UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

tesi di laurea

Progetto e sviluppo di un editor ladder per il tool UniSim

Anno Accademico 2004-05

relatore
Ch.mo prof. Alfredo Pironti

correlatore
Ing. Gianmaria De Tommasi

candidato
Giuliano Latini
matr. 534/000739

Indice

Introduzione

Attualmente i Controllori a logica programmabile (PLC) rappresentano la tecnologia più diffusa nell'automazione industriale, tuttavia ciò che penalizza maggiormente l'utilizzo di questi componenti è la scarsa interoperabilità tra i prodotti commerciali.

Un tentativo di superare tale difficoltà è stato intrapreso dal Comitato Elettrotecnico Internazionale (IEC) con l'introduzione di uno standard denominato IEC 61131–3, che allo stato attuale, rappresenta lo standard più diffuso nell'ambito dei controllori a logica programmabile. L'introduzione di tale standard giustifica la necessità di un modello software in grado di sviluppare progetti d'automazione.

UniSim è un tool che consente di completare il ciclo di sviluppo di un progetto dall'editing al rapid prototyping (RP), con validazione e testing mediante simulazione di controllo, simulazione hardware-in-the-loop e controllo di processi reali non critici.

Nella sua prima versione UniSim prevedeva la possibilità di scrivere programmi in uno solo dei linguaggi definiti dallo standard, in particolare l'SFC; in questo lavoro di tesi ci si è occupati di estendere le funzionalità del tool aggiungendo un editor grafico che permettesse anche la scrittura di programmi in LADDER, un altro dei linguaggi previsti dallo standard, nonché un modulo che permettesse la simulazione dei progetti ottenuti con il nuovo linguaggio.

L'inserimento di questo componente all'interno di UniSim beneficia dell'architettura modulare di cui è caratterizzato il progetto di partenza, preservandone le caratteristiche e seguendo un approccio object oriented: in pratica sono stati creati altri moduli ed inseriti nelle librerie già esistenti.

L'editor dà la possibilità di disegnare un "circuito ladder" utilizzando tutti i

componenti previsti dallo standard (contact, coil ecc.), selezionabili e trascinabili dall'utente, nonché l'interconnessione tra essi e la congiunzione dei *rung* così ottenuti con le linee di alimentazione.

Una volta aggiunto un componente è possibile attribuirgli un nome e collegargli una variabile booleana. È possibile, inoltre, spostare o eliminare i vari componenti dal diagramma.

Sono stati, altresì, realizzare altri moduli che permettono l'esportazione dei progetti tramite file in formato XML, conformemente agli standard previsti, nonché a validare i file XML importati.

Infine è stato sviluppato un modulo che interfacciandosi con il motore di simulazione già presente in UniSim, permette la simulazione anche dei progetti sviluppati in ladder.

Nel primo capitolo della tesi verrà descritto il tool UniSim; nel secondo capitolo verrà fornita un'introduzione allo standard IEC61131-3 ed un'analisi dell'XML Format for IEC 61131-3; nel terzo capitolo verrà illustrata l'architettura software e come questo nuovo componente è stato integrato nel progetto preesistente, infine il quarto capitolo richiamerà i tool utilizzati e riporterà esempi d'impiego.

Capitolo 1

Il tool Unisim

In questo capitolo verrà fornita una breve descrizione del tool UniSim. Si rimanda al terzo capitolo per una descrizione più approfondita della sua architettura.
Il software UniSim può essere utilizzato in tutti quegli ambienti in cui sono presenti architetture di controllo basate su plc. Il dominio applicativo dei sistemi automatici di controllo, oltre ad interessare applicazioni strettamente industriali, sovrasta ormai ogni settore della nostra società: dagli elettrodomestici ai giocattoli, dalle telecomunicazioni all'automazione casalinga e molti altri.

Il software UniSim

UniSim è un tool di sviluppo per software d'automazione estremamente flessibile che si presenta come un'ambiente integrato (VISUAL-EDITOR) con il quale è possibile sviluppare il proprio progetto, conformemente allo Standard IEC 61131-3, senza dover ricorrere a nessun altro tool di sviluppo. Il software permette di disegnare il proprio progetto in modo semplice utilizzando il solo mouse. Il tool permette la progettazione e la simulazione di progetti d'automazione in due linguaggi di programmazione (SFC e LADDER). Il software mette a disposizione una libreria di simboli, ognuno dei quali rappresenta un particolare componente (fase, azione, contatto, ecc.). L'utente, dopo aver scelto il tipo di linguaggio da

utilizzare, deve solamente piazzare i componenti e tracciare le linee di connessione attraverso le funzioni integrate nell'editor. Inoltre, il tool, grazie ad un motore di simulazione integrato, permette di validare i progetti sviluppati.

Una versione modificata dello stesso motore di simulazione è in grado di interfacciarsi con dispositivi di ingresso ed uscita. Questo modulo può essere utilizzato sia per la prototipizzazione rapida e la validazione hardware-in-the-loop di sistemi d'automazione, che per il controllo di processi non critici utilizzando semplici PC equipaggiati con schede di I/O a basso costo. Infine UNISIM utilizza il nuovo schema XML Formats for IEC 61131-3, per l'interscambio di tutte le informazioni relative ad un progetto di automazione (configurazioni, dati e programmi). Grazie a questa caratteristica il software sviluppato e testato con UNISIM potrà essere semplicemente importato e messo in esecuzione su piattaforme di sviluppo di tipo commerciale.

Attualmente UNISIM rappresenta l'unico applicativo esistente, che adotta questo formato per la memorizzazione su file dei progetti d'automazione. Una volta che le varie case produttrici di controllori a logica programmabile avranno recepito questo nuovo standard per l'interscambio dei dati, i progetti d'automazione sviluppati con UNISIM potranno essere semplicemente importati dai vari software di sviluppo proprietari, per poter poi essere mandati in esecuzione sulle varie piattaforme commerciali. Analogamente un progetto d'automazione sviluppato con un software di sviluppo proprietario potrà essere validato utilizzando il simulatore e/o il modulo di controllo di UNISIM.

Il motore di simulazione

Il motore di simulazione permette di visualizzare in modo dinamico la correttezza di un progetto ed apportare modifiche o correzioni sul programma in tempo reale, ossia anche durante la simulazione stessa, senza doverla interrompere. Questa scelta progettuale ha reso necessaria l'eliminazione della fase di compilazione del progetto. Le operazioni proprie di questa fase, come la verifica della correttezza sintattica e semantica o la traduzione dei nomi simbolici in indirizzi fisici di memoria, vengono eseguite contestualmente alle operazioni di editing.

Per garantire la consistenza della simulazione le funzioni di editing e di

simulazione vengono eseguite all'interno di thread concorrenti con sincronizzazione degli accessi agli oggetti comuni.
L'algoritmo di schedulazione è di tipo non-preemptive. La schedulazione dei task dipende dalle loro priorità e dagli intervalli di ciclo. Anche questi due valori sono modificabili in fase di simulazione. I *programs* e i *function blocks* che non sono assegnati ad alcun task sono considerati a priorità più bassa.

Il modulo di controllo

Il modulo di controllo consente di interfacciarsi con dispositivi di I/O permettendo la prototipizzazione rapida e la validazione hardware-in-the-loop del sistema di controllo. L'algoritmo di scansione è lo stesso utilizzato dal motore di simulazione. A differenza di quest'ultimo, però, è privo delle funzioni di sincronizzazione degli accessi (non essendo consentita la modifica on-line del software di controllo) e della modalità di esecuzione a passo singolo.
Queste differenze consentono di ridurre i tempi di scansione e dunque quelli di risposta del sistema di controllo. Con la versione attuale di UniSim viene fornito anche un driver che permette di utilizzare tutte le schede National Instruments con ingressi e uscite digitali di produzione recente.
Così come avviene durante la simulazione, è possibile monitorare in tempo reale l'evoluzione dei progetti ed i valori delle variabili. Si è preferito mantenere questa funzionalità, anche a discapito delle prestazioni, per poter facilitare l'individuazione di errori durante le fase di validazione hardware-in-the-loop dell'algoritmo di controllo.
È importante tenere conto che, essendo il modulo di controllo eseguito su un sistema operativo non real-time come Microsoft Windows, i tempi di risposta vengono influenzati delle caratteristiche del sistema e dal carico di processi da eseguire; inoltre la precisione dei timer è influenzata da quella dell'orologio di sistema. Nel caso di controllo di processi non critici dal punto di vista delle specifiche temporali, il modulo di controllo di UniSim può essere comunque utilizzato anche per realizzare sistemi d'automazione utilizzando semplici PC equipaggiati con schede di acquisizione a basso costo (*softPLC*).

Import/export dei progetti

Tutti i progetti sviluppati con UniSim vengono salvati come documenti XML seguendo lo schema *XML Formats for IEC 61131-3* specificato dal comitato tecnico *TC6-XML* dell'organizzazione *PLCOpen*. L'adozione di questo standard , che verrà descritto nel prossimo capitolo, consente il riuso del software in caso di migrazione tra sistemi di case produttrici diverse. Inoltre il software è munito di un *Validating XML Parsers* che controlla la conformità dei file importati, rendendo di fatto possibile l'eliminazione di ogni eventuale perdita di dati nello scambio di informazioni tra diversi sistemi che rispettano lo standard.

Capitolo 2

Lo standard IEC 61131-3 e l'XML Format for IEC 61131-

In questo capitolo verrà introdotto lo Standard IEC 61131-3 e si analizzerà nel dettaglio lo schema definito dall'XML Format for IEC 61131-3.

2.1 Lo standard IEC 61131-3

La progettazione e lo sviluppo del software di controllo per PLC [1] (Programmable Logic Controller) è un compito che presenta alcune problematiche dovute principalmente alla sua dipendenza dall'hardware del controllore da programmare. Il mercato dei controllori programmabili per l'industria offre una grande varietà di offerte, da parte di una moltitudine di produttori differenti. Ognuno di questi produttori mette a disposizione dei propri clienti un ambiente di sviluppo proprietario, talvolta diversificato persino per serie diverse dello stesso costruttore. Sebbene questi tools siano simili tra loro, le difformità tra le architetture hardware dei controllori e le implementazioni specifiche di funzionalità analoghe fanno sì che il set di istruzioni a disposizione del programmatore possa variare notevolmente, costringendo il progettista ad un attenta lettura dei manuali prima di poter sviluppare un'applicazione su un controllore non conosciuto. Inoltre, la necessità di sviluppare linguaggi diversi dal semplice linguaggio a contatti (linguaggi grafici o testuali di alto livello) per

permettere al programmatore di risolvere compiti più complessi di semplici operazioni booleane, ha portato ad una ulteriore diversificazione tra i produttori e tra gli ambienti di sviluppo.

Per favorire un punto di incontro tra i progettisti del controllo e i produttori di controllori industriali, negli anni recenti è stato introdotto dall'organismo internazionale IEC (Internation Electrotechnical Commission) uno standard che si propone di definire gli aspetti descrittivi e di programmazione dei dispositivi di controllo per l'Automazione Industriale. Tale documento, nominato IEC 61131 [2,5], ed in particolare la sua parte Terza "Programming Languages" è oggetto dell'analisi di questo capitolo.

Il modello software

Il modello software proposto dallo standard IEC 61131-3 è di tipo stratificato e gerarchico, e si basa su di un insieme ben preciso di elementi, ognuno dei quali si trova ad un determinato livello della gerarchia e "racchiude" gli elementi del livello sottostante. Tale gerarchia, può essere schematicamente descritta dalla figura 2.1.

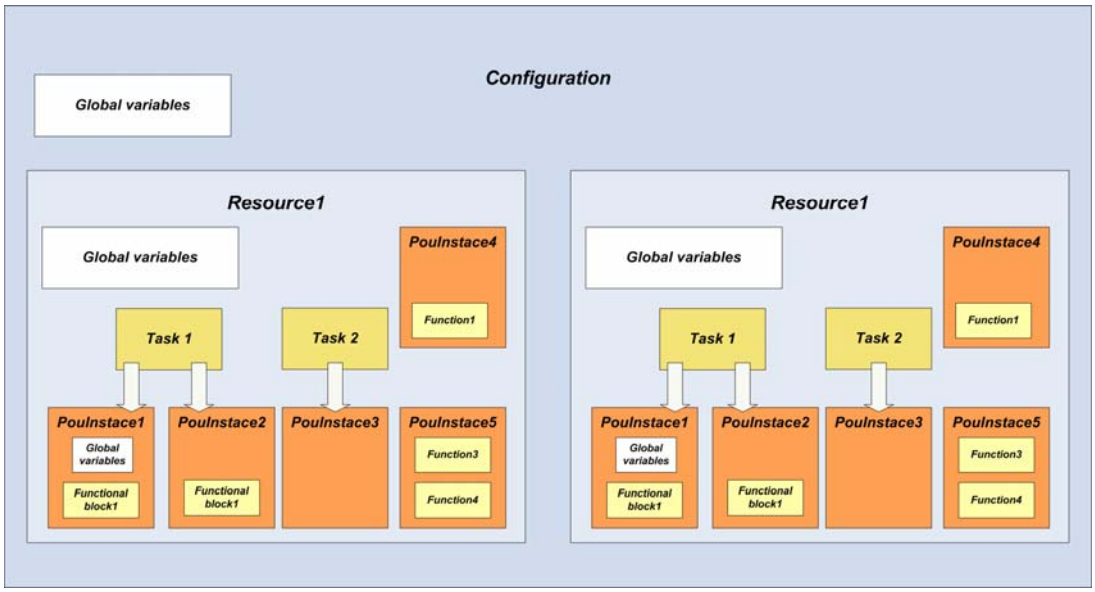


Figura 2.1: Schematizzazione del modello software IEC 61131–3

I componenti del modello software IEC 61131-3 sono, come detto, distribuiti secondo una struttura gerarchica che può essere descritta come un sistema ad oggetti, nel quale ogni elemento è dotato di caratteristiche ed attributi propri, è connesso ad altri oggetti secondo opportune relazioni di aggregazione, appartenenza o utilizzo. Nel seguito sono fornite le definizioni di tali oggetti, mettendo in evidenza, in modo testuale, le relazioni espresse graficamente nella figura 2.1.

Configuration

E' l'elemento più esterno della struttura, perciò contiene gli altri elementi, e corrisponde all'intero sistema controllore programmabile, ovvero, generalmente ad un PLC. In un sistema di controllo è possibile avere più *configuration*[2], le quali possono comunicare conformemente alle modalità previste e definite dello standard IEC 61131 ("Communications").

Resource

Sono contenute all'interno di una *configuration*: esse rappresentano il supporto di esecuzione dei programmi, o in altri termini, l'interfaccia di una "macchina virtuale" in grado di eseguire un programma, il quale per poter funzionare deve necessariamente essere caricato in una *resource* [2]. Quest'ultima, usualmente, è individuabile internamente al PLC, ma potrebbe essere anche simulata con un PC, oppure, ancora, individuabile all'interno di un PC. Lo standard fornisce una descrizione concisa di *Resource*: "Un elemento che corrisponde ad una unità funzionale di elaborazione dei segnali, connessa a dispositivi di interfaccia uomo-macchina e con funzioni di interazione con sensori ed attuatori". Quindi, una delle principali funzioni di questa componente del modello software è di fornire un'interfaccia fra il programma e gli I/O fisici del PLC. Le *Resource* sono, in generale, componenti del modello software tra loro autonome, ciò nonostante, è possibile l'accesso diretto di queste componenti alle Global Variables definite a livello di *Configuration*, le quali, di fatto, costituiscono un meccanismo di

comunicazione fra le *Resource*. E' interessante osservare che lo Standard, prevedendo a livello più esterno una *Configuration* che può contenere più *resource*, le quali a propria volta possono supportare più Program, conferisce al PLC la possibilità di caricare ed eseguire un certo numero di programmi totalmente indipendenti, se esistono "risorse" sufficienti.

Program

Lo standard IEC 61131-3 definisce quest'unità nel seguente modo: "L'insieme logico di tutti gli elementi di programmazione ed i costrutti necessari per la funzionalità di elaborazione di segnali richiesta ad un sistema controllore programmabile." I *Program* [2] si trovano necessariamente all'interno di una *Resource*. Essi sono i contenitori dei costrutti realmente eseguibili, programmati nei linguaggi previsti dallo standard. L'esecuzione di un *Program* avviene sotto il controllo di uno o più *Task* ; nelle realizzazioni più complesse, si possono avere diversi *Tasks* che controllano l'esecuzione dei diversi *Program*.

Task

E' il componente del modello software preposto al controllo dell'esecuzione dei *Programs*. Per ciascun Task può essere specificata un'attivazione periodica o al verificarsi di un determinato evento. Quindi, una volta attivati, i *Task* [2] mandano in esecuzione i moduli dell'applicazione ad essi associati, con le tempistiche o le attivazioni "ad interrupt" imposte.

Local and Global Variables

Gli identificativi delle variabili sono dei nomi simbolici cui il programmatore attribuisce un determinato significato, dipendente dal suo tipo di dato e dalla zona di memoria in cui si richiede che sia allocata. Possono essere definite come locali, quindi accessibili unicamente dalle unità nelle quali sono dichiarate (*Programs* e moduli di programma, *Resources*, *Configurations*). Tuttavia, le variabili possono essere anche definite globali: ciò implica la loro accessibilità da parte dell'elemento in cui sono dichiarate ed, inoltre, di tutti quelli in esso contenuti,

secondo la struttura gerarchica definita nel modello software IEC precedentemente introdotto.

Function Blocks

Essi sono moduli di programma riutilizzabili, grazie ai quali è possibile ottenere programmi strutturati. Due sono le principali componenti di un Function Block [2]:
• DATI: l'insieme delle variabili utilizzate solamente dal codice contenuto nel Function Block, costituito dai parametri in ingresso ed uscita, e dalle variabili interne (merker, variabili temporali ecc.);
• ALGORITMO: un insieme di istruzioni che costituiscono il corpo del modulo di programma, mandate in esecuzione ogni volta che il relativo Function Block è richiamato.
L'algoritmo processa il corrente valore degli ingressi e delle variabili interne, producendo così un nuovo set di valori per i parametri d'uscita e le variabili interne. Queste ultime possono essere locali o globali: nel primo caso esse sono dichiarate internamente al Function Block, nel secondo sono dichiarate a livello di Program, Resource o Configuration. Quindi, i Function Block possono essere visti come "dispositivi software" riutilizzabili, dotati di uno stato interno, tra loro interconnessi tramite i "morsetti" d'ingresso e di uscita.

Functions

Le *Functions* [2] sono anch'esse elementi software riutilizzabili, che ricevono in ingresso un set di valori, e producono in uscita un solo valore. Questa è una prima caratteristica che le differenzia dai *Function Blocks*. Inoltre, mentre questi ultimi possono contenere le dichiarazioni di variabili statiche, cioè il cui valore viene mantenuto tra due esecuzioni successive, le variabili interne di una *Function* possono essere solo temporanee. In sostanza, mentre i *Function Blocks* sono dotati di un proprio stato interno, le *Function* mancano di tale proprietà, per cui, in corrispondenza della stessa configurazione d'ingresso esse forniscono sempre lo stesso risultato.

Program Organisation Units

Lo standard IEC definisce come Program Organisation Units (POU) [2] i seguenti tre elementi del modello software: Programs, Function blocks e Functions.
Questi tre oggetti rappresentano la parte certamente più interessante della norma, perché definiscono finalmente una modalità di organizzazione dell'applicativo efficace ed orientata al riuso. Infatti, la proprietà comune di questi elementi software è che ognuno di essi può essere replicato in diverse parti di una applicazione, qualora le loro funzionalità debbano essere ripetute. Chiaramente la probabilità che ciò accada è maggiore quanto più questa funzionalità è di basso livello gerarchico (cioè quanto più è "piccolo" il modulo). Dato che lo standard punta molto, come detto, a fornire al programmatore gli strumenti per ottenere programmi modulari e ben organizzati gerarchicamente, le POU di livello inferiore (FB e Functions) rappresentano proprio gli elementi sui quali basare la fase di progettazione dell'applicativo. I vantaggi derivanti dall'uso di queste POU si possono così riassumere:

- incremento di leggibilità del codice;
- maggiore riutilizzabilità del codice;
- fattorizzazione dei compiti in sottounità più semplici;
- si evita la propagazione di errori sia in fase di stesura che di manutenzione;
- suddivisione dell'applicativo in moduli il cui sviluppo può essere eseguito in parallelo da diverse persone.

I linguaggi di programmazione

Come detto precedentemente, un aspetto dei controllori industriali tradizionalmente ostile per il progettista è la difformità dei linguaggi di programmazione [1,2]. In tal senso, la Norma intende regolamentare l'aspetto puramente sintattico dei linguaggi di programmazione di controllori industriali, definendo cinque linguaggi standard dalle caratteristiche differenti, che possono coprire tutte le necessità del progettista nello sviluppo di un programma di controllo. Grazie alla standardizzazione della sintassi dei linguaggi, il compito di programmazione risulta semplificato anche in caso di utilizzo di hardware fornito da differenti produttori, con una conseguente notevole riduzione dei tempi di sviluppo del software di controllo. I linguaggi definiti nella norma IEC 61131-3 si dividono in due categorie: testuali (Structured Text (ST) e Instruction List (IS))e

grafici (Sequential Functional Chart (SFC), Function Block Diagram (FBD) e Ladder (LD)). Detti linguaggi si rivelano più o meno adatti a seconda del tipo di algoritmo da implementare. L'SFC ad esempio si rivela particolarmente utile nel controllo di sistemi ad eventi discreti. L'FBD consente facilmente di definire blocchi funzionali o funzioni da riutilizzare all'interno dei programmi. Il Ladder si presta all'implementazione di operazioni che coinvolgono i singoli bit. Il linguaggio Structured Text richiama la programmazione procedurale dei linguaggi di alto livello come il Pascal. Infine il linguaggio Instruction List si avvicina molto alla programmazione in stile assembly. I linguaggi di tipo grafico possono essere utilizzati anche contemporaneamente nello stesso programma. Precisamente un programma in Ladder può contenere blocchi definiti con l'FBD, mentre un programma scritto in SFC può includere sia blocchi dell'FBD che oggetti del linguaggio Ladder. Un modello di programmazione classico utilizza l'SFC per implementare gli algoritmi di controllo di livello più alto. Le azioni eseguite vengono quindi implementate mediante uno degli altri linguaggi o mediante lo stesso SFC. Nei prossimi paragrafi si farà particolare riferimento alla sintassi del linguaggio Ladder.

2.2 XML formats for IEC 61131-3

Lo schema *XML Formats for IEC 61131-3* [5,13]fornisce un'interfaccia aperta per l'interscambio di informazioni (progetti, programmi e librerie) tra differenti piattaforme, secondo lo standard IEC 61131-3. Questo schema è il risultato del lavoro del comitato tecnico TC6–XML dell'organizzazione vendor e product-independent PLCOpen [13] e la sua prima versione ufficiale è datata 28 aprile 2005.
Il formato utilizza il linguaggio non proprietario XML [7], definendo la struttura dei documenti attraverso un XML Schema [3,8]. Questa tecnologia permette la produzione di documenti la cui validità (conformità allo schema) può essere verificata attraverso l'utilizzo dei Validating XML Parsers [3,8], diffusi strumenti software anche di tipo freeware ed open source. *XML Format for IEC 61131-3* consente lo scambio di tutte le informazioni definite dallo standard e delle relative estensioni a carattere grafico per la scrittura dei programmi. Ciò rende possibile l'eliminazione di ogni eventuale perdita di dati nello scambio di informazioni tra sistemi che rispettano lo standard.
Queste caratteristiche forniscono al nuovo XML Format for IEC 61131-3 ottime possibilità di diventare uno standard nell'ambito dei moderni sistemi di automazione.
Nei seguenti schemi, tratti rispettivamente da sito web di PLCopen [13] e dal documento ufficiale di specifiche, si evidenziano il contesto in cui si colloca il formato ed alcuni dei possibili casi d'uso.

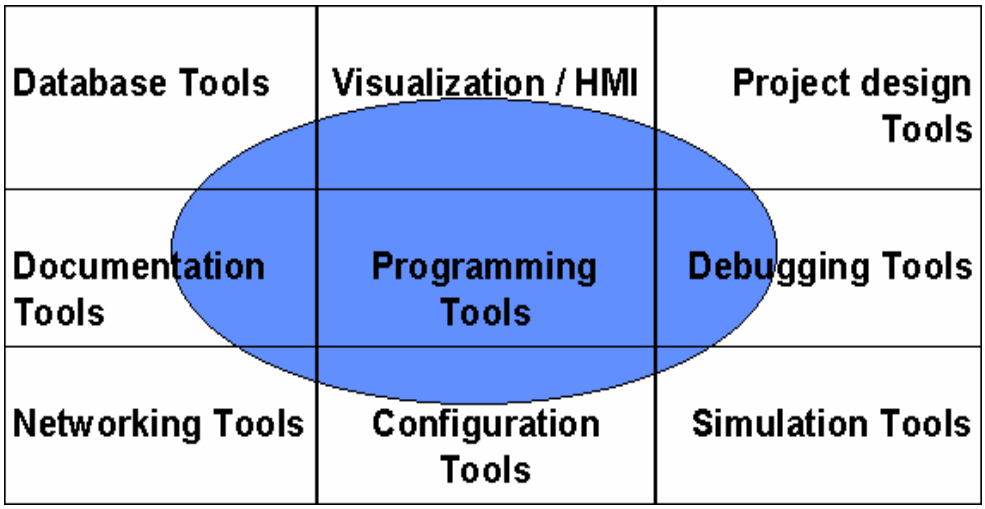


Figura 1. Contesto di inserimento dell' XML Formats for IEC 61131-3. (*Tratta dal sito web di PLCopen: www.plcopen.org*)

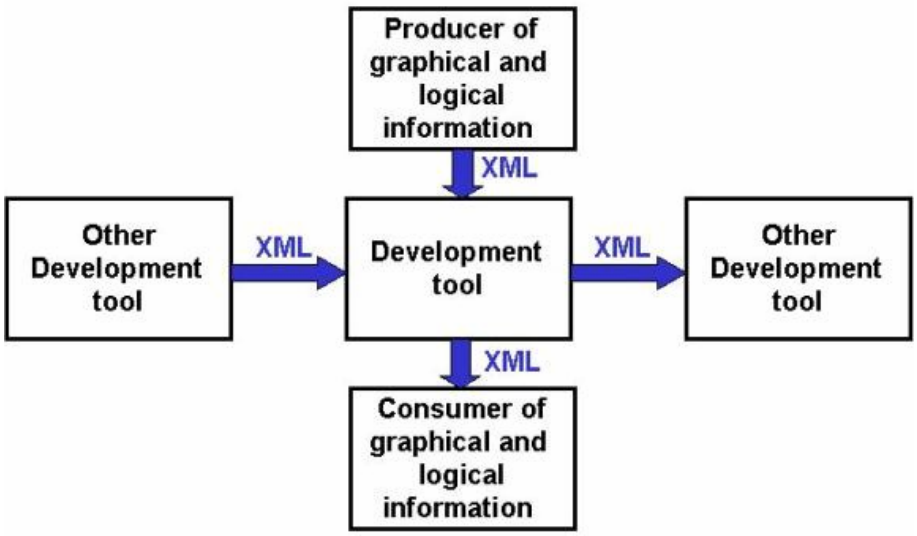


Figura 2. Possibili casi d'uso dell' XML Formats for IEC 6931-3. (*Tratta dal documento 'XML Formats for IEC 61131-3 Version 1.01 – Official Release'*)

Lo scambio di informazioni tra i diversi tool avviene attraverso file xml. Un caso d'uso emblematico può essere il seguente: con un tool di editing grafico (nella figura

Producer of graphical and logical information) si può produrre una libreria di programmi, blocchi funzionali e funzioni. La libreria può essere esportata in un file xml. Un tool per lo sviluppo di progetti (*Development tool)* può importare la libreria verificandone la validità attraverso un *validating XML parser*. Una volta definita attraverso questo tool la configurazione dell'ambiente di esecuzione (task, variabili globali, connessioni tra i dispositivi,…) e delle istanze dei programmi della libreria, il progetto completo può essere importato, sempre previa validazione, da un tool di simulazione e poi di compilazione. (*Consumer of graphical and logical information*). In questo esempio il formato costituisce il "collante" tra le diverse fasi di sviluppo di un intero progetto di automazione realizzato con l'ausilio di diversi strumenti, eventualmente appartenenti a diverse piattaforme di sviluppo proprietarie o non.

XML e XML Schemas

L'XML nasce come meta-linguaggio per la definizione dei cosiddetti linguaggi di markup, fornendo un insieme standard di regole sintattiche per modellare il contenuto dei documenti. Queste regole sono indipendenti dalla piattaforma hardware e software facendo dell'XML un formato universale per lo scambio di dati. Un documento XML [7,9] ha una struttura gerarchica composta da elementi e dai loro eventuali attributi. L'XML non fornisce direttamente gli strumenti per definire la rappresentazione dei tipi di dato e la composizione della struttura gerarchica. Per fare ciò è possibile utilizzare diverse tecnologie, tra cui gli schemi XML, che sono a loro volta definiti attraverso il linguaggio XML. La correttezza di un documento XML può essere verificata attraverso l'utilizzo di particolari software detti parser che analizzano il documento applicando le regole specificate nel relativo schema.

Lo schema XML Formats for IEC 61131-3

Lo schema XML Formats for IEC 61131-3 [3,8] definisce la struttura di un documento XML contenente tutti i dati ed il codice di un progetto d'automazione sviluppato secondo lo standard IEC 61131-3.
In particolare l'elemento *project* rappresenta l'elemento radice (*root node*) del

documento e contiene quattro elementi: *fileHeader*, *contentHeader*, *types* e *instances.*

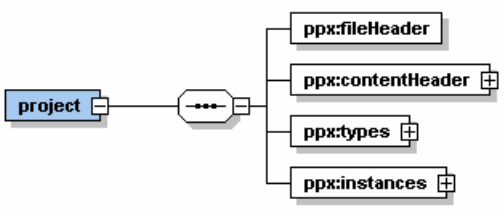


Figura 1 – Vista parziale dell'elemento *project* dello schema XML for IEC 61131-3

L'elemento filerHeader

L'elemento *fileHeader* non contiene ulteriori elementi, ma solo attributi che fanno riferimento alla *compagnia produttrice* e al *prodotto* contenuto nel file xml di import/export del progetto. Essi sono i seguenti:

- *companyName*
- *companyURL*
- *productName*
- *productVersion*
- *productRelease*
- *creationDateTime*
- *contentDescription*

Gli attributi *companyName*, *productName*, *productVersion* e *creationDateTime* sono obbligatori, gli altri possono essere omessi.

L'elemento contentHeader

L'elemento *contentHeader* contiene le informazioni generali relative al progetto. Gli attributi sono:

- *Name*
- *ModificationDateTime*
- *Organization*
- *Version*
- *Author*
- *Language*

E' richiesto solo l'attributo *name*. La struttura dettagliata di *conterHeader* è riportata in figura. *ContentHeader* contiene inoltre l'elemento *Comment*, atto a contenere eventuali commenti, e l'elemento *coordinateInfo*. Quest'elemento contiene le informazioni per il mapping delle coordinate grafiche degli oggetti utilizzati dai linguaggi grafici (i linguaggi LD, FBD ed SFC) quando il file viene esportato ed importato tra sistemi diversi. Attraverso gli elementi scaling relativi ai tre linguaggi è possibile scalare proporzionalmente le coordinate sui piani bidimensionali di disegno per adattare l'eventuale diversità dell'unità di misura di disegno utilizzata.

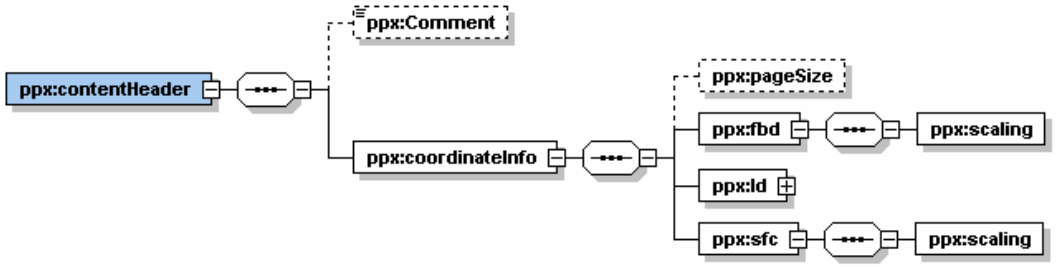


Figura 3. Struttura dell'elemento *conterHeader*

L'elemento types

L'elemento *types* contiene le definizioni dei tipi di dati utilizzati nel progetto (elemento *dataTypes*) e le definizioni delle POUS (elemento *POUS*).

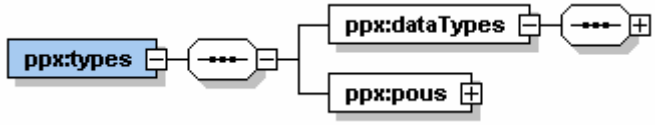


Figura 4. Struttura dell'elemento *types*

dataTypes

L'elemento *dataTypes* può contenere più elementi *dataType*, ognuno dei quali definisce il nome del tipo di dato (attributo *name*) ed il relativo tipo di base ed eventualmente il valore iniziale.

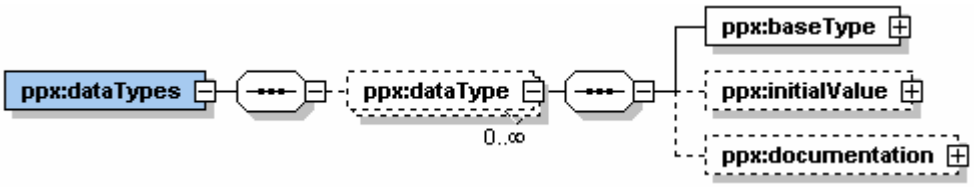


Figura 5. Struttura dell'elemento datatypes

Il tipo di base è una *scelta* tra uno dei gruppi definiti nello schema: *elementaryTypes*, *derivedTypes* e *extended*. A sua volta ogni gruppo definisce una scelta tra i tipi che ne fanno parte.

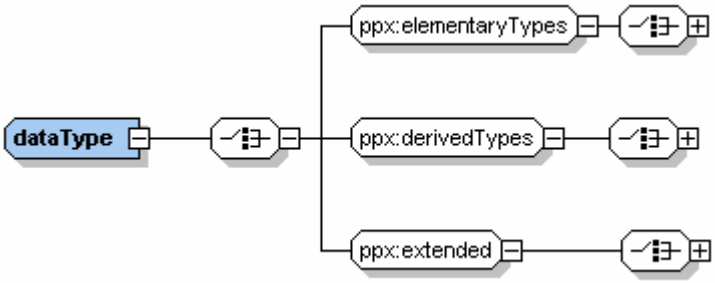


Figura 6. Struttura dell'elemento *dataType*

Il gruppo *elementaryTypes* contiene i seguenti tipi di base, così come definiti dallo standard IEC 61131-3:

- *BOOL*
- *BYTE*
- *WORD*
- *DWORD*
- *LWORD*
- *SINT*
- *INT*
- *DINT*
- *LINT*
- *USINT*
- *UINT*
- *UDINT*
- *ULINT*
- *REAL*
- *LREAL*
- *TIME*
- *DATE*
- *DT*
- *TOD*
- *String*
- *Wstring*

Il gruppo derivedTypes è composto dai seguenti tipi sempre definiti dallo standard IEC 61131-3:

- *ARRAY*
- *DERIVED*
- *ENUM*
- *SUBRANGESIGNED*
- *SUBRANGEUNSIGNED*
- *STRUCT*

In aggiunta ai tipi di base definiti dallo standard vi è il tipo *pointer*, unico tipo del gruppo *extended*. Esso esprime un puntatore ad un tipo di base, dunque nella definizione occorre dichiarare il tipo a cui punta nel proprio elemento *baseType*.

POUS

Questo elemento contiene tutte le definizioni delle unità organizzative di programma del progetto. Ognuna è contenuta nel relativo elemento POU contenuto a sua volta nell'elemento POUS. Il nome della POU è espresso dall'attributo *name*, mentre il tipo (*program*, *fuctionalBloc* o *function*) è espresso dall'attributo *pouType*. Una POU contiene in sequenza i seguenti elementi:

- *interface*
- *actions*
- *transitions*
- *body*
- *documentation*

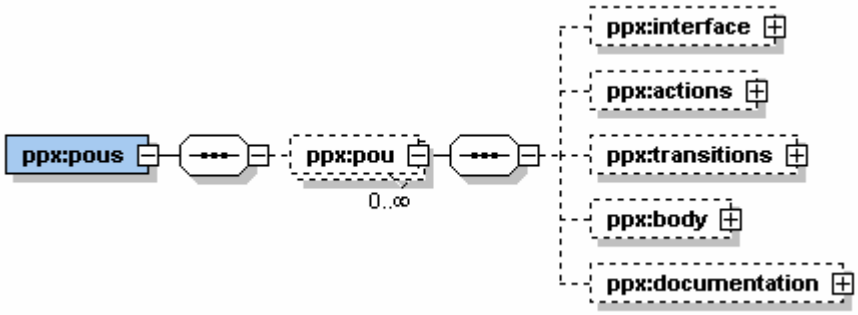


Figura 7. Struttura dell'elemento *POUS*

L'elemento *interface* rappresenta l'interfaccia della POU. Contiene le liste delle variabili sia a visibilità locale che esterna, ed eventualmente un elemento che ne definisce il tipo restituito. Quest'ultimo è l'elemento *returnType* ed è di tipo *dataType*. Oltre all'elemento *documentation*, *interface* contiene le seguenti liste di variabili: *LocalVars, che* contiene le variabili locali, *tempVars,* che contiene le variabili temporanee, *inputVars, outputVars* e *inOutVars,* che contengono rispettivamente le variabili di ingresso, le variabili di uscita e quelle sia d'ingresso che di uscita. *ExternalVars* contiene le variabili esterne, cioè quelle dichiarate esternamente alla POU ma visibili anche all'interno di essa. Ad esempio le variabili globali di una risorsa possono essere visibili all'interno di tutte le POUS istanziate in essa, ma per utilizzarle all'interno di una POU è necessario dichiararle *external*. L'elemento *globalVars* contiene tutte le variabili con visibilità globale definite all'interno della POU. Tali variabili globali possono essere definite solo nei *programs*

e sono visibili in tutte le POU esistenti nel programma in cui sono dichiarate come *external*. Infine *accessVars* contiene le variabili che hanno accesso diretto alle locazioni di memoria.

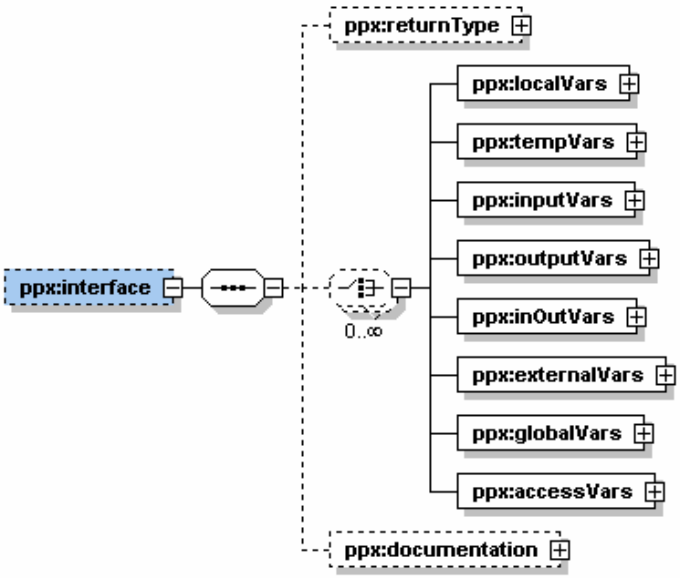


Figura 8. Struttura dell'elemento *interface* della POU

Una POU può contenere inoltre un *body*, una lista di azioni (*actions*) ed una lista di transizioni (*transition*s). Il body contiene l'implementazione di una POU, una azione o una transizione attraverso uno dei cinque linguaggi dello standard, dunque l'elemento *body* contiene una *scelta* tra uno dei cinque elementi: IL, ST, FBD, LD e SFC.

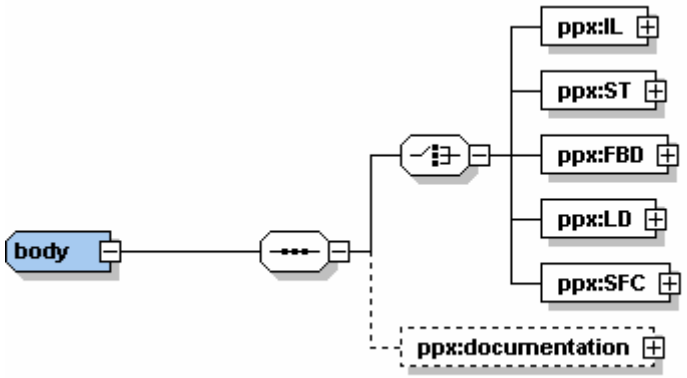


Figura 9. Struttura dell'elemento *body*

L'elemento *body è* presente anche negli elementi *action* e *transition* contenute nelle liste di azioni e di transizioni, dove ognuna è identificata da un nome diverso. Dichiarando un'azione o una transizione all'esterno del body di una POU si ha la possibilità di riutilizzarla più volte all'interno facendo semplicemente riferimento ad essa tramite il nome. I cinque elementi che identificano i diversi linguaggi contengono gli elementi che rappresentano l'implementazione di tipo testuale dei linguaggi IL e ST o che rappresentano gli oggetti per l'implementazione di tipo grafica dei linguaggi FBD, LD e SFC. Gli elementi di tipo testuale sono di tipo *formattedText*, che rappresenta un testo formattato secondo le specifiche XHTML 1.1.

Ogni oggetto grafico definito da uno dei linguaggi FBD, LD e SFC è rappresentato da un diverso elemento ed è contenuto in uno dei quattro gruppi: *commonObjects*, *fbdObjects*, *ldObjects* ed *sfcObjects*. Inoltre dispone dell'attributo *localId* di tipo *unsigned-long* che lo identifica univocamente all'interno di una POU. Nell'implementazione grafica di un programma gli oggetti generalmente sono collegati tra essi attraverso linee. Ad esempio una fase di un SFC è collegata alle transizioni o ad un blocco di azioni, un contatto di un programma in Ladder è collegato ad altri contatti e bobine, nell'FBD un blocco è collegato ad un altro blocco, ecc. Un collegamento grafico è rappresentato dall'elemento *connection*.

Gli oggetti grafici dispongono dell'elemento *connectionPointIn* che contiene tutti gli elementi *connection* che rappresentano le connessioni in ingresso. A sua volta un elemento *connection* dispone dell'attributo obbligatorio *refLocalId* che indica il *localId* dell'oggetto da cui inizia la connessione (uscita). Un elemento *connection* può contenere una serie di elementi *position* che ne esprimono il percorso grafico sul piano. Il gruppo *commonObject* contiene gli elementi che definiscono gli oggetti comuni ai tre linguaggi grafici.

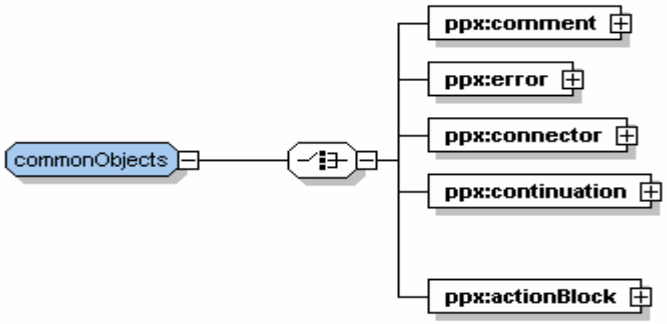


Figura 10. Il gruppo *commonObjects*

Tra essi vi è l'elemento *actionBlock* che rappresenta un blocco di azioni all'interno di un *body*.

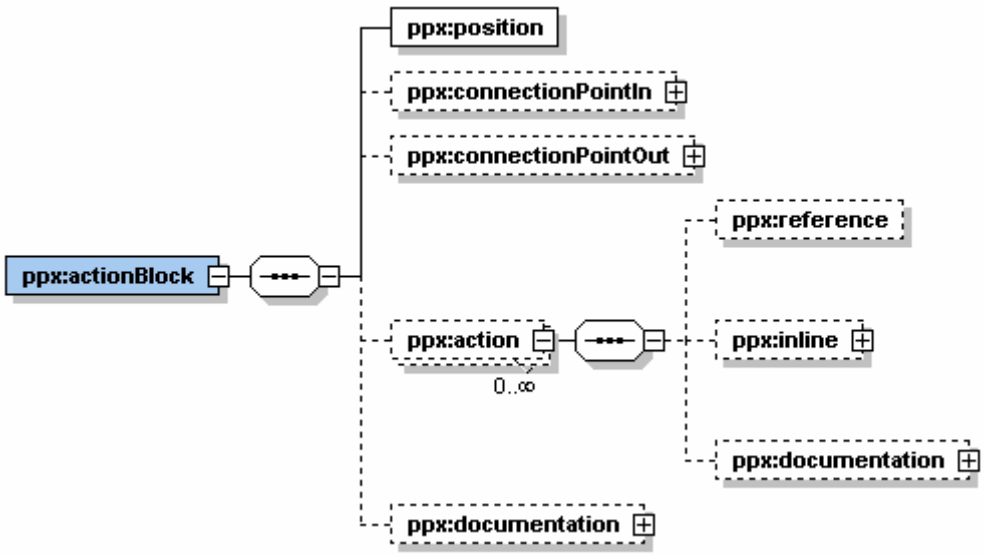


Figura 11. Struttura dell'elememto *actionBlock*

Un blocco di azioni è definito, oltre che dagli elementi di carattere grafico, da una lista di azioni (*action*). Un elemento *action* interno ad un *body* (in questo caso è interno al *body* della *POU*) può far riferimento ad un'azione definita nella lista di azioni della POU (attraverso l'elemento *reference*) o può essere implementata direttamente nel proprio elemento *inline* che è di tipo *body*.

In particolare gli oggetti del linguaggio Ladder sono contenuti negli elementi del gruppo *ldObjects*. Essi sono quattro: *contact* (contatto) e *coil* (bobina), ognuno dei quali, tra l'altro, contiene l'elemento *variable,* che esprime il riferimento ad una variabile booleana, *leftPowerRail* (parte sinistra dell'alimentazione) e *rightPowerRail* (parte destra dell'alimentazione).

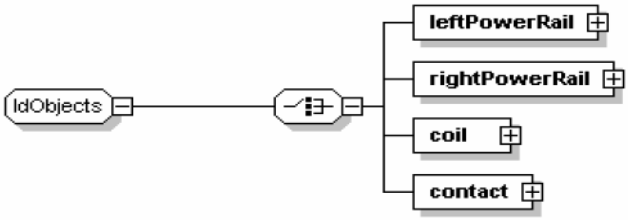


Figura 12. Il gruppo ldObjects

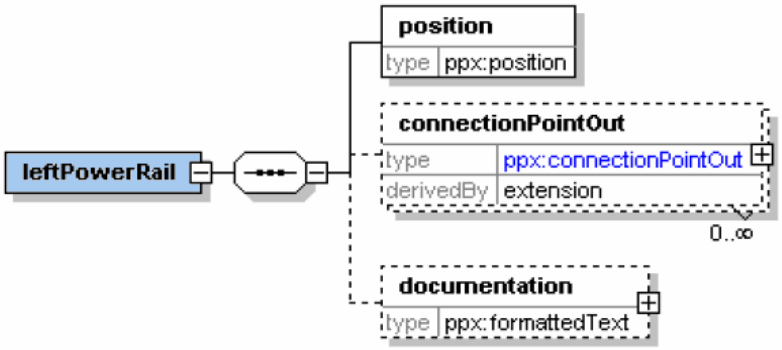


Figura 13. Il Gruppo LeftPowerRail

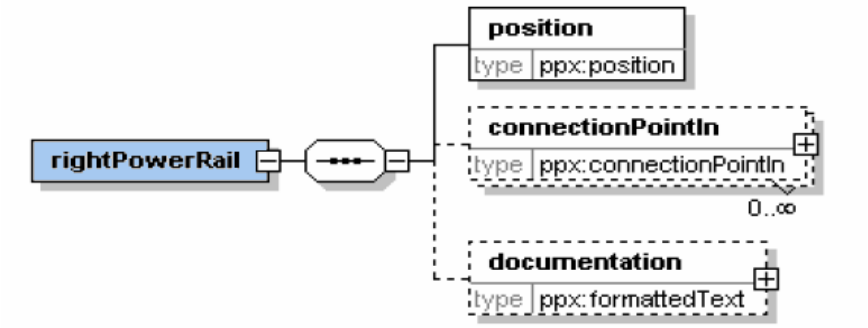


Figura 14. Il gruppo RightPowreRail

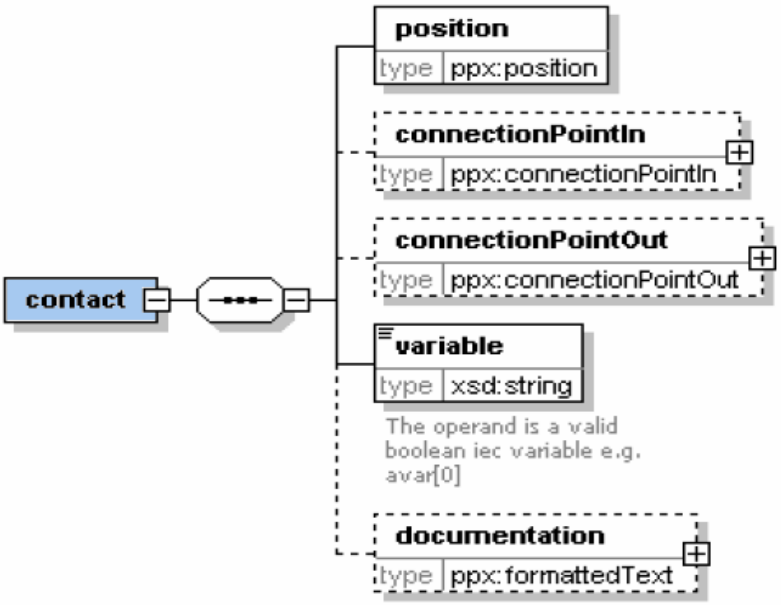


Figura 15. Il Gruppo Contact

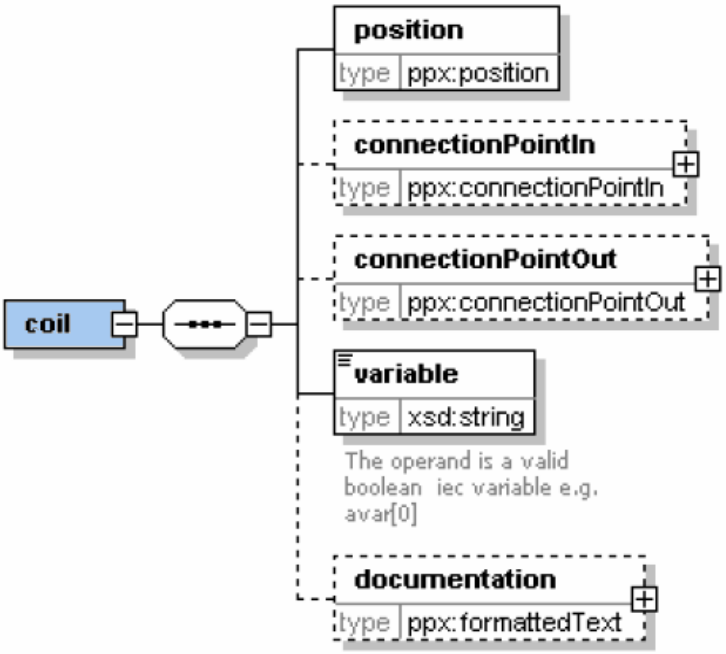


Figura 16. Il Gruppo Coil

Elemento instances

L'elemento *instances* contiene le informazioni relative alle configurazioni degli ambienti in cui sono eseguiti i programmi, quindi dei dispositivi (*resource*) con i relativi compiti (*task*) e le istanze dei programmi (*pouInstance*). E' possibile definire più configurazioni diverse con i propri gruppi di risorse e variabili globali. L'elemento *instances* dunque contiene l'elemento *configurations* che a sua volta può contenere una lista di elementi *configuration*.

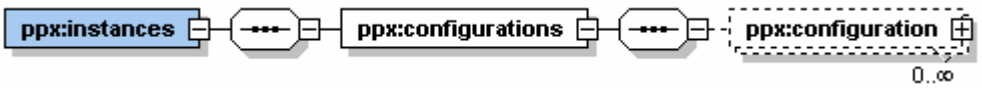


Figura 17. Struttura dell'elemento *instances*

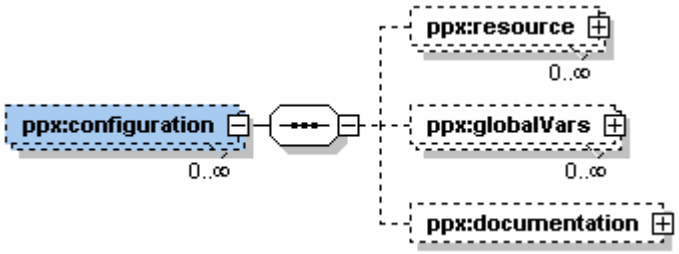


Figura 18. Struttura dell'elemento *configuration*

Configuration contiene un insieme di elementi *resource* e di liste di variabili globali (*globalVars*). Le variabili di queste liste sono visibili all'interno di tutte le risorse appartenenti alla configurazione e quindi permettono la comunicazione tra di esse.
Una risorsa è un dispositivo fisico in grado di eseguire i programmi. Deve contenere le definizioni dei task con le istanze dei programmi e le variabili globali per la comunicazione internamente alla risorsa. L'elemento *resource* quindi può contenere più elementi *task*, *pouInstance* e *globalVars*.

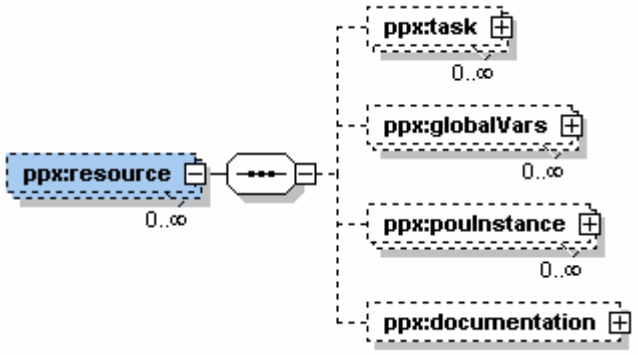


Figura 19. Struttura dell'elemento *resource*

Un elemento *task* definisce un compito del dispositivo. Esso dispone degli attributi *nome*, *priority*, *interval* e *single*. Gli ultimi tre sono relativi alla schedulazione dei task da parte del dispositivo: *priority* esprime la priorità con un valore che varia da 0 a 65535 (0 è la priorità massima), *interval* determina l'intervallo ciclico di esecuzione e *single* rappresenta il riferimento al valore che indica che il task è pronto per essere eseguito. *Task* contiene una lista di elementi *pouInstance* che rappresentano l'istanza della definizione di un *program* o un *functionBlock*. Essi vengono eseguiti quando viene eseguito il task. Gli elementi *pouInstance* possono essere contenuti anche direttamente nella *resource*. In questo caso rappresentano le

istanze non associate a nessun *task*, quindi a priorità più bassa e con intervallo di esecuzione nullo. Infine l'elemento *globalVars* della risorsa contiene le liste delle variabili globali che permettono la comunicazione tra le istanze delle POU della risorsa.

Struttura generale definita dallo schema

Nella pagina seguente si riporta la struttura generale definita dello schema.

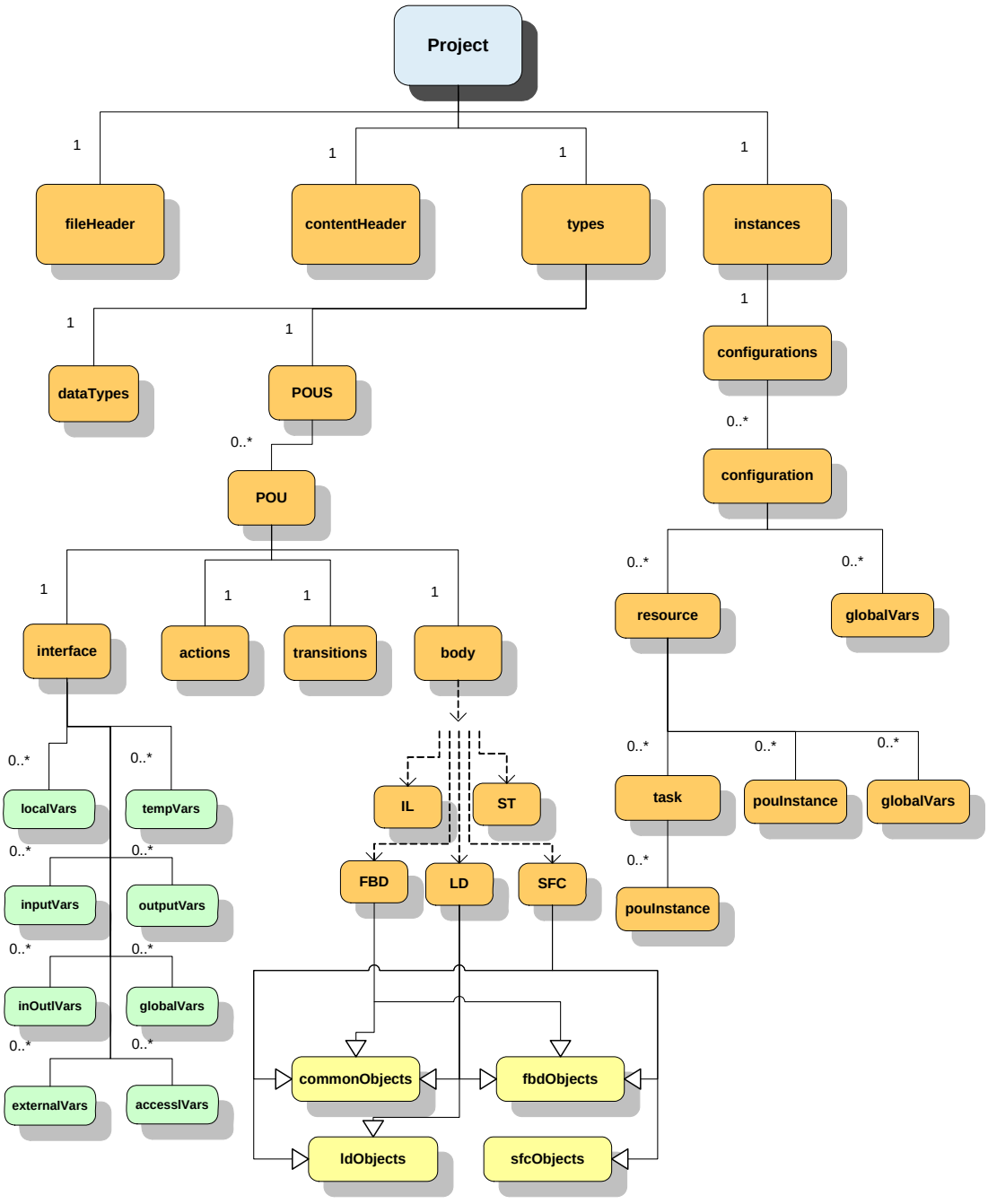


Figura 20. Struttura generale definita dallo schema

Capitolo 3

Architettura del progetto

In questo capitolo verrà illustrata l'architettura del progetto e come si è intervenuti per aggiungere i moduli sviluppati in questo lavoro di tesi.

Piattaforma e strumenti di sviluppo

Il tool è stato sviluppato su piattaforma Microsoft .NET Framework. [11,12,13]Questa nuova tecnologia fornisce una serie di compilatori per i principali linguaggi supportati da Microsoft ed un ambiente di esecuzione detto Common Language Runtime (CLR). Quest'ultimo richiama in parte la tecnologia Java. Il codice generato in fase di compilazione non dipende dal tipo di processore su cui deve essere eseguito. I compilatori di codice per piattaforma .NET Framework generano lo stesso tipo di codice detto Mil (Microsoft Intermediate Language) che viene gestito in fase di esecuzione dal CLR. Esso dispone di un compilatore JIT (Just In Time) che a tempo di esecuzione traduce il codice nel linguaggio macchina del sistema su cui viene eseguito. Generalmente la traduzione di un'istruzione avviene alla prima esecuzione.

In fase di sviluppo è stato utilizzata la versione object-oriented del linguaggio VB.NET. Inoltre sono state utilizzate diverse classi messe a disposizione dalla libreria di .Net Framework, in particolare per l'implementazione delle funzioni di

import/export e di validazione dei progetti e per la realizzazione dell'interfaccia utente.

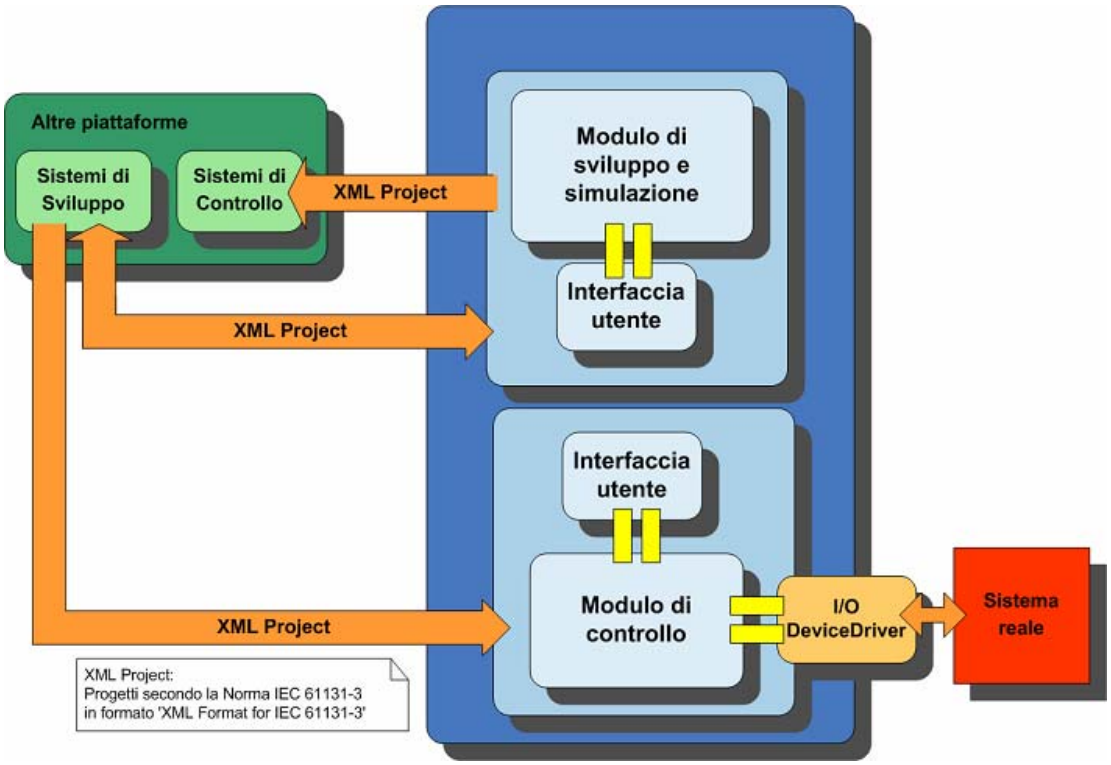


Figura 21. Diagramma a blocchi dell'architettura del tool UniSim

In Figura 21 è riportato un diagramma a blocchi della struttura di UniSim. Il tool è composto da tre componenti principali: l'*interfaccia utente* per lo sviluppo del progetto, il *simulatore* per il test e la validazione, ed il *modulo di controllo*, che consente di interfacciarsi con l'hardware di I/O. Il simulatore ed il modulo d'interfaccia con l'hardware di acquisizione implementano entrambi un algoritmo di schedulazione non-preemptive con priorità per l'esecuzione dei vari task ciclici.

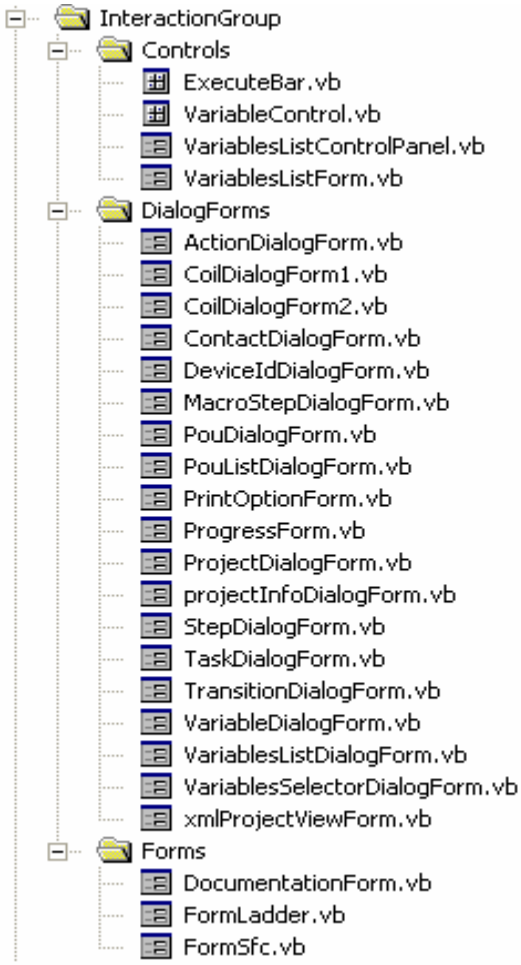
3.1 Architettura generale

L'architettura software di questa applicazione rispecchia il modello ad oggetti definito dallo schema *XML Formats for IEC 61131-3* descritto nel capitolo

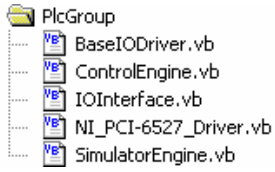
precedente che, a sua volta, deriva dalla struttura specificata dallo standard IEC 61131-3, infatti dal punto di vista progettuale il tool è stato realizzato seguendo un paradigma object oriented. Rispecchiando lo standard IEC 61131-3, l'architettura identificata una struttura ad oggetti di tipo gerarchico, così come definito dallo schema *XML Formats for IEC 61131-3*. La fase di progettazione dunque ha prodotto un modello che in gran parte riprende tale struttura. Le classi si possono suddividere nei seguenti gruppi:

- *InteractionGroup*: contiene le classi per l'implementazione dell'interfaccia grafica;
- *ProjectGroup*: contiene le classi per l'implementazione dell'intero progetto e dell'interfaccia di import/export. E' composto dai seguenti sottogruppi:
    - *ProjectInfoGroup:* contiene le classi finalizzate alla memorizzazione delle informazioni relative al progetto;
    - *TypeGroup:* contiene le classi che implementano i tipi di dati e le unità organizzative di programma (POU, Program Organization Units);
    - *InstencesGroup:* contiene le classi che implementano gli oggetti dei livelli più alti del modello software dello standard.
- *PLCGroup:* contiene le classi che implementano il motore di simulazione, il motore di controllo e l'interfaccia verso i dispositivi di I/O.

InteractionGroup



PLCGroup



ProjectGroup

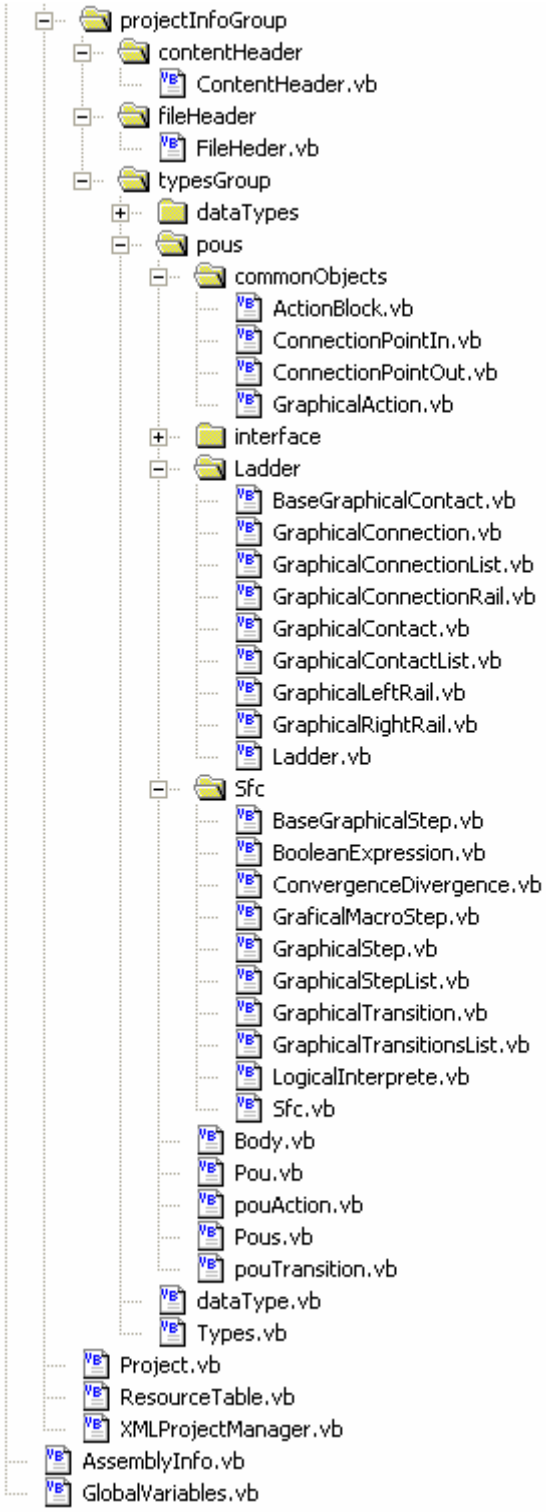


Figura 22. Suddivisione delle classi in gruppi

Le classi sono contenute in un'unica libreria utilizzata dai moduli eseguibili delle due applicazioni. Ognuno di questi moduli implementa esclusivamente le rispettive interfacce utente.

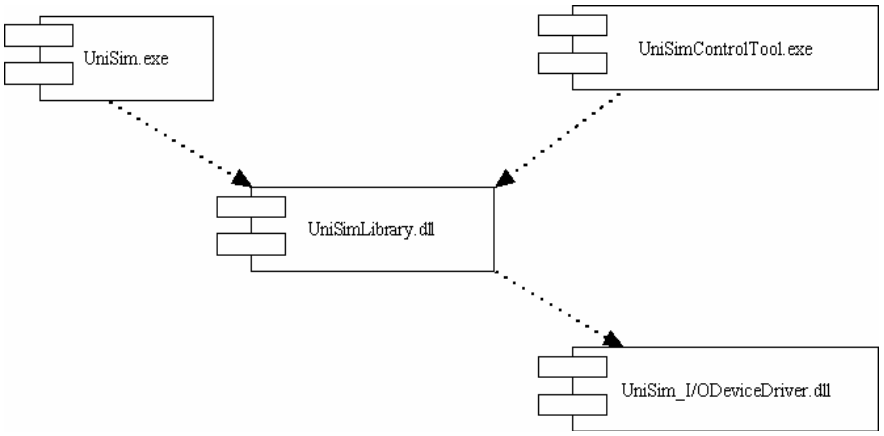


Figura 23. Component diagram

I compiti relativi ad un determinato oggetto sono implementati tutti dalla relativa classe, dunque le operazioni sono realizzate attraverso una serie di collaborazioni tra classi dello stesso livello gerarchico o di livelli differenti. Il ciclo di simulazione, ad esempio, viene controllato da un oggetto di tipo *SimulatorEngine* e viene realizzato attraverso una serie di chiamate innestate a metodi forniti dalle classi di livello inferiore che rappresentano i task, le unità di programma e gli oggetti utilizzati per la relativa scrittura (fasi, transizioni e azioni dell'SFC, connessioni, contatti LADDER, variabili, ecc.). In modo analogo vengono eseguiti l'import e l'export in formato xml del progetto.

3.2 Implementazione del modello di progetto

Il modello di progetto viene implementato dalle classi del *ProjectGroup*. Al vertice della gerarchia vi è la classe *project*. Da questo punto in poi viene ripresa la gerarchia di elementi definita dello schema dell'XML Format for IEC 61131-3. La classe *project* contiene i riferimenti ad oggetti istanze delle classi *fileHeader*, *contentHeader*, *types* e *instances*.

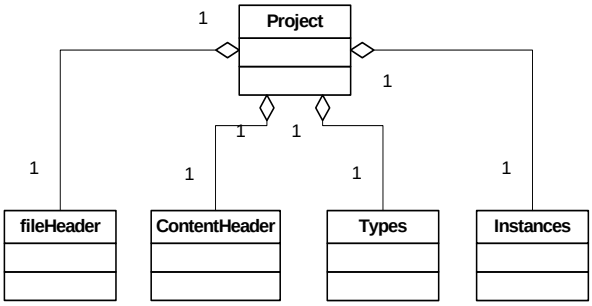


Figura 24. Class diagram 1

Le classi *fileHeader* e *contentHeader* contengono gli attributi relativi alle informazioni contenute dai corrispondenti elementi dello schema xml. La classe *types* e la classe *instances* implementano gli oggetti rappresentati dagli elementi omonimi dello schema che stanno al vertice di due sottoalberi: del primo fanno parte le classi per la definizione e l'implementazione dei tipi (dati e programmi), mentre del secondo le classi che definiscono ed implementano le istanze di programmi e variabili e le configurazioni dell'ambiente di esecuzione.

Implementazione dei tipi

Un oggetto di tipo *types* contiene le istanze di due liste di oggetti: m_*dataTypes* atta

a contenere gli oggetti di tipo *dataType*, ed *m_Pous*, che contiene gli oggetti di tipo *Pou*. Un oggetto *dataType* definisce un tipo di dato, mentre un oggetto *Pou* rappresenta un'unità organizzativa di programma. *m_Pous* è un istanza della classe *Pous*. Questa classe eredita la classe A*rrayList* [11,13], che implementa una lista dinamica e fornisce i metodi per l'aggiunta, la rimozione e la ricerca in lista delle istanze dalla classe *Pou* .

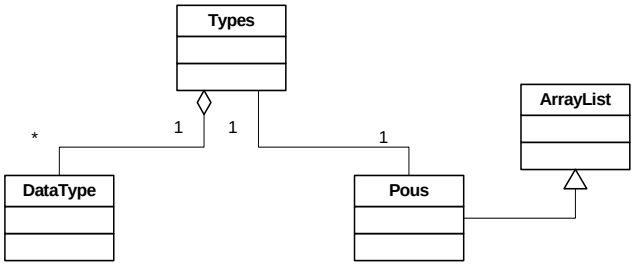


Figura 25. Class diagram 2

Le unità organizzative di programma (POU)

Gli oggetti di tipo *Pou* sono creati ed aggiunti alla relativa lista quando l'utente richiede la definizione di un nuovo programma. Ogni oggetto di questo tipo contiene gli attributi che ne definiscono il nome e il tipo (tipo enumerativo *EnumPouType* con membri *program*,*f*unction e *functionBlock*). Allo stato attuale il tool permette la creazione di POU solo di tipo *program*.

Un oggetto di tipo *Pou* contiene inoltre un'istanza della classe *pouInterface*, *m_pouInterface*, che implementa l'interfaccia della POU. Inoltre dispone di due liste dinamiche, *m_actions* e *m_transitions*, atte a contenere gli oggetti *pouAction* e *pouTransition*, ed di un'istanza della classe *body* (*m_body*). Un oggetto di tipo *body* contiene i riferimenti agli oggetti che rappresentano il corpo del programma della POU. L'attributo *m_BodyType è* di tipo *EnumBodyType*, così definito:

```
Public Enum EnumBodyType
   tSFC
   tST
   tLD
   tIS
   tFBD
End Enum
```

Ogni membro dell'enumerazione corrisponde ad un linguaggio di programmazione utilizzato per la scrittura di un programma contenuto nel body. Allo stato attuale sono disponibili solo le classi per l'implementazione del linguaggio SFC e del LADDER. L'oggetto *m_sfc* contenuto nella classe body è un'istanza della classe *sfc* e rappresenta il programma scritto nell'omonimo linguaggio, analogamente l'oggetto *m_ladder* anch'esso contenuto nella classe body è un'istanza della classe *ladder* e ne rappresenta l'omonimo programma.

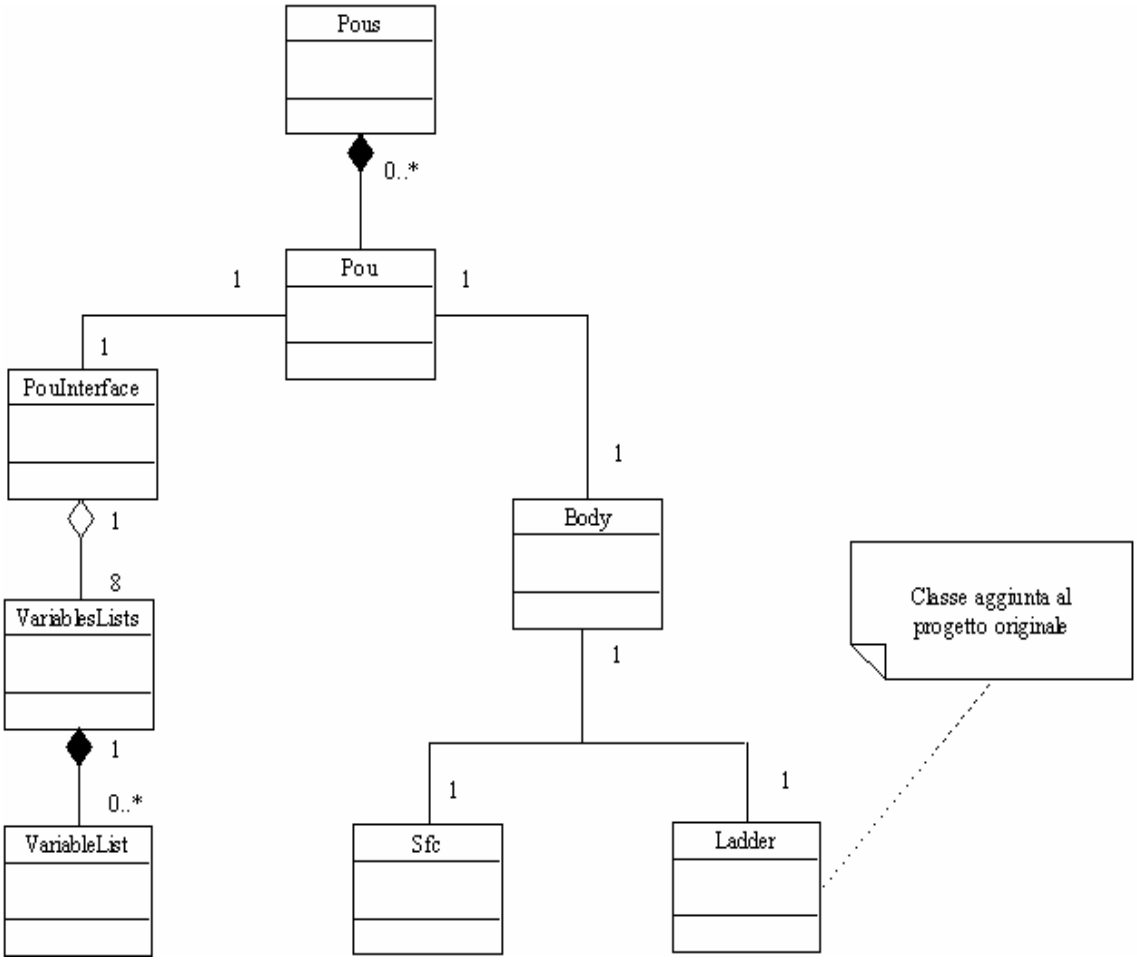


Figura 26. Class Diagram 3

Implementazione dei programmi in linguaggio LADDER

Unisim sin dall'inizio è stato sviluppato in modo altamente modulare, agevolando così un eventuale successivo ampliamento. Proprio da questo principio si è partiti per inserire le classi necessarie per permettere al software di sviluppare progetti completi anche in linguaggio Ladder. Nella fattispecie è stata introdotta la classe *Ladder* che dispone degli attributi e dei metodi per l'editing dei programmi, per l'esecuzione degli stessi in fase di simulazione, per il relativo monitoraggio dell'evoluzione dello stato nonché per l'import ed export XML. Le altre classi introdotte per implementare gli specifici oggetti definiti dal linguaggio sono: *Contact* e *Coil* (classe *GraphicalContact*), *Connection* (classe *GraphicalConnection*) e le linee di alimentazione *Left* e *Right Power Rail* (classi *GraphicalLeftRail* e *GraphicalRightRail*). All'atto della definizione di una connessione, l'utente stabilisce semplicemente il punto di inizio (la linea sinistra di alimentazione o l'uscita di un contatto/bobina) e quello di fine (l'ingresso di un contatto/bobina o la linea destra di alimentazione) ed automaticamente viene disegnato il collegamento.

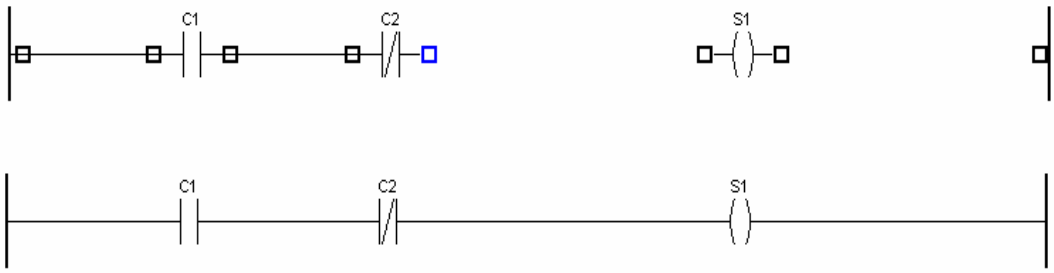


Figura 27. Disegno delle connessioni tra componenti (prima e dopo)

Le classi *GraphicalContact, GraphicalLeftRail* e *GraphicalRightRail* ereditano la classe astratta *BaseGraphicalContact* e ne ridefiniscono i metodi virtuali. *BaseGraphicalContact* contiene gli attributi e i metodi che ne definiscono il comportamento comune. Insieme alla classe *GraphicalConnection* dispongono di una serie di attributi a carattere grafico che ne definiscono la dimensione sul piano di disegno, i colori ed il font. Un riferimento ad un oggetto di tipo G*raphics* (rappresenta il piano di disegno sul video) determina l'oggetto su cui disegnare.

Questo parametro viene passato ai rispettivi costruttori all'atto della creazione dell'oggetto e viene memorizzato nell'attributo m_*GrapToDraw*. Le suddette classi inoltre dispongono dei metodi per le gestione delle operazioni a carattere grafico: disegno, spostamento, cancellazione, calcolo dell'area occupata, ecc.

La classe *BaseGraphicalContact* contiene l'attributo booleano *Drawstate* che stabilisce se devono essere disegnati i rispettivi indicatori di stato che riferiscono il valore della variabile booleana associata al contatto o alla bobila.

All'atto della creazione di un contatto, un'interfaccia grafica (*CoilDialogForm*) acquisisce i parametri necessari: nome, tipo (contact o coil), variabile associata, qualificatore che identifica il tipo di contatto (N.C., N.O., ecc.), i quali vengono passati al rispettivo costruttore per istanziare l'oggetto di tipo *GraphicalContact* selezionato e disegnarlo a video sul piano di disegno.

La classe *FormLadder* che rappresenta l'interfaccia grafica, implementa i metodi *AddRail()* e *RemoveRail()* che permettono rispettivamente di disegnare ed eliminare le linee d'alimentazione. Il disegno delle Power Rail è graduale, ossia ogni volta che viene invocato *AddRail()*, si aggiungono due segmenti verticali in successione partendo dall'ultimo già disegnato; questo per rendere più concentrati quei "circuiti" con un numero limitato di *rung*. Analogamente il metodo *RemoveRail()* rimuove i segmenti di linee inutilizzate partendo dal basso, ossia quelli ai quali non è collegato alcun *rung*.

Tutti gli oggetti di tipo *GraphicalContact* e *GraphicalConnection* sono contenuti rispettivamente in due liste dinamiche di un oggetto *Ladder*, *m_GraphicalContactList*, istanza della classe *GraphicalContactList, e m_GraphicalConnection*, istanza della classe *m_GraphicalConnectionList*. Queste due classi ereditano la classe *ArrayList* e dispongono degli attributi e dei metodi per l'aggiunta, la rimozione e il disegno dei contatti e delle connessioni, nonché dei metodi per l'implementazione dell'esecuzione del Ladder.

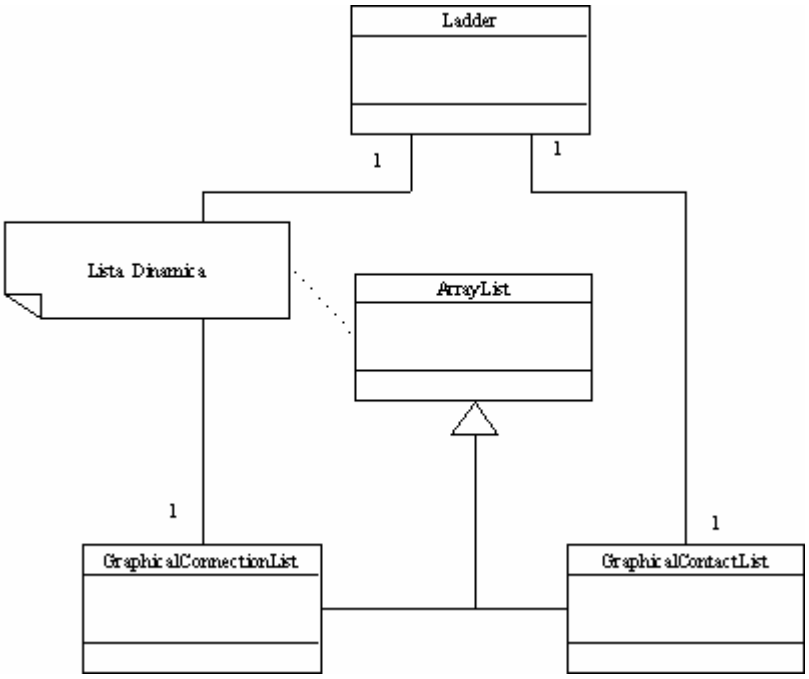


Figura 28. Class diagram 4

Un oggetto di tipo *GraphicalConnection* dispone inoltre degli oggetti *m_PreviousGraphicalContactList* e *m_NextGraphicalContactList*, entrambi istanze della classe *GraphicalContactList*. Queste liste contengono, rispettivamente, i riferimenti ai contatti precedenti alla connessione ed ai contatti successive. Il metodo che effettua il disegno della connessione legge le posizioni sul piano dei suddetti contatti e traccia automaticamente i collegamenti tra gli oggetti. Come sarà illustrato successivamente le liste sono inoltre utilizzate dei metodi per l'esecuzione del programma per determinare se la connessione è l'unica ad alimentare un contatto oppure il contatto è alimentato da altri *rung*.

Algoritmo di esecuzione del LADDER

Prima di procedere con la descrizione dei passi dell'algoritmo, verranno illustrati le proprietà fondamentali di ogni componente.

Le classi *GraphicalContact* e *GraphicalLeftRail* ereditano dalla classe astratta *BaseGraphicalContact* e rappresentano rispettivamente i contatti (contact o coil) e la linea di alimentazione sinistra. E' presente, simmetricamente, anche la classe *GraphicalRightRail*, che tuttavia non interagisce ai fini dell'evoluzione

dell'algoritmo. Una caratteristica importante di queste classi è l'attributo booleano *m_Final* che rappresenta il valore di "uscita" di ogni componente.
Per definizione, nella classe *GraphicalLeftRail*, questo viene settato a true al momento della rappresentazione grafica, e resta in tale valore per tutto il periodo di simulazione.

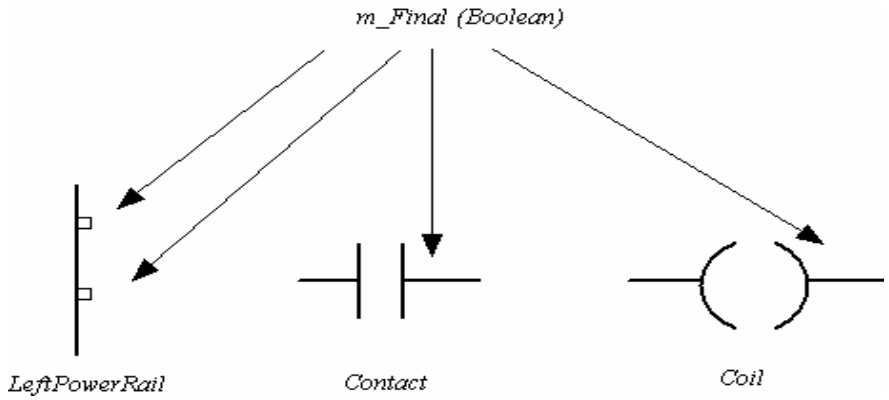


Figura 29. Evidenziazione dell'attributo *m_Final* nei componenti grafici.

Un altro parametro fondamentale per l'esecuzione dell'algoritmo è l'attributo *m_Initial* che rappresenta il valore di "ingresso" di ogni componente, naturalmente non viene utilizzato nella classe *GraphicalLeftRail.*

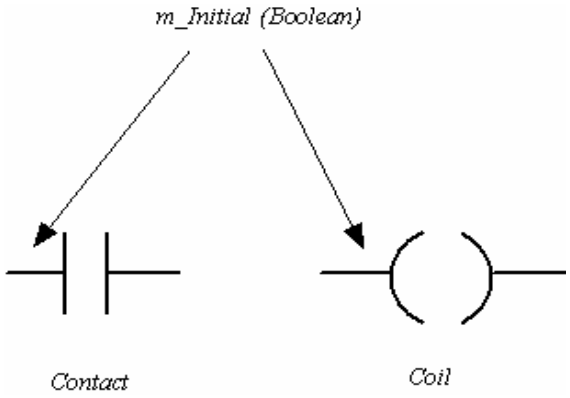


Figura 30. Evidenziazione dell'attributo *m_Initial* nei componenti grafici.

Inoltre, la classe *GraphicalConnection* rappresenta la connessione grafica e logica tra i vari componenti dei *rung*. Essa tramite il metodo pubblico *ReadLeftContact*

legge la lista di componenti collegati alla sua sinistra, *GraphicalContactList,* e invocando la funzione *ReadFinalValue* per ognuno ne acquisisce il valore dell'attributo *m_Final.*

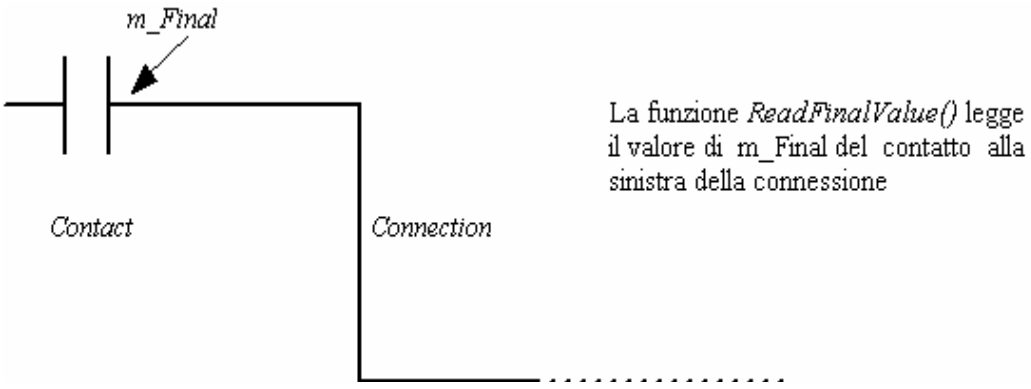


Figura 31. Funzione ReadFinalValue()

Analogamente il metodo *ReadRightContact* legge la lista di componenti alla destra di ogni connessione e la funzione *SetInitialValue(true)* ne setta l'attributo *m_Initial* a true.

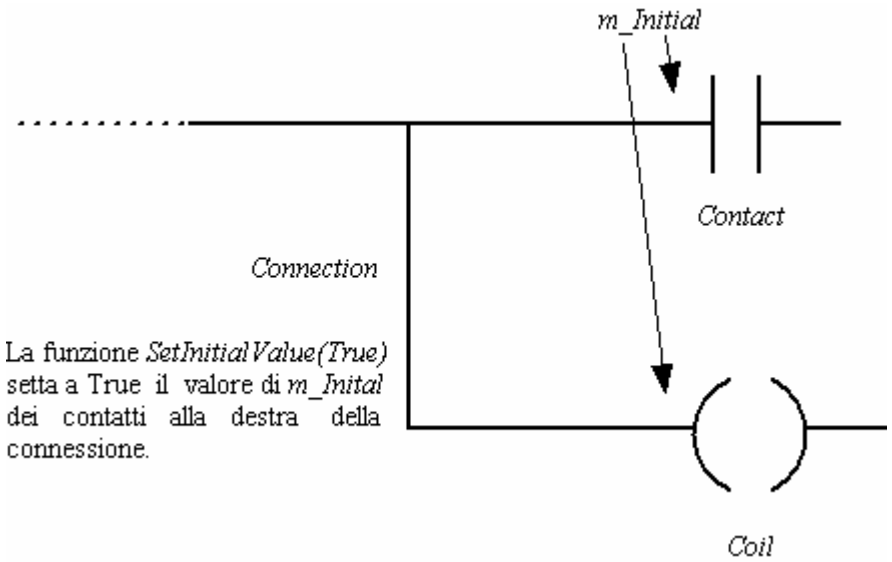


Figura 32. Funzione *SetInitialValue(True)*

L'algoritmo di esecuzione del programma Ladder effettua sostanzialmente due passi:

1. per ogni connessione legge il valore dell'attributo *m_Final* del componente di sinistra, se vale true invoca il metodo *SetInitialValue(true)* settando l'ingresso (*m_Initial*) dei contact o coil alla sua destra, altrimenti non fa nulla, effettuando di fatto una propagazione logica del valore alto attraverso le connessioni;
2. per ogni contact o coil presente, effettua la chiamata al metodo *SetFinalValue* che sintetizza la funzione del particolare contatto, come descritto successivamente, e resetta il valore dell'attributo *m_Initial* tramite chiamata a *ResetInitial()*, questo per annullare una eventuale memoria del contatto.

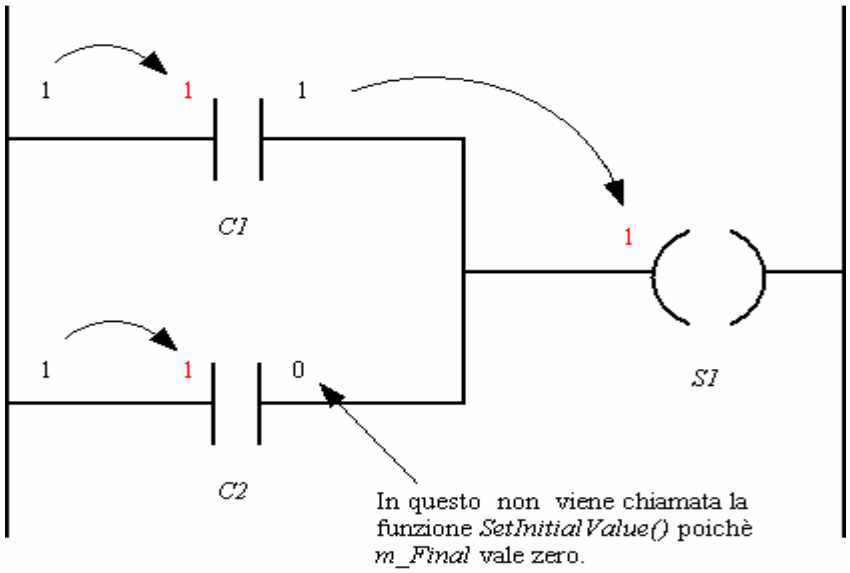
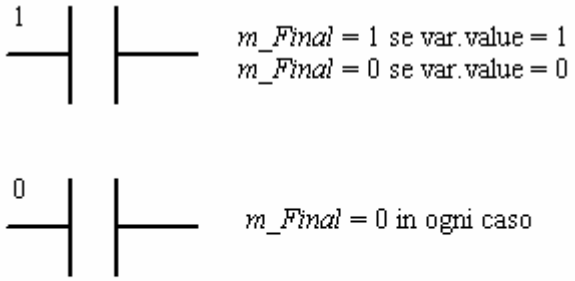
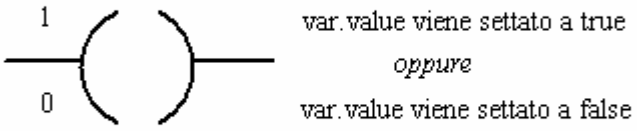


Figura 33. Primo passo dell'algoritmo



Per i Contact la chiamata alla funzione *SetFinalValue()* effettua l'operazione AND tra m_Initial ed il valore della variabile (oppure il valore negato se si tratta di Negative Contact).



Per i Coil la chiamata alla funzione *SetFinalValue()* setta a m_Initial il valore della variabile associata (oppure a NOT *m_Initial* se si tratta di Negative Coil).

Figura 34. secondo passo dell'algoritmo

La funzione *SetFinalValue* richiede in ingresso un *Id* di tipo intero che rappresenta il contatto (contact o coil) ed un qualificatore di tipo string che rappresenta il comportamento del componente (Normal Open, Normal Closed, ecc).

Secondo il tipo di contattato e del comportamento, viene effettuata l'operazione booleana con il valore della variabile associata, settando l'attributo *m_Final* se si tratta di un contact oppure il valore stesso della variabile associata se si tratta di un coil.

Come descritto al primo passo dell'algoritmo, se *m_Final* vale zero, non viene chiamato il metodo *SetInitialValue(true)*, questo per evitare che in caso di contatti in parallelo, qualora uno o più di essi assumesse valore zero per l'attributo *m_Final*, questo non influisse sulla corretta propagazione del segnale, effettuando, di fatto, un'operazione di OR logico. Per questo motivo è infatti necessario resettare l'attributo *m_Initial* ad ogni passo dell'algoritmo.

Implementazione delle configurazioni software

Un oggetto di tipo project contiene un'istanza della classe *instaces*. Quest'ultima sta al vertice della gerarchia di classi che definisce le configurazioni software. Un progetto infatti può contenere più configurazioni. La classe *instances* infatti dispone di una lista dinamica (m_configurations) che può contenere oggetti di tipo configuration. Per consentire la comunicazione tra le risorse[1] contenute, ogni configurazione dispone una serie di liste di variabili globali. L'oggetto m_globalVars è un istanza della classe VariablesLists che implementa una collezione di liste di variabili (oggetti di tipo VariablesList).

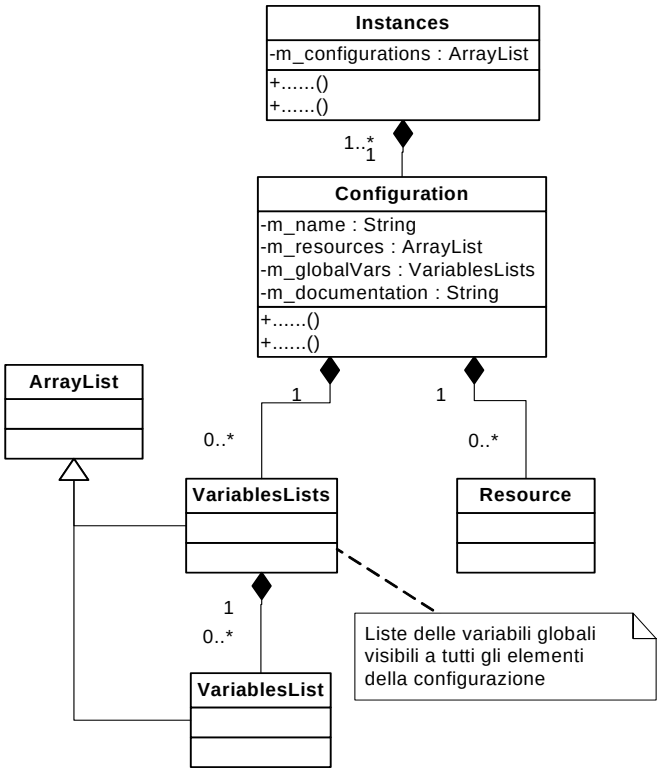


Figura 35. Class diagram 5

Le variabili

Allo stato attuale il tool consente l'utilizzo di variabili di tipo booleane. In visione di un successivo sviluppo degli altri linguaggi previsti dallo standard, che utilizzano anche variabili di tipo diverso, per implementare gli oggetti di tipo variabile è stata

[1] Nel modello software dello standard IEC 61131-3, una risorsa rappresenta un dispositivo in grado di eseguire programmi.

definita una classe di base astratta (*BaseVariable*). Ogni classe che implementa un tipo di variabile deve ereditare tale classe e ridefinirne in metodi virtuali. La classe *BaseVariable* contiene gli attributi comuni a tutti i tipi di variabile definiti dall'*XML format for IEC 61161-3*, nome (*m_name*), valore iniziale (*m_initialValue*), tipo di variable (*m_dataType)* ed indirizzo (*m_address)*. Quest'ultimo consente di effettuare il *mapping* della variabili sugli indirizzi. Inoltre la classe *BaseVariable* dispone di due eventi pubblici che vengono generati contestualmente al cambio del valore o del nome della variabile. Ciò consente agli oggetti che la utilizzano di intercettare questi eventi ed effettuare le opportune operazioni. Ad esempio gli oggetti che effettuano il monitoring grafico del valore della variabile possono aggiornare il relativo tracciato grafico del valore nel tempo. Nella figura seguente sono riportate la classe di base e la classe che implementa le variabili di tipo booleano con i relativi attributi e metodi.

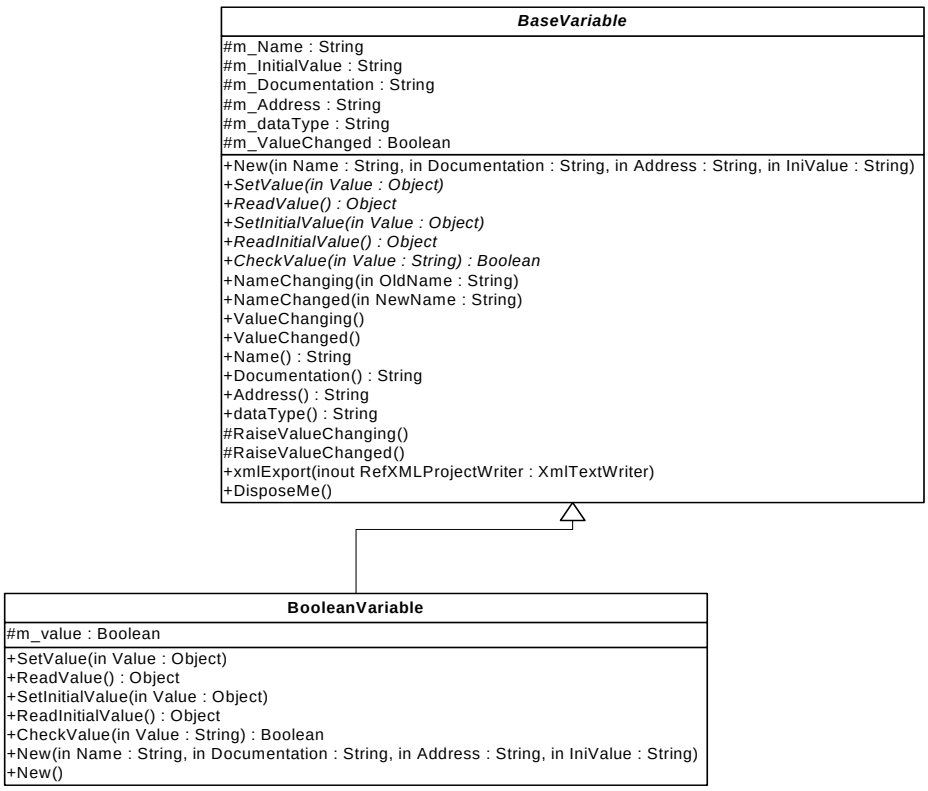


Figura 36. Class diagram 6

Una classe che eredita la classe *BaseVariable* deve effettuare l'override anche del metodo *CheckValue*. Questo metodo deve consentire di verificare se un valore assegnato alla variabile appartiene all'insieme dei valori che la variabile può assumere. Esso viene utilizzato per evitare assegnazioni di valori errati da parte

dell'utente.

Le risorse

Una risorsa corrisponde ad un'istanza della classe *resource*. Ogni istanza di questa classe deve avere un nome diverso da ogni altra contenuta nello stesso oggetto *configuration*. Un oggetto *resource* dispone di una collezione di liste di variabili globali, una lista di task ed una lista di istanze di POU prive di task. Le liste di variabili globali sono contenute, come nel caso di una configurazione, da un oggetto di tipo *VariablesLists*. Le definizioni dei task sono contenute in una lista dinamica (*m_tasks*) mentre le istanze delle unità organizzative di programma nella lista *m_pouInstances*, entrambe di tipo *ArrayList*.

L'identificazione stabilita dal modello software dello standard di una risorsa come dispositivo fisico in grado di eseguire programmi [2] è realizzata mediante le relazioni di contenimento tra un'istanza della classe *resource* e due istanze rispettivamente delle classi *SimulatorEngine* e *ControlEngine*. Queste ultime sono le classi delegate alla gestione del controllo della simulazione e delle funzioni di controllo del processo reale. Entrambe le classi saranno analizzate in dettaglio successivamente.

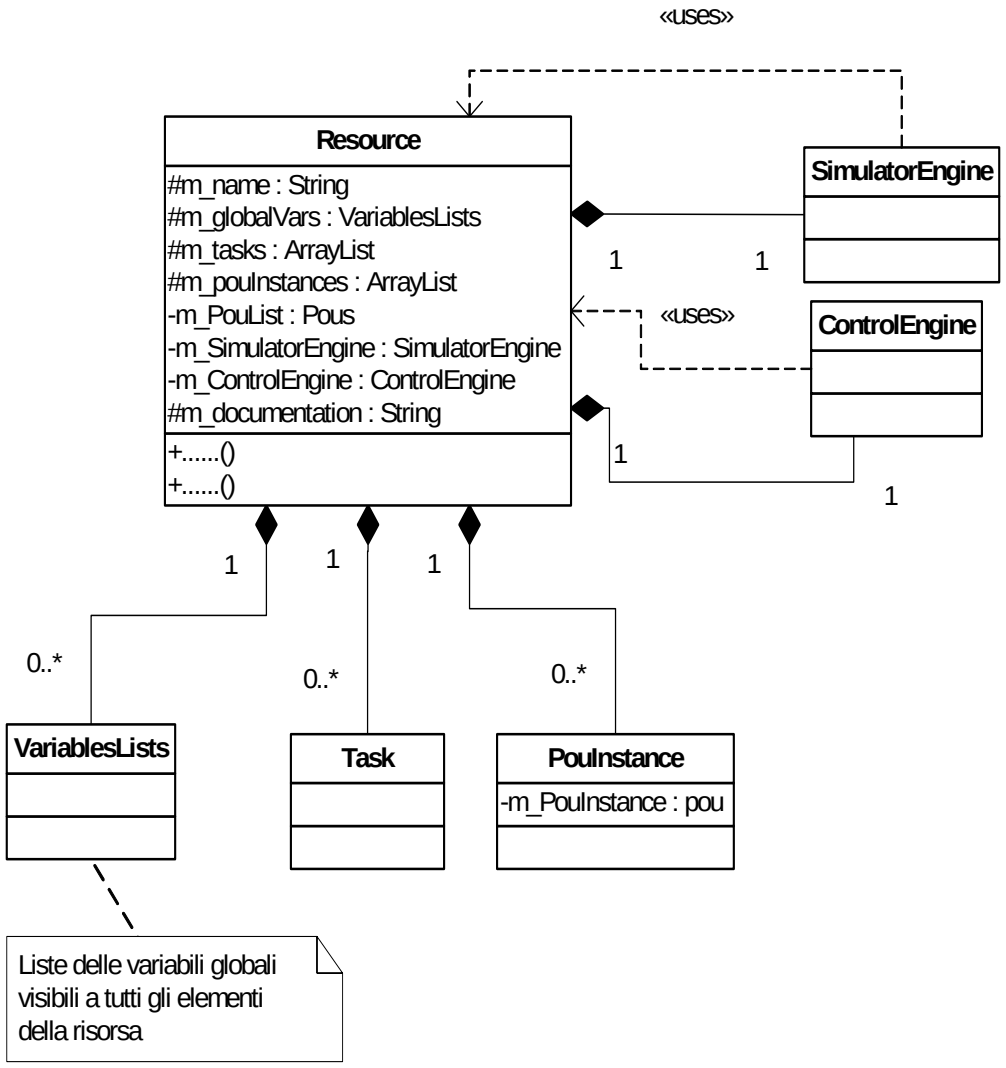


Figura 37. Class diagram 7

I task

La classe *task* implementa un compito della risorsa. Un oggetto *task* contiene l'attibuto *m_name* di tipo *string* che ne definisce il nome, l'attributo *m_inteval*, di tipo *TimeSpan*, che rappresenta l'intervallo ciclico di esecuzione del task, l'attributo *m_priority*, che ne determina la priorità, e l'attributo *m_single*, che rappresenta un collegamento con un evento il cui verificarsi coincide con una richiesta di esecuzione del task. L'attributo *m_priority* è di tipo *Uint*, cioè intero senza segno, e può assumere valori compresi tra 0 (priorità massima) e 65535 (priorità minima).
Inoltre dispone dell'attributo m_lastActivation, di tipo DateTime, atto a memorizzare l'istante di tempo in cui è iniziata dell'ultima esecuzione del task, e di una lista dinamica (*m_pouInstances*) che contiene le istanze delle POU del task.

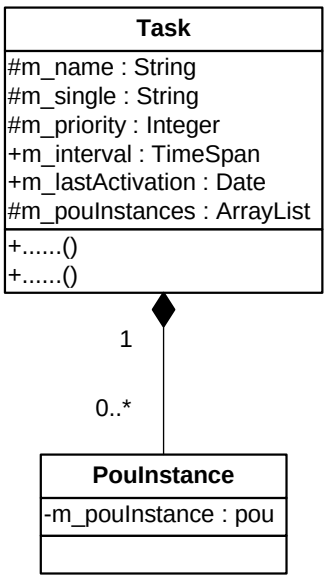


Figura 38. Class diagram 8

Le istanze delle unità organizzative di programma

Ogni configurazione software contiene le istanze delle unità organizzative di programma create dall'utente. Un oggetto *pouInstance* dispone di due riferimenti (*m_pou* ed *m_pouInstance*) ad oggetti della classe *pou*. Entrambi referenziano lo stesso programma, con la differenza che mentre *m_pou* referenzia direttamente l'oggetto *pou* in memoria creato dall'utente, *m_pouIstance* referenzia una copia

(dunque un'istanza) in memoria dello stesso oggetto. Lo scopo di ciò è dovuto ai diversi tipi di oggetti coinvolti nella fase di simulazione e di controllo del processo reale. La simulazione infatti viene effettuata eseguendo direttamente i programmi creati dall'utente. Questa scelta progettuale consente di effettuare la validazione dei programmi, testandone il comportamento, già in fase di editing. Ciò inoltre permette di apportare modifiche ad essi anche in corso di simulazione. Il controllo del processo reale invece viene effettuato eseguendo le istanze dei programmi. Prima dell'avvio della relativa esecuzione il metodo *CreateIstance()* della classe *PouInstance* crea una copia in memoria del programma da eseguire (ne crea un'istanza), con la collaborazione dei metodi omonimi di cui dispongono tutti gli oggetti che lo implementano. Ognuno di questi metodi crea in memoria una copia dell'oggetto a cui appartiene, chiamandone il relativo costruttore con i dovuti parametri, quindi restituisce un riferimento alla copia realizzata. Attraverso una serie di chiamate innestate vengono generate le copie di tutti gli oggetti di una POU.

Le funzioni di import/export del progetto

L'import/export dei progetti rispetta il formato definito *dall'XML format for IEC 61131-3* illustrato nel capitolo 2. La classe *XMLProjectManager* gestisce sia l'importazione che l'esportazione e dispone di un XML Validating Parser che effettua la validazione all'atto dell'importazione dei progetti.

Il metodo *xmlImport* riceve in input una stringa che contiene il testo in xml di un progetto letto da un file, il percorso del file contenente lo schema ed un riferimento ad un oggetto *project* in cui caricare il progetto. La stringa contenente il progetto ed il percorso dello schema vengono passati al parser (oggetto *m_XmlValidatingReader*, istanza della classe *XmlValidatingReader* [11,12]). Con il metodo *StartValidate()* inizia la validazione. Se va a buon fine attraverso una serie di chiamate ai metodi *xmlImport* degli oggetti corrispondenti agli elementi presenti nel file xml viene creato il progetto. Precisamente ogni oggetto crea gli oggetti di livello gerarchico inferiore e successivamente ne chiama il relativo metodo *xmlImport*, passandogli come parametro un riferimento all'oggetto *m_XmlTextReader*. Questo oggetto contiene il testo del file valido e fornisce i

metodi per leggerne il contenuto. Ogni metodo *xmlImport*, leggendo il contenuto degli elementi, fa avanzare la posizione di lettura del suddetto oggetto. L'importazione prosegue sino al termine degli elementi contenuti nel file.

L'esportazione del progetto avviene seguendo lo stesso metodo. La classe *XMLProjectManager* dispone dell'oggetto *m_XmlTextWriter* che viene passato ai metodi *xmlExport* degli oggetti che compongono il progetto. Ognuno di essi scrive le relative informazioni in un buffer di questo oggetto. Terminata la scrittura viene creato il file xml contente il progetto.

Capitolo 4

Schermate grafiche

In questo capitolo verranno illustrate alcune schermate grafiche significative del tool in funzione, per rappresentarne simbolicamente il funzionamento.

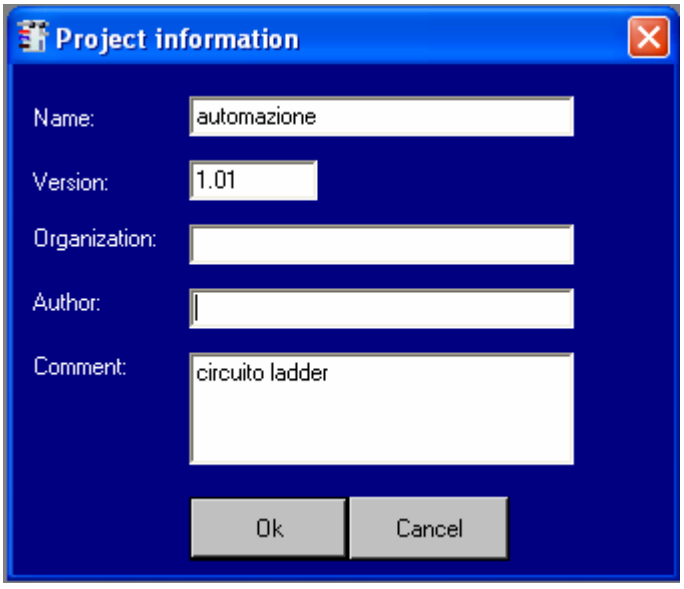


Figura 39. Creazione di un nuovo progetto

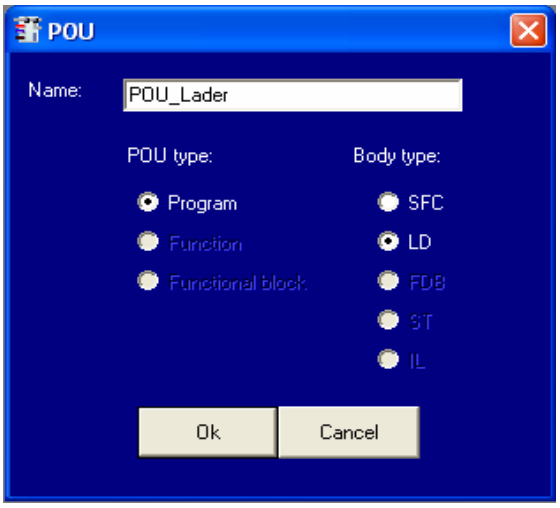


Figura 40. Creazione di una POU tipo LADDER

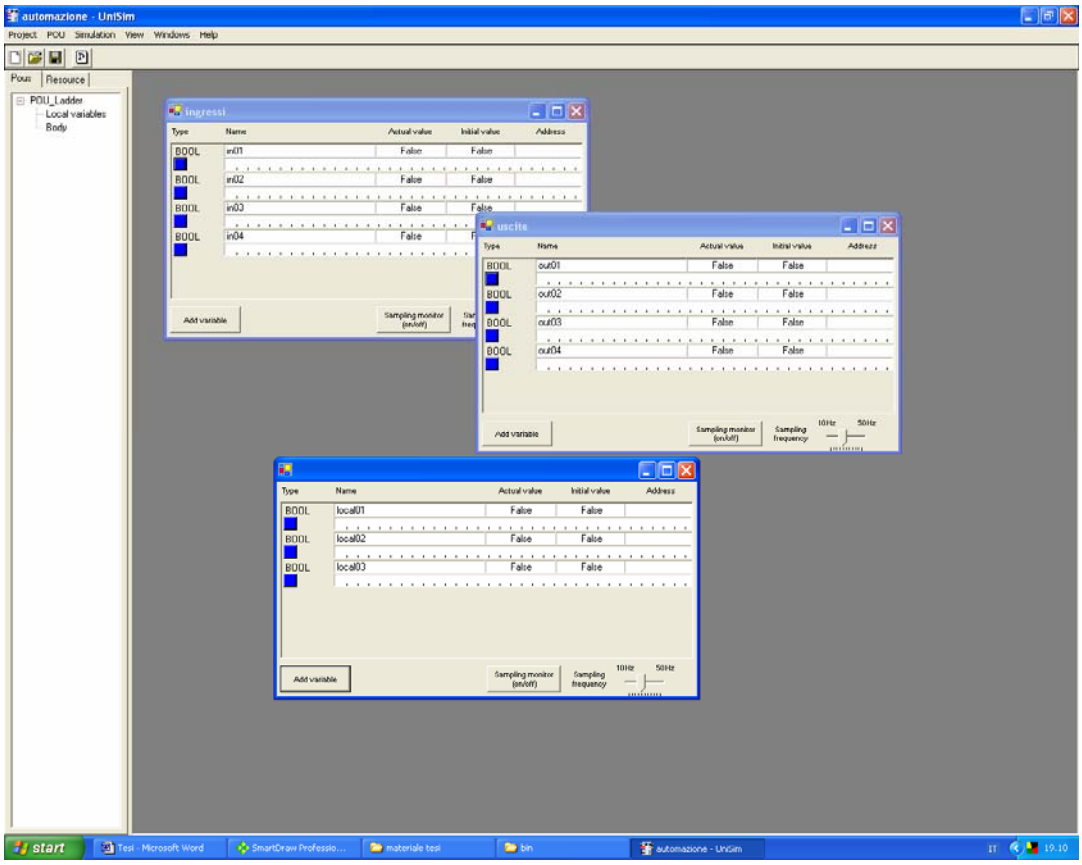


Figura 41. Definizione di variabili (globali e locali)

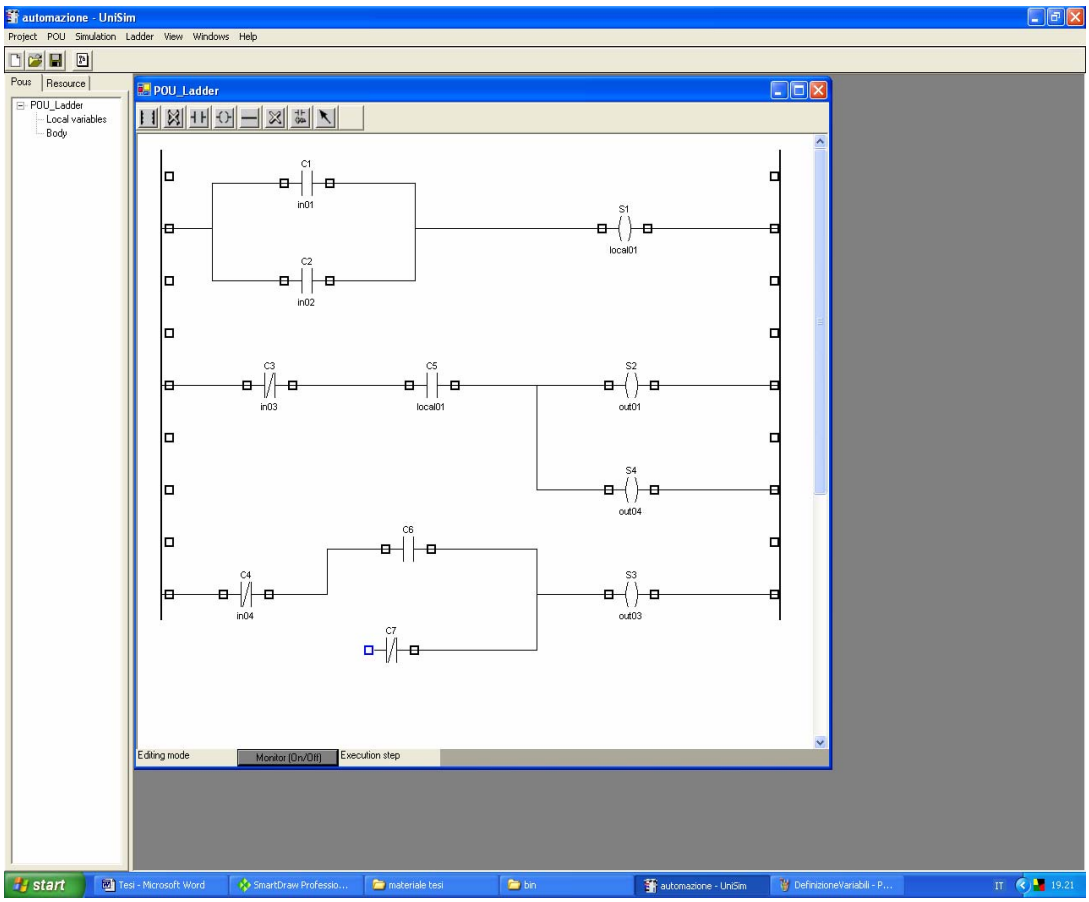


Figura 42. Creazione del circuito

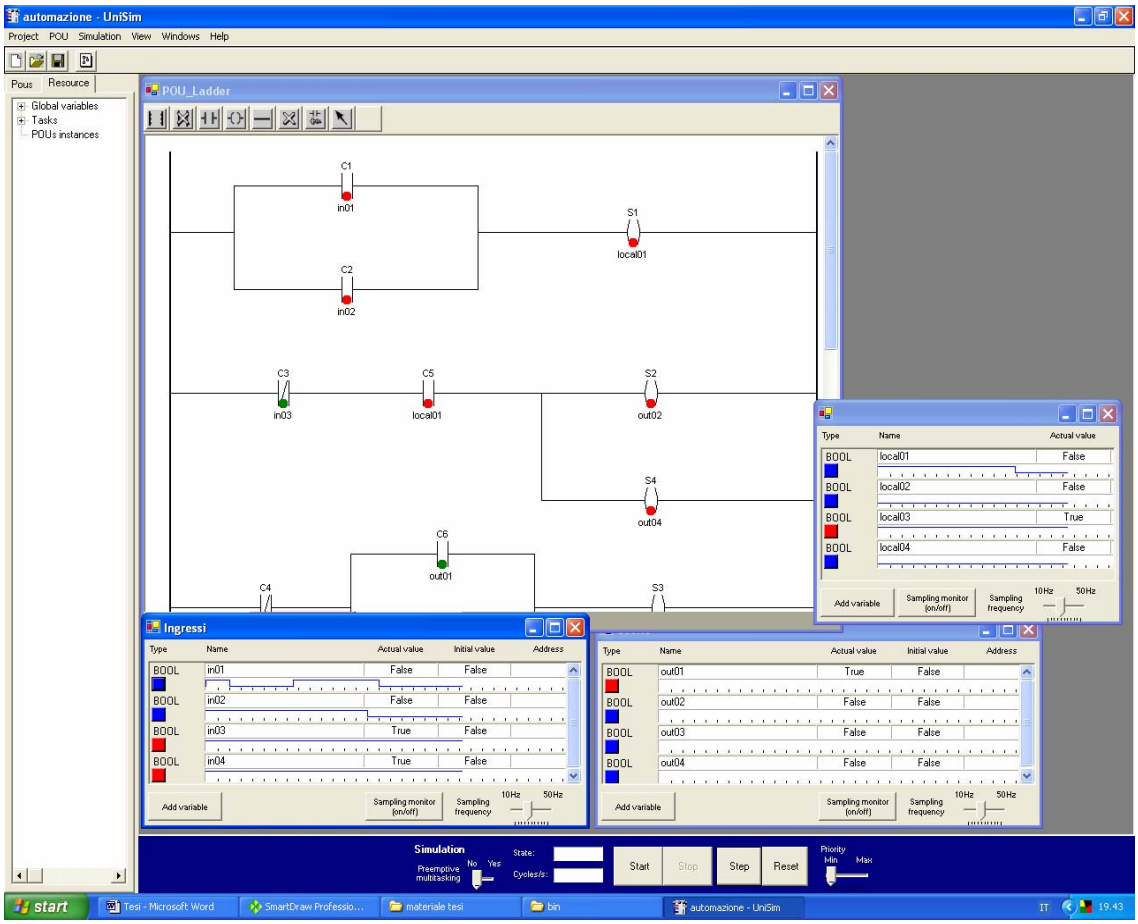


Figura 43. Simulazione del circuito

Conclusioni

UniSim è un tool di sviluppo per la creazione di progetti d'automazione secondo le direttive dettate dalla norma IEC 61131-3. La sue caratteristiche principali sono:

- la presenza di un editor grafico per lo sviluppo dei progetti;
- la presenza di un simulatore di controllo;
- la presenza di un modulo per il controllo dei processi reali;
- l'adozione dell'XML Formats for IEC 61131-3 per l'interscambio dei dati relativi ai progetti.

Attraverso questi strumenti consente di sviluppare i progetti, di validarne il contenuto e di effettuarne una prototipazione rapida. Inoltre l'adozione del formato XML consente di riutilizzare i progetti sviluppati e validati con UniSim su piattaforme commerciali.

La versione attuale di UniSim presenta tuttavia alcune limitazioni, tra le quali, le più importanti sicuramente sono:

- l'implementazione solo parziale dei linguaggi e dei tipi di dato previsti dalla norma IEC 61131-3;
- la possibilità di gestire un solo dispositivo (risorsa) all'interno del progetto d'automazione

La scelta di un approccio object-oriented per la progettazione dell''architettura, ha facilitato l'introduzione dei moduli aggiuntivi.

Bibliografia

[1] P. Chiacchio, 1996, "PLC e automazione industriale", McGraw Hill

[2] P. Chiacchio, R.Basile, 2004, "Tecnologie informatiche per l'automazione" McGraw Hill

[3] David Gulbransen, "XML Schema", McGraw-Hill

[4] PLCOpen, 2005, "XML Formats for IEC 61131-3 v1.0 – Official Release"

[5] "IEC 61131-3 International Standard 2nd Edition", 2003, IEC publishing

[6] http://plcopen.org/pages/fr_tc6.htm

[7] http://www.w3.org/XML/

[8] http://www.w3.org/XML/Schema

[9] http://www.w3.org/

[10] http://msdn.microsoft.com/library/

[11] Microsoft, 2003. "Manuale dello sviluppatore di .NET Framework"

[12] Microsoft, 2003. "Documentazione di .NET Framework"

[13] http://www.plcopen.org