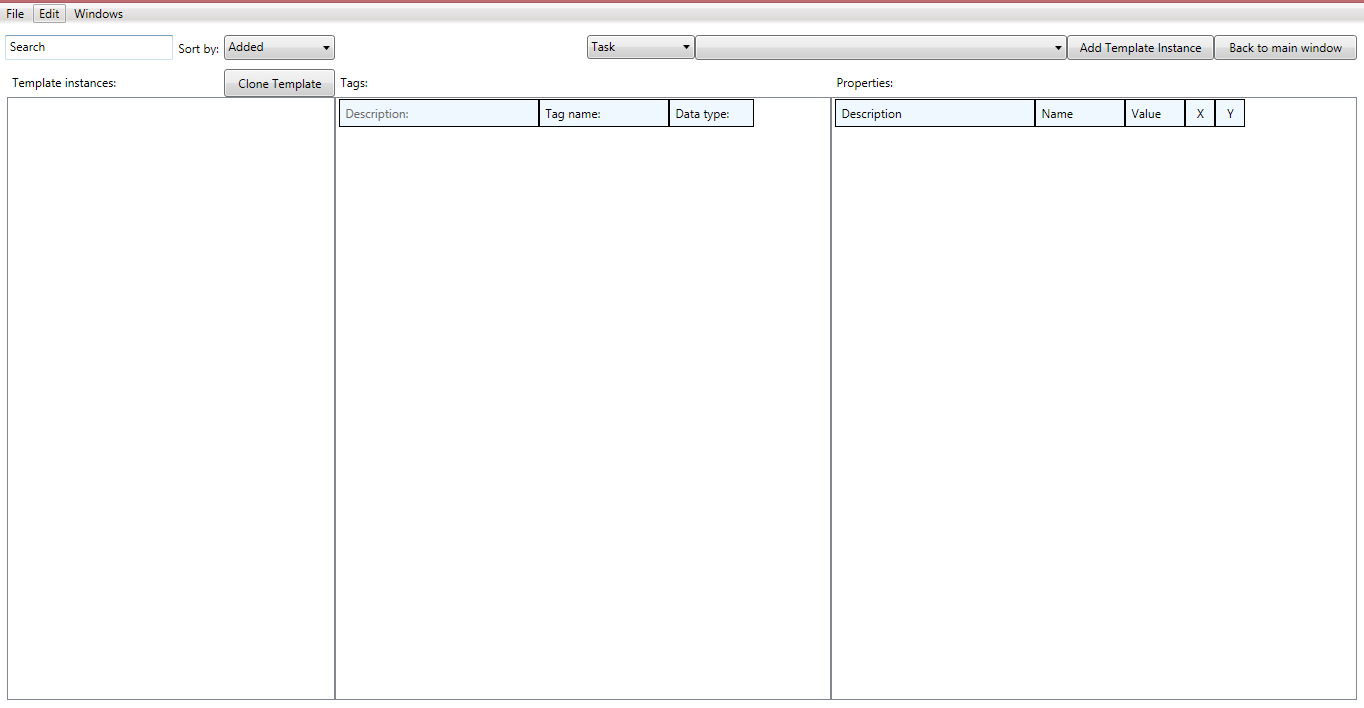
*Фигура 4.24. Начален екран на системата за генериране, експорт и параметризиране на метамодели*

**Менюто „File“ предлага възможност за създаване на нов модел, за отваряне на съхранен такъв или за записването на създадения модел. Менюто „Edit“ позволява конфигуриране на директорията съдържаща предварително създадените шаблони, които се използват за създаването на метамоделите на системите. Също така чрез менюто „Edit“ може да се настройва дали да се показват или не скритите портове от конфигурацията с цел откриване на грешки. Освен това може да е настрои да се скрива или показва администрирането на типовете данни на отделните портове, както и други настройки. Бутонът „Export“ се използва за експортиране на създадената конфигурация във формат подходящ за интерпретатора на програмния генератор PROCONOS. След натискане на бутона се отваря диалогов прозорец, чрез който се избира директорията, в която да се запази експортирания модел на системата. Диалоговият прозорец може да се види на Фигура 4.25.**

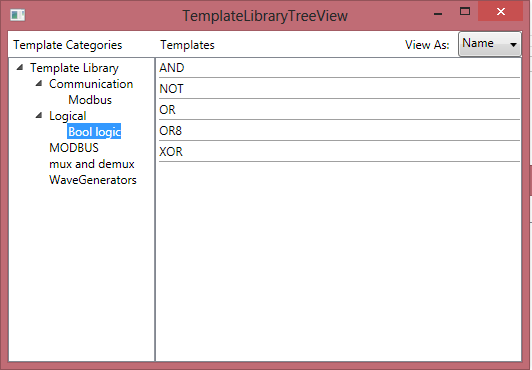


***Фигура 4.25. Диалогов прозорец,позволяващ да се избере директорията, в която да се експортира генерирания метамодел.***

**Текстовото поле „Task name“ се използва, за да се даде име на задачата (Task), която се създава. След като се зададе име на задачата се избира дали тя ще се изпълява периодично или само веднъж посредством полето, намиращо се непосредствено под полето „Task name“. Двете позволени стойности са „Periodic“ и „One Shot“. Полето, намиращо се точно под последното, дава възможност да се избере дали задачата ще бъде синхронна или асинхронна. Позволените стойности са „Synchronous“ и „Asynchronous“. След като са въведени настройките на задачата се натиска бутонът „+“ и задачата се създава и името и се показва в списъка със задачи, намиращ се директно под бутона „Task name“. По същия начин могат да се създават нови задачи. Ако е нужно да се редактират описаните по-горе параметри на задача, е достатъчно тя да бъде избрана от списъка със задачи и да се променят желаните параметри. За да се промени редът на задачите в списъка се използват бутоните стрелки, след като е избрана задачата, която е нужно да се премести. Всяка задача от списъка може да бъде изтрита посредством бутона „-“. За да бъде описана логиката на всяка задача е необходимо в списъка със задачите да се кликне два пъти върху желаната задача. Така се отваря диалоговият прозорец, чрез който се добавят инстанции на шаблони към мета-модела и също така се конфигурират. Той може да бъде видян на Фигура 4.26. Освен този диалогов прозорец се отваря и диалоговият прозорец, съдържащ библиотеката от шаблони, които могат да се използват. Фигура 4.27 показва диалоговия прозорец ,съдържащ библиотеката от шаблони.**

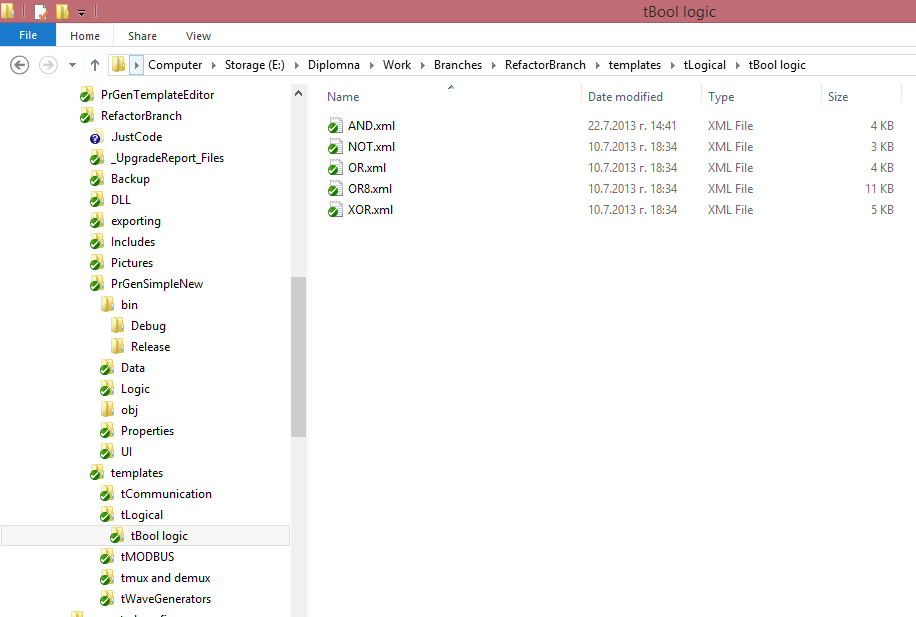


***Фигура 4.26. Прозорец за инстанциране и настройка на шаблоните***



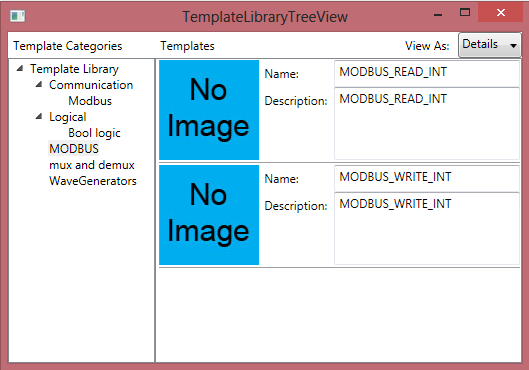
*Фигура 4.27. Диалоговв прозорец, съдържащ библиотеката с шаблони*

**Както се вижда на Фигура 4.27, шаблоните са разделени по категории и всяка категория може да бъде избрана. Категориите се съдържат в дървовидната структура от лявата страна. В дясната страна се показват съдържащите се в категорията шаблони. За да се дефинира библиотеката от шаблони се използва директория, в която всяка категория е отделна директория и в нея са поставени предварително генерираните шаблони. Пътят до директорията с шаблоните може да бъде настроен. Тази настройка ще бъде разгледана по-късно. Фигура 4.28 показва структурата на директорията с шаблони, която е показана в диалоговия прозорец от Фигура 4.27.**



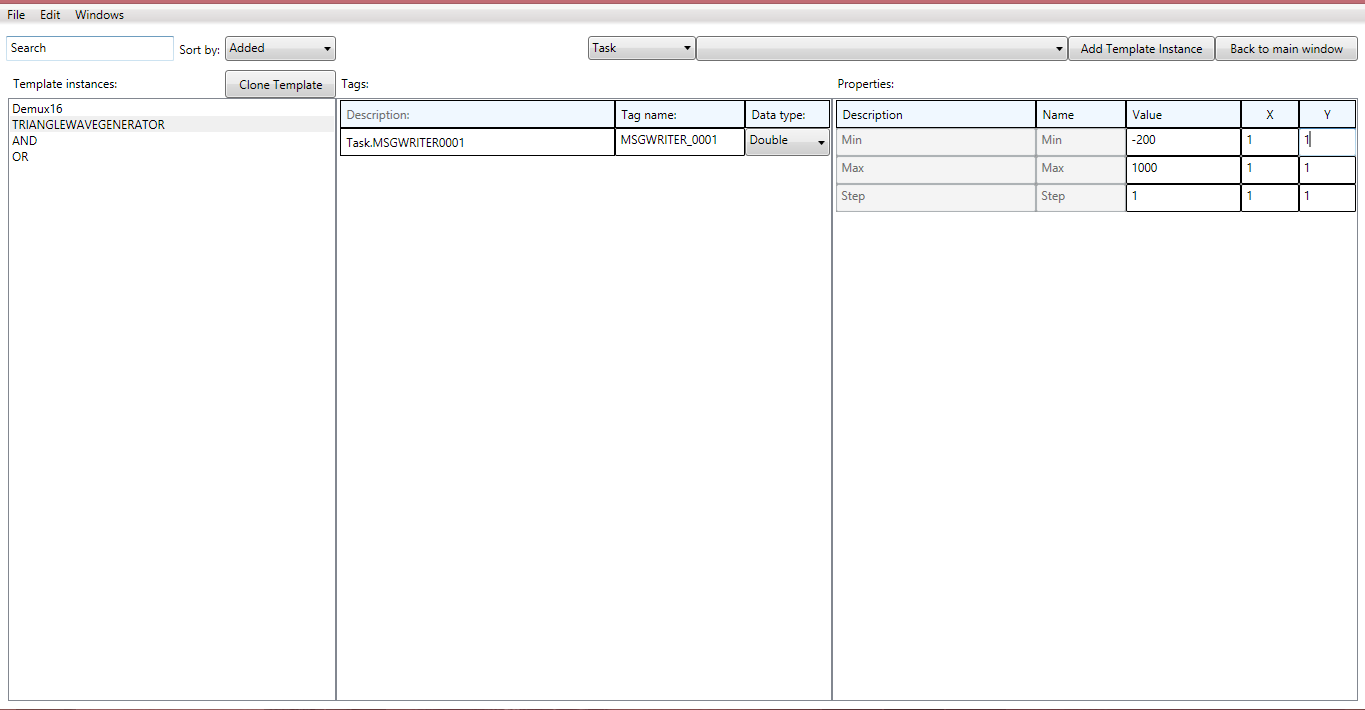
*Фигура 4.28. Структура на директорията, съдържаща библиотеката от шаблони.*

За да бъде показана директория, съдържаща шаблони, в библиотеката е необходимо името и да започва с буквата t“. Следващата част от името на директорията представлява името на категорията в библиотеката с шаблони. Библиотеката с шаблони позволява два различни изгледа на съдържаните шаблони във всяка категория. Единият изглед е показаният на Фигура 4.27, другият може да се види на Фигура 4.29. Този изглед се настройва посредством бутона „View as“, който позволява изгледите „Name“ и „Details“. Детайлният изглед, който е показан на Фигура 4.29, показва картинка, която може да визуализира шаблона, както и името му и кратко описание, ако са зададени в описанието на шаблона.



*Фигура 4.29. Детайлен изглед на галерията с шаблони.*

За да бъде инстанциран избраният шаблон е необходимо да се кликне два пъти върху него. В този момент той се добавя в списъка с инстанции на шаблони, който се намира в лявата страна на прозореца от Фигура 4.26. На Фигура 4.30 се вижда диалоговият прозорец, който съдържа инстанциите на шаблоните и позволява конфигурирането им. От лявата му страна има списък с инстанциите на шаблоните. Всяка инстанция може да бъде избрана и по този начин в съседните две части от диалога се показват настройваемите параметри на инстанцията. В средната част се показват настройваемите входове и изходи, а в дясната се показват настройваемите им параметри. Когато се кликне два пъти върху създадена инстанция на шаблон се влиза в режим, който позволява преименуването и. За по-лесно намиране на желаната инстанция на шаблон е имплементирана опция за търсене, която при въвеждане на текст в полето за търсене филтрира показваните инстанции на шаблони. Показват се само тези, които съдържат в името си въведения текст в полето за търсене. Освен това е реализирано и сортиране на инстанциите по три критерии. Това се определя чрез бутона „Sort by“. Единият критерий е времето на добавяне – „Added“, другият е сортиране по имената на инстанциите на шаблони – „Name“ и последният е по тип на шаблона – „Type“.

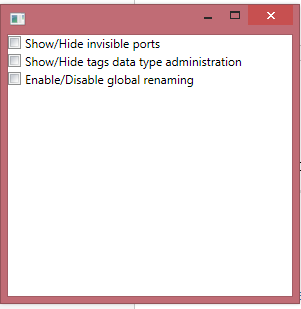


*Фигура 4.30. Прозорецът, който съдържа инстанциите на шаблоните и позволява тяхното редактиране*

Бутонът „Clone Template“ се използва, когато е необходимо бързо създаване на инстанция от вече инстанциран шаблон. Това става като се избере съответната инстанция на шаблон и се натисне споменатият бутон.

Когато има повече от една задача, тя може да бъде сменена бързо чрез бутона за избор на задачи, намиращ се в централната част на прозореца. Когато се натисне се показва списъкът със създадени задачи и когато се избере някоя от тях, в диалога се зареждат инстанциите на шаблони, съдържащи се в нея. Съседният бутон позволява бърз достъп до библиотеката с шаблони. Когато се избере шаблон и се натисне бутонът „Add Template Instance“ се добавя новя инстанция на избрания шаблон. Последният бутон – „Back to main window“ се използва за връщане към началния прозорец, който е описан по-горе, в който се създават и редактират задачите. Диалогът съдържа и няколко менюта – „File“, „Edit“ и „Windows“.

**Менюто „File“ предлага възможност за създаване на нов модел, за отваряне на съхранен такъв или за записването на създадения модел. От менюто „Edit“ може да се настойва пътят до директорията с библиотеката от шаблони. Това става чрез опцията на менюто „Change template dir“. Освен това се дава възможност да се правят настройки и на други особености на приложението. Това става от бутона „Options“, който отваря нов диалогов прозорец, който се вижда на Фигура 4.31.**



*Фигура 4.31. Диалогов прозорец, позволяващ допълнителни настройки на системата.*

**Предлаганите настройки са „Show/Hide invisible ports“. Тази настройка позволява да се показват абсолютно всички портове, които се съдържат в даден шаблон. Обикновено част от тях са скрити, поради причината, че не е необходимо всеки да се настройва. Другата настройка е „Show/Hide tags data type administration“. Когато тази настройка е включена на мястото на типа на съответния порт се показва бутон, чрез който се избира дали за съответния порт е позволена редакция на типа или не.**

## Глава 5. Оценка на резултатите

При съставяне на конфигурация от 200 функционални модула на инженера би му отнело 4 часа да я състави, а с помощта на описваната система би отнело 20 мин. Валидацията също отнема по-малко време при разработване, заради необходимостта от по-малко проверки.

Изградената система за генериране, експорт и параметризиране на метамодели е работоспособна и спестява усилия, предотвратява фактора човешка грешка при еднотипни конфигурации за събиране на данни, повишава производителността на инженера, използващ системата. Чез реализираната система са решени основните проблеми описани в дипломната работа.

## Глава 6. Заключение

Бъдещото развитие на разработваната система би било интеграцията и с програмния генератор PROCONOS, както и разработване на анализатор за генерираните конфигурации. Анализаторът ще показва дали съответната конфигурация е валидна. Примерен критерий за валидация би бил валидация на комуникационните адреси. Също така валидация на повтарящите се параметри.

# Графична част и приложения