Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

tesi di laurea

Un tool di sviluppo, validazione e controllo per progetti di automazione

Anno Accademico 2004-05

relatore
Ch.mo prof. Alfredo Pironti

correlatore
Ing. Gianmaria De Tommasi

candidato
Pierangelo Di Sanzo
matr. 831/000127

Indice

Introduzione

SiValPro (**Si**mulation, **Va**lidation and rapid **Pro**totyping) è il tool realizzato in questo lavoro di tesi ed è stato progettato per lo sviluppo di software d'automazione. Gli strumenti da esso forniti consentono 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. Il tool è stato sviluppato con l'impiego di tecniche orientate agli oggetti e consente di realizzare i progetti rispettando le specifiche dello standard IEC 61131-3 [5]. Allo stato attuale le differenti piattaforme che adottano tale standard non forniscono la possibilità di importare o esportare progetti, programmi o librerie attraverso un formato comune. SiValPro utilizza il nuovo *XML Format for IEC 61131-3* [4,8], rilasciato dall'organizzazione internazionale *vendor* e *product-indipendent* PLCOpen [24] . Il formato costituisce un'interfaccia aperta per lo scambio di software progettato secondo il suddetto standard ed è definito mediante la tecnologia degli XML Schemas [3,11].

Allo stato attuale il tool consente lo sviluppo dei programmi in linguaggio SFC [5,6] ed è stato progettato in visione di un futuro sviluppo degli altri linguaggi previsti dallo Standard. L'architettura infatti rispecchia il modello software dello standard IEC, ripreso dal relativo formato di PLCOpen, ed è indipendente da uno specifico linguaggio. Un motore di simulazione consente di validare il software prodotto o importato con l'ausilio di un interfaccia grafica che permette di monitorare l'evoluzione dello stato dei programmi, del valore temporale delle variabili, dell'esecuzione dei task e delle istanze. E' possibile inoltre optare per un multitasking di tipo preemptive o non-preemptive [7,19]. Anche in fase di simulazione, grazie alle tecniche utilizzate per la sincronizzazione dei threads [18,19], sono consentite la modifica e la creazione dei programmi e delle variabili,

la creazione e l'eliminazione dei task, la modifica delle relative priorità ed intervalli ciclici, ed altre operazioni, senza dover interrompere l'esecuzione. In corso di simulazione è prevista anche la possibilità di passare dalla modalità non-preemptive a quella preemptive e viceversa.

Un modulo di controllo, fornito di una propria interfaccia utente, importa i progetti dal formato xml adottato dal tool e, interfacciandosi con dispositivi di I/O di tipo digitale, consente il controllo di processi reali non critici. Come in fase di simulazione, è possibile controllare l'evoluzione dello stato degli oggetti. Ciò è particolarmente utile ai fini della la validazione del software prodotto.

Il primo ed il secondo capitolo della tesi forniscono rispettivamente un'introduzione allo standard IEC 61131-3 ed un'analisi dell'XML Format for IEC 61131-3. Il capitolo terzo e quarto contengono la descrizione dettagliata dell'architettura, delle caratteristiche del tool e delle relative scelte progettuali. Infine viene effettuata una valutazione dei possibili sviluppi futuri.

Capitolo 1
Lo standard IEC 61131-3

La programmazione tradizionale dei controllori a logica programmabile [1] ha sempre sofferto di forti limitazioni: poca riusabilità del software, linguaggi non sufficientemente conformi ad uno standard, limiti nella definizione di strutture dati complesse, utilizzo limitato della modularità, ecc. Il tentativo di superare tali difficoltà è stato intrapreso dalla Commissione Elettrotecnica Internazionale (IEC [25]) con l'introduzione dello standard denominato IEC 61131–3 [2,5]. Le specifiche coprono dalla definizione dei tipi di variabili a quella di un intera configurazione di un sistema software. Vengono standardizzati cinque linguaggi di programmazione (Function Block Diagram, Ladder, Structured Text, Instruction List e Sequential Functional Chart), ognuno dei quali presenta particolari caratteristiche che lo rendono più o meno adatto all'implementazione di alcuni tipi di algoritmi piuttosto che ad altri.

Allo stato attuale l'IEC 61131-3 rappresenta lo standard più diffuso nell'ambito dei controllori a logica programmabile. Il motivo principale potrebbe essere sicuramente individuato nella relativa definizione di in un modello software efficiente e versatile.

Questo capitolo vuol essere una breve introduzione al suddetto modello, adottato dal tool sviluppato in questo lavoro di tesi, presentandone la struttura generale ed evitando di entrare nei dettagli implementativi dei linguaggi di programmazione.

1.1 Il modello software dello standard IEC 61131-3

L'approccio che si seguirà nel descrivere il modello software definito dallo standard IEC 61131-3 sarà di tipo top-down.

Struttura del modello software

Il modello definito ha una struttura gerarchica. Al vertice vi è una 'configurazione' (*configuration*): rappresenta l'insieme di tutto il software utilizzato da tutti i dispositivi che realizzano il controllo di un processo. Il software infatti non rappresenta solo i programmi, ma definisce anche la configurazione di tutti i dispositivi di controllo. Un dispositivo fisico è identificato come una 'risorsa' (*resource*). Più precisamente una risorsa rappresenta un entità capace di eseguire programmi[2]. La comunicazione tra le risorse di una configurazione viene realizzata mediante le variabili globali definite a livello di configurazione.

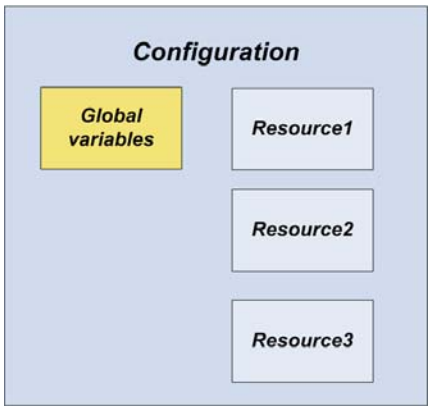


Figura 1. Configurazione contenente variabili globali e risorse

Un PLC monoprocessore non interfacciato con altri PLC potrebbe essere associato ad una configurazione contenente un'unica risorsa; un PLC multiprocessore ad una configurazione contenente tante risorse quanti sono i processori; ugualmente una serie di PLC monoprocessori collegati da un bus di campo possono essere associati ognuno ad una risorsa contenuta in un unica configurazione.

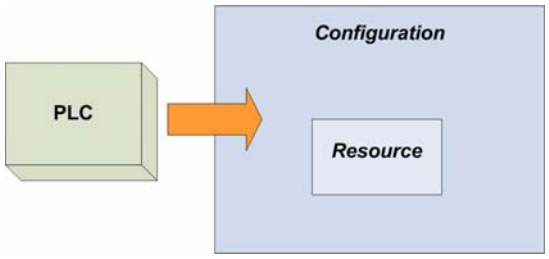


Figura 2. Associazione tra un PLC ed una configurazione contenente un'unica risorsa

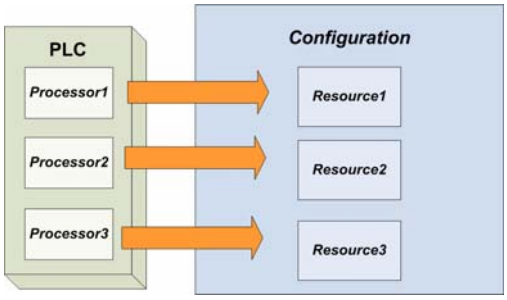


Figura 3. Associazione tra un PLC multiprocessore ed una configurazione con più risorse

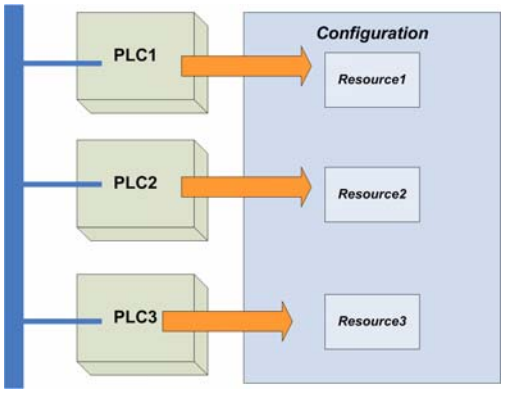


Figura 4. Associazione tra più PLC monoprocessore collegati ad un bus ed una configurazione contene più risorse

Una risorsa contiene compiti (*task*), variabili globali per la comunicazione interna ed istanze di unità organizzative di programma (*Program Organizzation Unit*, POU) prive di task.

Un task determina l'esecuzione delle istanze delle unità organizzative di

programma che contiene. L'esecuzione può essere di tipo periodica o legata al verificarsi di un evento. Il periodo di esecuzione per i task di tipo ciclico è definito dal relativo attributo *Interval*. Il verificarsi di un evento associato all'esecuzione dei task non ciclici è identificato dal fronte di salita di una variabile booleana associata al task, identificata come *Single*. Un task inoltre dispone di un valore che ne definisce la priorità rispetto agli altri task (*priority*). Esso può variare da 0 (priorità massima) a 65535 (priorità minima). Le istanze delle POU che non dispongono di un task sono considerate a priorità minima e con intervallo di ciclo nullo. Ciò ne implica l'esecuzione solo in caso in cui non vi siano task pronti ad essere eseguiti

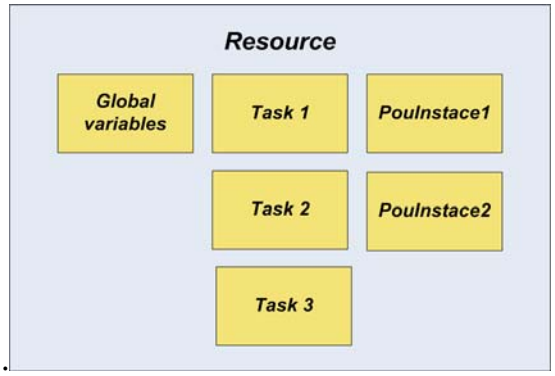


Figura 5. Risorsa contenente variabili globali, task e istanze di POU

Le istanze delle unità organizzative di programma si riferiscono a programmi o blocchi funzionali definiti attraverso i linguaggi di programmazione previsti dallo standard.

La possibilità di creare delle istanze consente il riutilizzo del codice scritto. Infatti un programma o un blocco funzionale possono essere utilizzati più volte all'interno di una configurazione. Ciò può essere particolarmente utile per controllare più entità dello stesso tipo o compiere operazioni uguali richieste in punti diversi di un programma. Un esempio potrebbe essere il caso di una serie di turbine che richiedono lo stesso algoritmo di controllo.

Le POU in realtà poso essere di tre tipi: programmi (*program*), blocchi funzionali (*functional block*) e funzioni (*function)*. La differenza tra un blocco funzionale ed una funzione risiede nel fatto che il primo può tenere memoria del proprio stato tra due cicli di scansione successivi, mentre ciò non può essere effettuato da una funzione. Un'importante differenza dei programmi rispetto ai blocchi funzionali

ed alle funzioni è la possibilità dei primi di poter utilizzare diversi tipi di variabile:

- variabili indirizzate direttamente verso locazioni memoria;
- variabili di accesso, cioè variabili accessibili anche dall'esterno da parte di altri programmi;
- variabili globali, cioè variabili accessibili dai blocchi funzionali interni al programma;
- variabili che rappresentano le locazioni di memoria su cui sono mappati gli ingressi e le uscite fisiche dei dispositivi.

Si riporta in seguito uno schema globale che illustra la definizione di una configurazione.

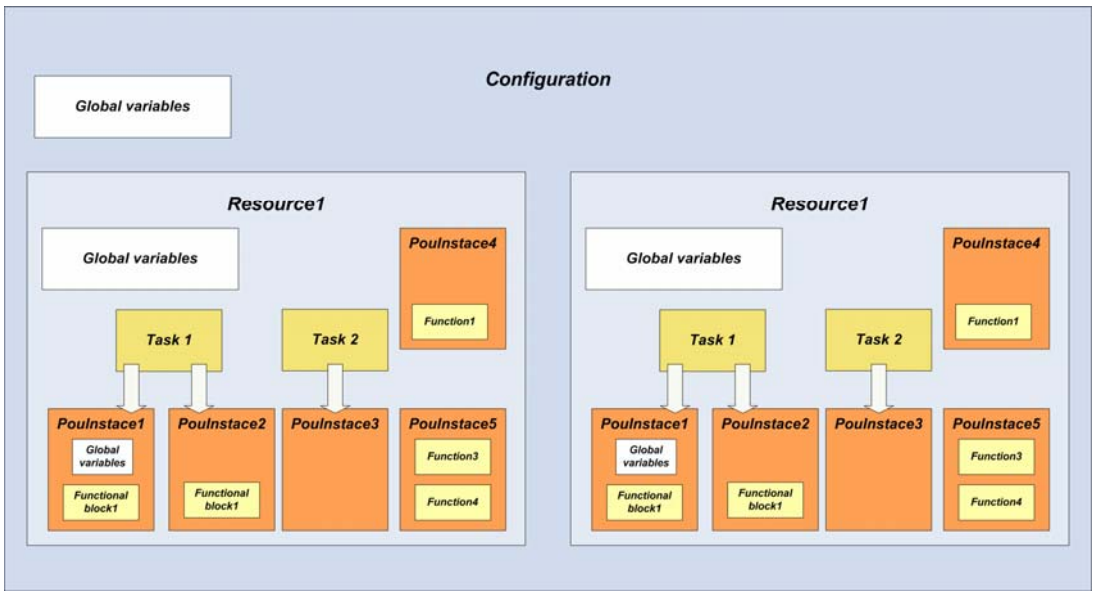


Figura 6. Esempio di configurazione

Caratteristiche generali dei linguaggi

I linguaggi definiti dallo Standard si rivelano più o meno adatti a seconda del tipo di algoritmo da implementare. Il Sequential Functional Chart (SFC) ad esempio si rivela particolarmente utile nel controllo di sistemi ad eventi discreti. E' un linguaggio di tipo grafico come il Function Block Diagram (FBD) ed il Ladder (LD). 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 (ST) richiama la programmazione procedurale dei linguaggi di alto livello come il Pascal. Infine il linguaggio Instruction List (IS) 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.

Capitolo 2
L'XML Format for IEC 61131-3

Lo standard IEC 61131-3 fornisce una definizione completa del modello software di un progetto di automazione. Non definisce però un'interfaccia per l'interscambio delle informazioni tra gli strumenti che lo adottano. L'organizzazione PLCopen[1][24] ha rilasciato il 28 aprile 2005 la prima versione ufficiale dell'XML Format for IEC 61131-3 [5]. Si tratta di un'interfaccia aperta per l'import/export di progetti, programmi e librerie sviluppati secondo il suddetto standard. Il formato utilizza il linguaggio non proprietario XML [10], definendo la struttura dei documenti attraverso un XML Schema [3,11]. Questa tecnologia permette la produzione di documenti la cui validità (conformità allo schema) può essere verificata attraverso l'utilizzo dei Validating XML Parsers [3,11], 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

[1] PLCOpen (http://www.plcopen.org) è un'organizzazione internazionale *vendor* e *product-indipendent* costituita dalle maggiori aziende ed istituti mondiali attivi nel campo dei sistemi di automazione industriale (es. Siemens, BeckOff, Rockweller Automation, Ifak, Thoshiba, Panasonic, TetraPack,…). PLCOpen è stata fondata nel 1992 con sede in Olanda con *"la missione di costituire l'associazione leader nella risoluzione delle questioni relative al control programming, per favorire l'utilizzo degli standard internazionali in questo campo"* (dal sito web di PLCOpen).

automazione.

Lo sviluppo di SiValPro ha avuto inizio quando il formato era ancora in fase di sperimentazione con la "version 0.99 – Release for Comment". I primi utilizzi di questa versione in fase di sviluppo del tool hanno rivelato la presenza di alcuni errori di coerenza con le regole per la definizione degli schemi xml. Essi sono stati prontamente segnalati agli autori. La prima versione ufficiale ne è risultata priva ed arricchita in molte sue parti.

In questo capitolo dopo una breve introduzione sul contesto di riferimento del nuovo formato e sulla tecnologia degli schemi xml, si analizzerà nel dettaglio lo schema definito dall'XML Format for IEC 61131-3.

2.1 Il contesto di riferimento dell' 'XML Format for IEC 61131-3'

"[..]. PLCopen creates a complete new market, in which the focus is on reusability of software development from libraries up to complete control projects.[..]" [4,pag.6]. Da questa citazione, tratta dal documento ufficiale delle specifiche dell'*XML Format for IEC 61131-,3* si evince l'intenzione di PLCopen di aver voluto creare un'interfaccia per lo scambio di informazioni con la prospettiva di affermarsi come standard nel mercato del software di automazione. In effetti si tratta di un'interfaccia aperta che si colloca tra tutta una serie di tool di sviluppo, di debugging, di simulazione, di validazione, di controllo, ecc. Nei seguenti schemi, tratti rispettivamente da sito web di PLCopen e dal documento ufficiale di specifiche, si evidenziano il contesto in cui si colloca il formato ed alcuni dei possibili casi d'uso.

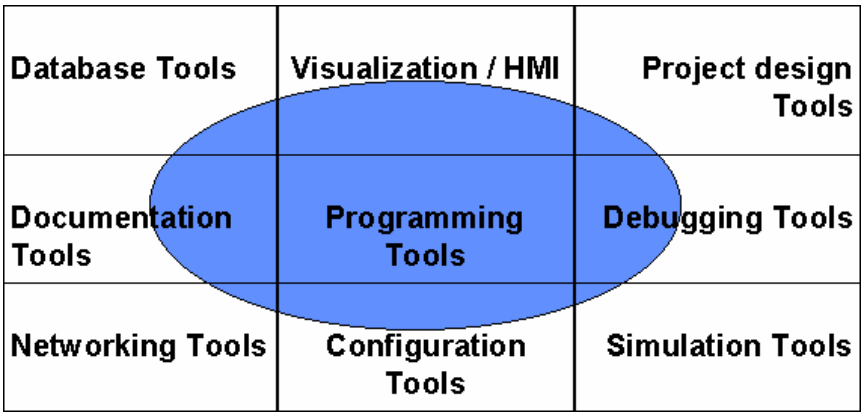


Figura 7. Contesto di inserimento dell' XML Formats for IEC 61131-3. (*Tratta dal sito web di PLCopen: www.plcopen.org)*

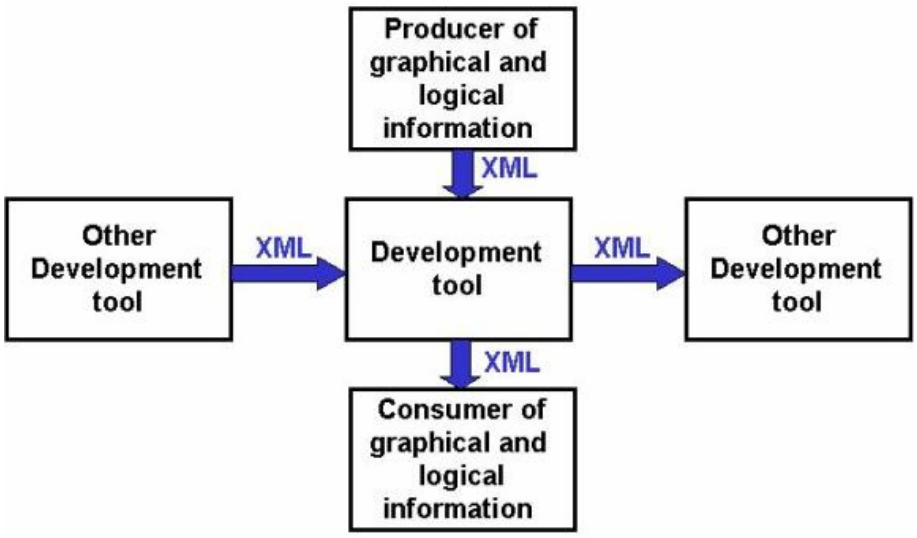


Figura 8. Possibili casi d'uso dell' XML Formats for IEC 61131-3. (*Tratta dal documento 'XML Formats for IEC 61131-3 Version 1.0 – Official Release')*

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.

2.2 L'XML e gli XML Schemas

XML (eXtensible Markup Language) è un meta-linguaggio per la definizione di linguaggi di mark-up. Le specifiche ufficiali sono state definite dal W3C (Worl Wide Web Consortium) [12]. Derivato dall'SGML[2] (*Standard Generalized Markup Language),* XML fornisce un insieme standard di regole sintattiche per modellare la struttura dei dati. E' un linguaggio *estensibile* in quanto a differenza di un linguaggio di mark-up con un insieme predefinito di tags fornisce la possibilità di definirne nuovi. Ciò permette a sua volta la definizione stessa di nuovi linguaggi di mark-up. Ad esempio HTML, esistente già prima di XML, può essere definito tramite XML stesso.

La struttura di un documento xml è di tipo gerarchico, rappresentabile attraverso un albero, il cosiddetto *document tree*, le cui foglie sono gli elementi del documento. Quest'ultimo viene definito *ben formato* (*well formed*) se rispetta le regole sintattiche definite dall'XML. Se un documento sia ben formato può essere verificato attraverso gli *xml parsers.*

```
<?xml version="1.0" encoding="UTF-8"?>
<biblioteca>
    <pubblicazione id="lt12g " tipo="libro">
        <titolo>Computer Networks</titolo>
        <autore>Andrew S. Tanenbaum</autore>
        <editore>Prentice Hall</editore>
        <anno>2003</anno>
    </pubblicazione>
    <pubblicazione id="bg42v" tipo="libro">
        <titolo> Computer Networking: a top-down approach featuring the Internet </titolo>
        <autore>James F. Kurose</autore>
        <autore>Keith W.Ross</autore>
        <editore>McGraw-Hill</editore>
        <anno>2001</anno>
```

[2] SGML è definito dallo standard ISO 8879:1986 *Information processing---Text and office systems---Standard Generalized Markup Language*

```
        </pubblicazione>
</biblioteca>
```

Figura 9. Esempio di documento xml

Nella figura precedente è riportato un esempio di un documento xml che contiene informazioni relative alle pubblicazioni di una biblioteca. *biblioteca* è l'elemento root che a sua volta contiene due elementi *pubblicazione*. Ognuno di questi contiene due attributi relativi all'identificativo (*id*) e al tipo, e gli elementi relativi al titolo, agli autori, all'editore e all'anno di pubblicazione.

Le regole dell'XML consentono di definire la struttura dei dati, ma non forniscono gli elementi per definire i tipi di dati o il modello di struttura di un documento. Ad esempio, per il documento in figura precedente nulla stabilisce che ad un elemento *pubblicazione*, oltre agli elementi titolo, autore, editore ed anno, non si possa aggiungere un elemento che non abbia alcun collegamento semantico con la pubblicazione (ad esempio l'elemento *tavolo)*. Ugualmente non esiste alcuna regola che vieti di inserire in una pubblicazione due volte l'elemento *anno*. In questi casi sarebbe compito delle applicazioni evitare di generare tali contraddizioni.

In supporto di ciò vi sono altre tecnologie. Tra queste vi è quella degli XML Schemas [3,11].

Un XML Schema è un documento definito attraverso l'XML stesso. Tra le diverse tecnologie per la definizione dei documenti xml è probabilmente la più complessa, ma anche quella che dispone del maggior potere espressivo. Un XML Schema quindi è un documento xml redatto secondo determinate specifiche che contiene la definizione formale degli elementi e della struttura di un altro documento xml. Quest'ultimo se rispetta le regole definite dallo schema di riferimento viene definito *valido*. In uno schema xml una serie di costrutti, espressi sempre sotto forma di tags, permettono ad esempio di definire la sequenza con cui devono susseguirsi gli elementi in un documento, il relativo numero di occorrenze, i possibili valori che può assumere un attributo, ecc.

In un documento valido non è possibile utilizzare elementi, attributi o strutture se non definite dal proprio schema. Uno schema per il documento precedente potrebbe essere il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="biblioteca">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="pubblicazione" minOccurs="0" maxOccurs="unbounded">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="titolo"/>
                            <xsd:sequence maxOccurs="unbounded">
                              <xsd:element name="autore"/>
                            </xsd:sequence>
                            <xsd:element name="editore"/>
                            <xsd:element name="anno"/>
                        </xsd:sequence>
                        <xsd:attribute name="id" type="xsd:string" use="required"/>
                        <xsd:attribute name="tipo" type="xsd:string" use="required"/>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Questo schema impone che il root-element sia necessariamente *biblioteca*. Esso contiene un *tipo complesso* (*complexType*)costituito da una *sequenza* (*sequence*) di "zero" o più elementi *pubblicazione* (attibuti *minOccurs="0", maxOccurs= "unbounded"*). Ogni elemento *pubblicazione* deve avere necessariamente i due attributi *id* e *tipo* (*use="required")* e deve contenere in sequenza un elemento *titolo*, uno o più elementi *autore*, l'elemento *editore* e l'elemento *anno*.

Uno schema xml contiene riferimenti a spazi dei nomi (*namespace*) esterni, cioè a documenti che contengono ulteriori definizioni di elementi che possono essere utilizzati dallo schema, evitando di doverli ridefinire nello schema stesso. Nello schema precedente l'attributo di *schema 'xmlns:xsd="http://www.w3.org/ 2001/XMLSchema'* fa riferimento allo spazio dei nomi standard degli schemi xml. Ad esso appartengono i nomi degli elementi di base per la definizione degli schemi. Il prefisso posposto a *xmlns* (in questo caso *xsd*) occorre nel seguito del documento per dichiarare che un elemento appartiene a quello spazio dei nomi.

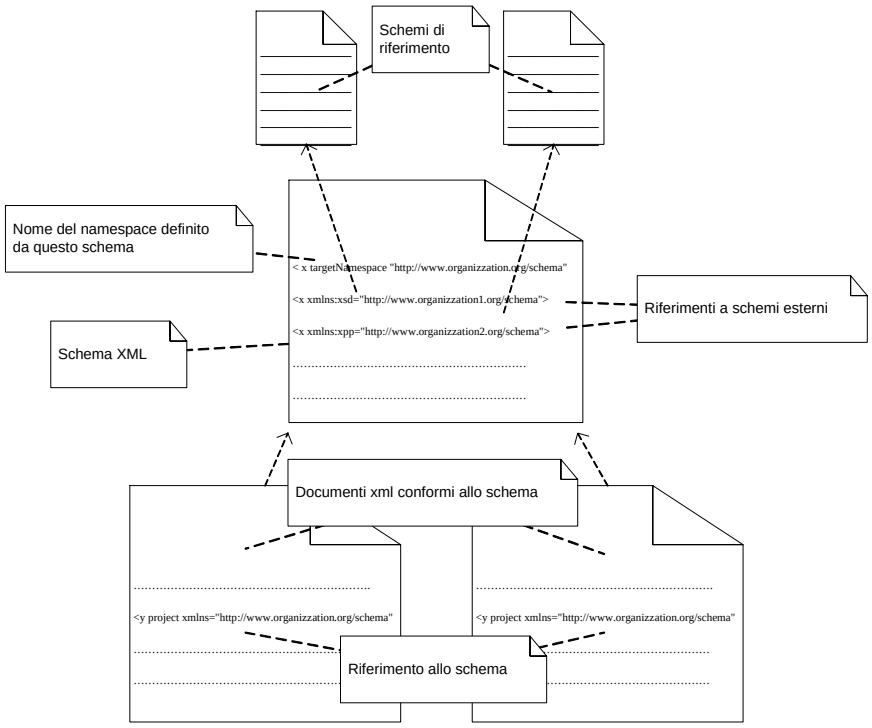


Figura 10. Utilizzo di spazi dei nomi esterni per la definizione di uno schema XML: i due documenti in basso sono definiti secondo lo schema centrale. Quest'ultimo utilizza gli elementi definiti nei due schemi in alto.

La validità di un documento xml redatto secondo uno schema di riferimento può essere verificata attraverso i *validating xml parsers*. Questi strumenti ricevono in input il documento ed il relativo schema e, oltre a controllare se il documento risulta ben formato, sono in grado di verificare se rispetta tutte le regole definite dallo schema, fornendo eventualmente la lista delle inconformità.

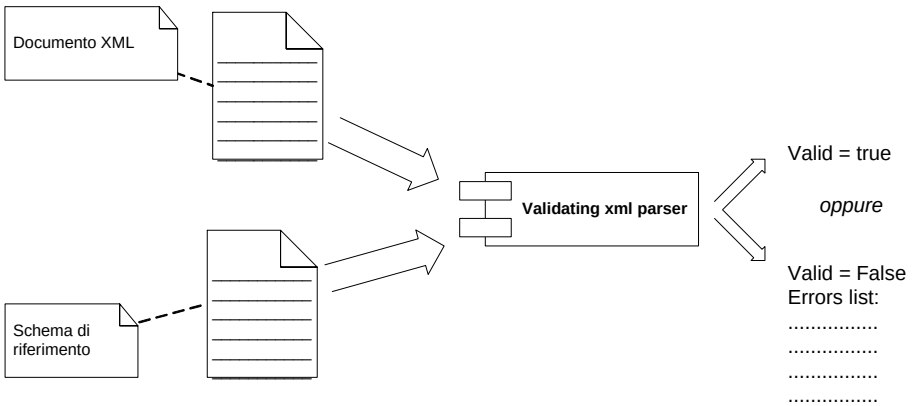


Figura 11. Verifica della validità di un documento xml

Un'applicazione che utilizza documenti xml definiti secondo uno schema può eventualmente includere un parser di questo tipo per verificarne immediatamente la validità.

2.3 Lo *schema dell'* 'XML Format for IEC 61131-3'

Le specifiche dell'*XML Format for IEC 61131-3* forniscono lo schema che definisce la struttura dei documenti che rispettano questo formato. Lo schema è lo stesso per qualunque tipo di informazione relativa ad un progetto di automazione, che sia una libreria di programmi, di funzioni o blocchi funzionali, una definizione di una risorsa, una configurazione di più risorse o un intero progetto. E' compito del sistema che importa i dati eventualmente selezionare le parti che interessano. Lo spazio dei nomi definito dallo schema ha come URI[3] (*Universal Resource Identifier*) *http://www.plcopen.org/xml/tc6.xsd* e viene referenziato all'interno dello stesso schema con il prefisso *ppx*. Gli spazi dei nomi esterni cui fa riferimento sono lo spazio dei nomi standard *http://www.w3.org/2001/XMLSchema*, referenziato con il prefisso *xsd*, e lo spazio *http://www.w3.org/1999/xhtml*, con prefisso `xhtml`.
Nell'elemento root dello schema (x*sd:schema)* sono definiti l'elemento *project*, che a sua volta contiene la definizione dell'intero progetto, i tipi di dati e i gruppi di oggetti utilizzati.
Lo schema stabilisce che un progetto debba essere composto dai quattro elementi in sequenza: *fileHeader*, *contentHeader*, *types* e *instances*.

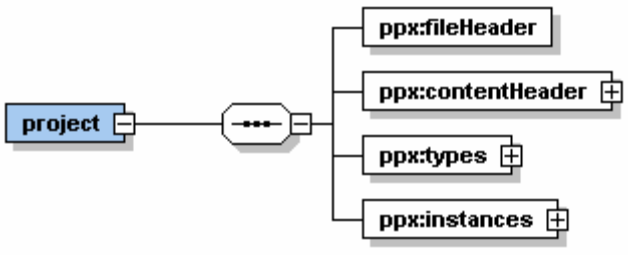


Figura 12. Struttura dell'elemento *project*

[3] Uno spazio dei nomi è identificato da un URI (*Universal Resource Identifier*). Il fatto che l'URI assuma lo stesso aspetto dell'URL di una risorsa non vuol dire che nella locazione individuata dall'URI debba esserci necessariamente un server con un oggetto pubblicato. In realtà costituisce un identificativo simbolico per lo spazio dei nomi (si veda David Gulbransen , 'XML Schema', Ed. McGraw-Hill)

I primi due elementi contengono informazioni generali relative al progetto ed al file xml di import/export che lo contiene. L'elemento types *contiene* le definizioni delle *unità organizzative di programma* (*POU – Program Organization Units,* ossia i programmi, le funzioni e i blocchi funzionali) e dei tipi di dati utilizzati. L'elemento *instances* contiene la definizione delle configurazioni con le relative risorse e variabili globali. All'interno di questo elemento vengono definite le istanze delle POUS. Grazie a questa netta separazione tra tipi e istanze, l'XML Format for IEC 61131-3 permette l'import/export di librerie di programmi, funzioni e blocchi funzionali indipendentemente dai progetti e dalla configurazione dell'ambiente nel quale devono essere eseguiti, promuovendo in questo modo l'aspetto fondamentale del riutilizzo del codice.

Successivamente sarà analizzata la struttura dei quattro elementi principali di un progetto. In molti casi gli elementi contengono l'elemento opzionale *documentation*. Quest'ultimo contiene la documentazione relativa al contenuto dell'elemento di cui fa parte ed è composto da un elemento di tipo *formattedText*. Un elemento di questo tipo è un qualunque elemento definito dallo spazio dei nomi '*http://www.w3.org/1999/xhtml'*, cioè contenuto nelle specifiche ufficiali di XHTML 1.1

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*. *FileHeader* 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.
La struttura dettagliata di *conterHeader* è riportata in figura.

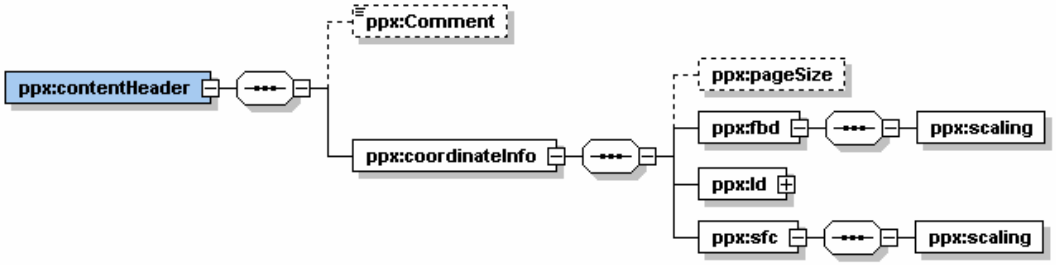


Figura 13. Struttura dell'elemento *conterHeader*

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*).

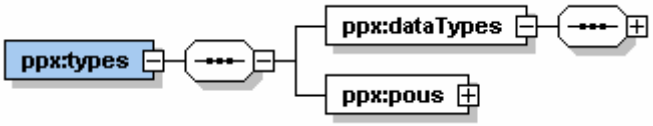


Figura 14. Struttura dell'elemento *types*

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.

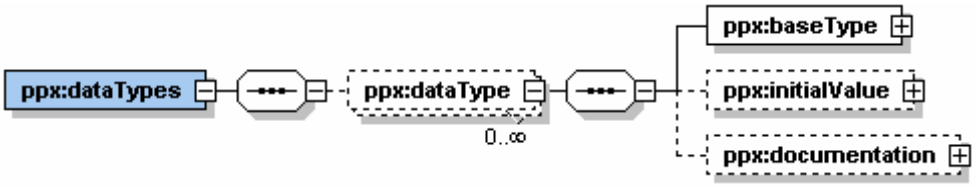


Figura 15. Struttura dell'elemento datatypes

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.

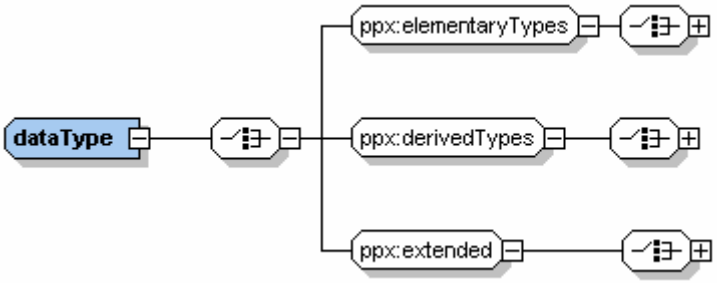


Figura 16. Struttura dell'elemento *dataType*

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*

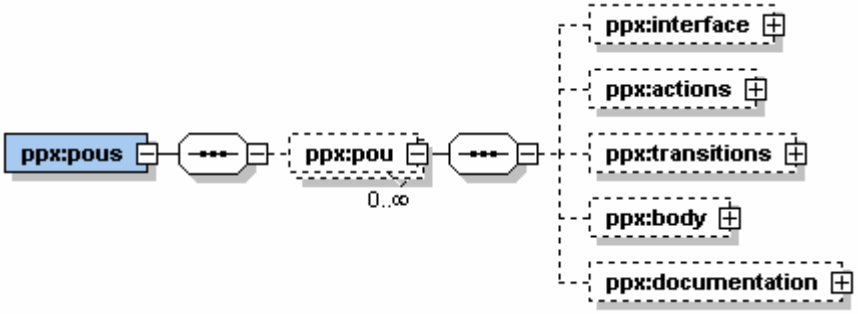


Figura 17. Struttura dell'elemento *POUS*

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

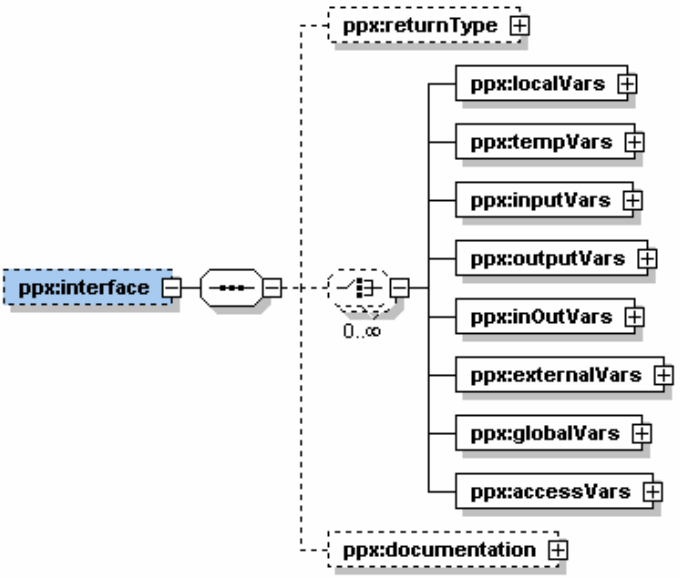


Figura 18. Struttura dell'elemento *interface* della POU

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.

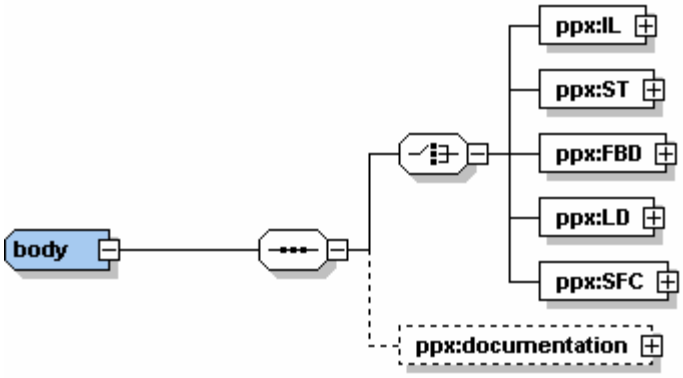


Figura 19. Struttura dell'elemento *body*

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.

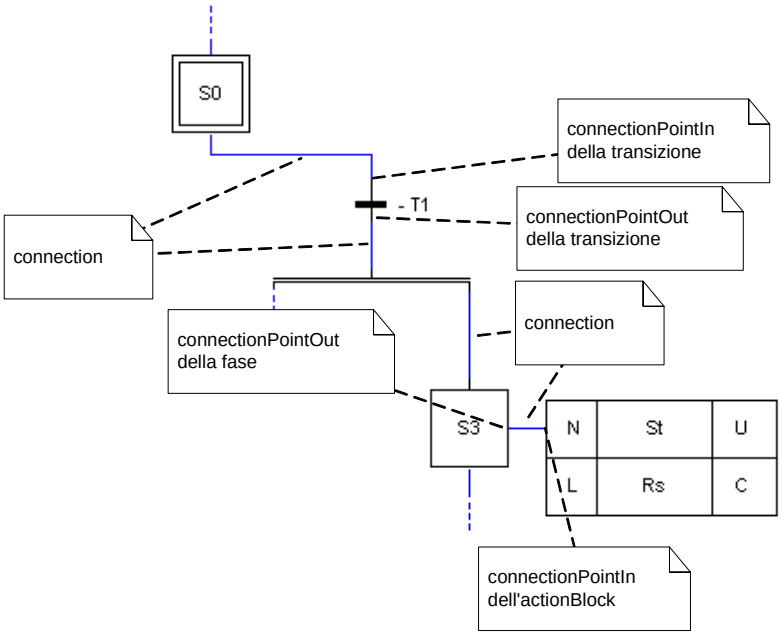


Figura 20. Esempio di elementi connection, conntctionPointIn e connectionPointOut nell'SFC

Il gruppo *commonObject* contiene gli elementi che definiscono gli oggetti comuni ai tre linguaggi grafici.

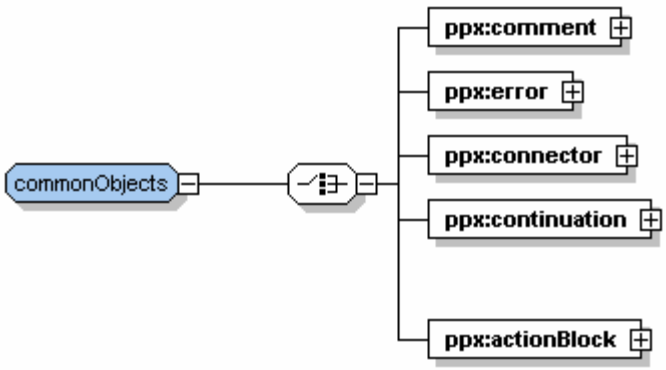


Figura 21. Il gruppo *commonObjects*

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

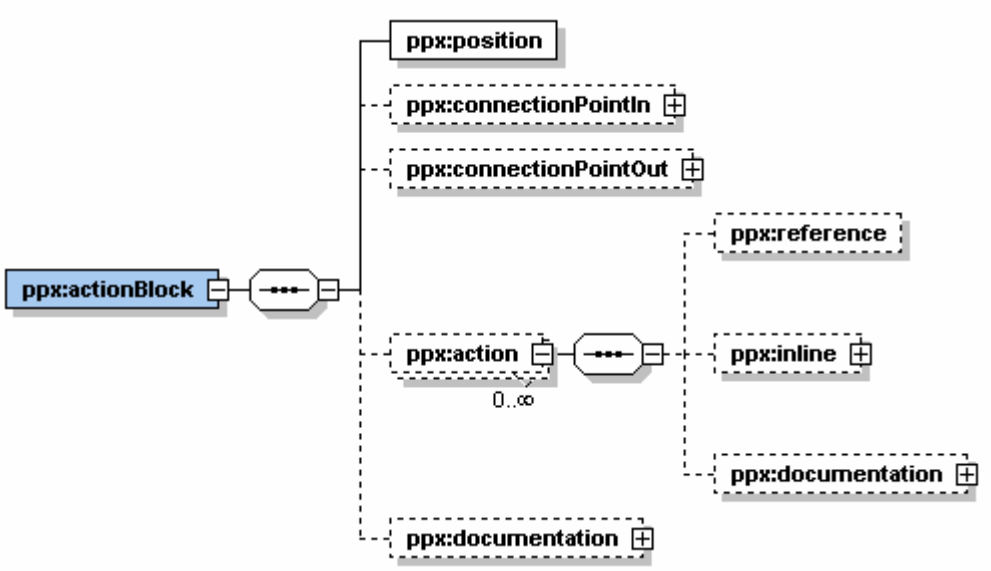


Figura 22. Struttura dell'elememto *actionBlock*

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*.

Il gruppo *fbdObject* contiene gli elementi che definiscono gli oggetti del linguaggio Functional Block Diagram.

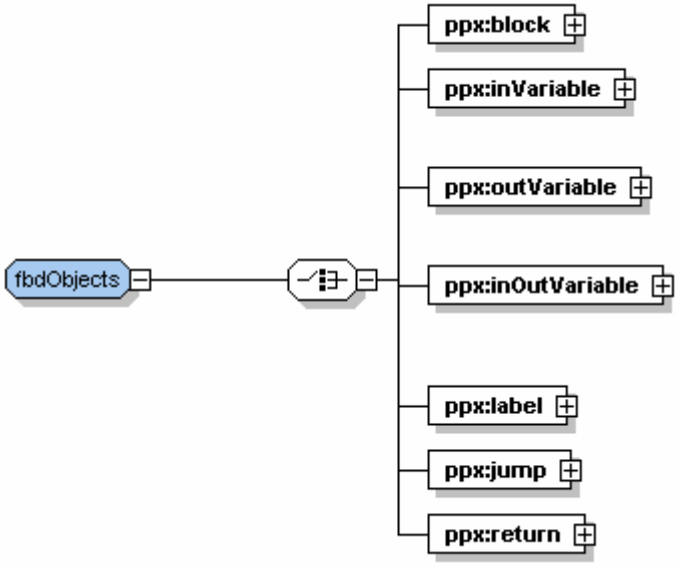


Figura 23. Il gruppo fbdObjects

In particolare l'elemento *block* rappresenta un blocco dell'FBD e dispone degli elementi che rappresentano le liste delle variabili di ingresso, di uscita e di ingresso-uscita.

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).

Infine, il gruppo *sfcObjects* contiene gli elementi che definiscono gli oggetti del linguaggio SFC: *step* (fase), *macroStep* (macrofase), *jumpSteps* (fase di salto), *transition* (transizione), *selectionDivergence* (divergenza selettiva), *selectionConvergence* (convergenza selettiva), *simultaneousDivergence* (divergenza simultanea) e *simultaneousConvergenge* (convergenza simultanea).

Un elemento *step* può contenere l'elemento *connectionPointOutAction* che esprime le coordinate grafiche del collegamento con un *actionBlock*. Un elemento macro*Step* contiene un proprio elemento *body* che implementa la macroazione. Un elemento *transition* dispone dell'elemento *condition* che esprime la condizione della transizione. Come per le azioni, la condizione di una transizione può far riferimento ad una transizione esterna al body (definita ed implementata nella *POU*) o può essere implementata direttamente nel proprio elemento *inline* di tipo *body*.

Figura 24. Il gruppo sfcObjects

L'elemento *jumpStep* è in aggiunta allo standard IEC 61131-3 e contiene l'attributo *targetName* che indica il punto in cui "saltare" quando la fase diventa attiva durante l'esecuzione del programma.

Gli elementi relativi alle convergenze e alle divergenze contengono gli elementi relativi alle connessioni con le fasi e le transizioni.

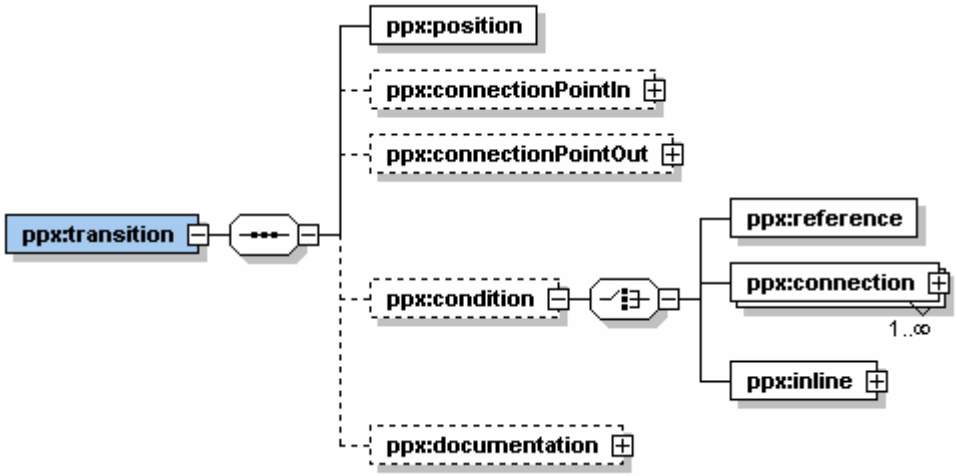


Figura 25. Struttura dell'elemento *transition*

Un *body*, come visto precedentemente, può contenere uno degli elementi IL, ST, FBD, LD o SFC. Gli elementi che definiscono oggetti grafici possono essere contenuti solo all'interno degli elementi FBD, LD ed SFC. In particolare:

- un elemento FBD può contenere elementi dei gruppi *fbdObjects* e *commonObjects;*
- un elemento LD può contenere elementi dei gruppi *ldObjects*, *fbdObjects* e *commonObject;*
- un elemento SFC può contenere elementi dei gruppi *sfcObjects*, *ldObjects*, *fbdObjects* e *commonObjects*.

L'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*.

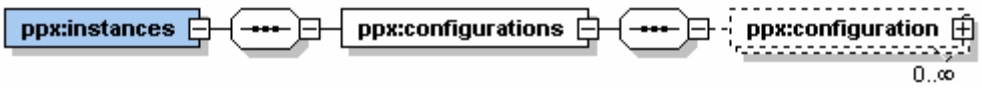


Figura 26. Struttura dell'elemento *instances*

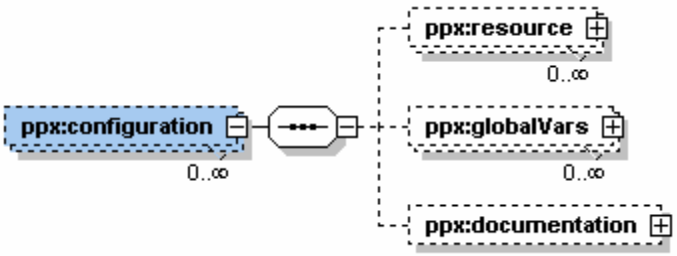


Figura 27. Struttura dell'elemento *configuration*

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*.

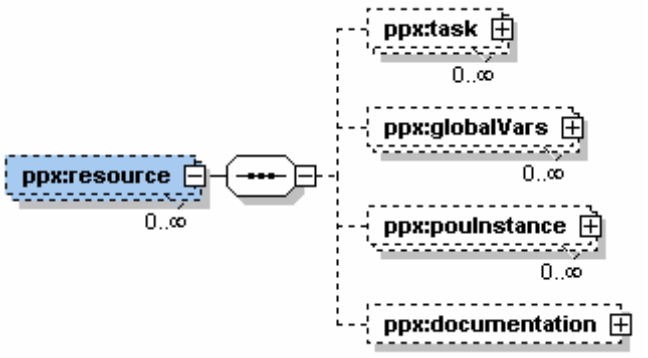


Figura 28. Struttura dell'elemento *resource*

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 sequente si riporta la struttura generale definita dello schema.

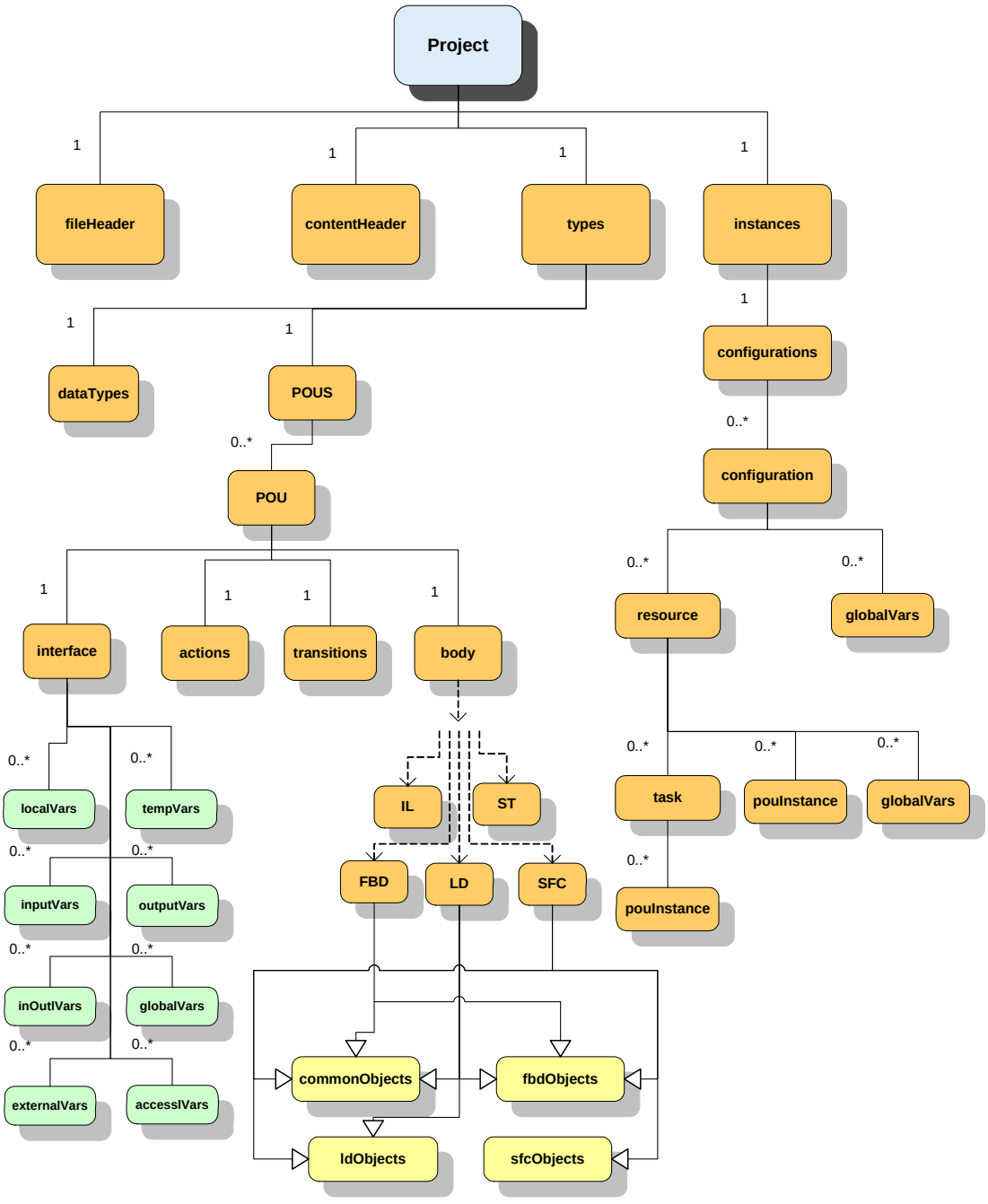


Figura 29. Struttura generale definita dallo schema

Capitolo 3
Il tool: architettura, caratteristiche, scelte progettuali

SiValPro fornisce i seguenti strumenti: un editor grafico per lo sviluppo dei progetti, un motore di simulazione, un motore di controllo che si interfaccia con dispositivi di I/O di tipo digitale e un'*XML validating parser* [3,11] che consente la validazione dei progetti importati. L'architettura del tool rispecchia il modello software definito dallo standard IEC 61131-3 e ripreso dal relativo XML Format for IEC 61131-3. Ciò consente lo sviluppo dei progetti e l'import/export da e verso altre piattaforme che adottano lo stesso standard e lo stesso formato per lo scambio delle informazioni.

La scelte effettuate in fase di progettazione del tool sono dovute in particolare alle finalità principali per cui è stato realizzato: lo sviluppo, la validazione e il rapid prototiping di progetti di automazione aderenti allo standard IEC 61131-3, con la possibilità di effettuarne il testing attraverso il controllo del processo. Si è preferito dunque porre particolare attenzione agli ambienti di sviluppo, di simulazione e di monitoraggio dello stato di esecuzione dei programmi ed alle funzioni di import/export delle informazioni. In questa ottica, ad esempio, il tool consente anche in corso di simulazione di effettuare la modifica dei programmi, di creare, eliminare o modificare i task, di passare da una gestione del multitasking di tipo preemptive al tipo non preemptive [11, 19] e viceversa, di variare la frequenza di campionamento delle variabili. Grazie agli strumenti di cui dispone, SiValPro si inserisce nel contesto dei sistemi di sviluppo di software di automazione, consentendo di completare il ciclo di sviluppo di un progetto dall'editing alla creazione di un prototipo.

Piattaforma e strumenti di sviluppo

Il tool è stato sviluppato su piattaforma Microsoft .NET Framework [13,14,16,17]. 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. Per i relativi dettagli si rimanda alla relativa documentazione [17].

3.1 Architettura generale

Dal punto di vista utente SiValPro fornisce due applicazioni indipendenti: una per l'editing, la simulazione e l'import/export dei progetti, ed un'altra per il controllo dei processi reali. Quest'ultima importa i progetti dal formato xml.

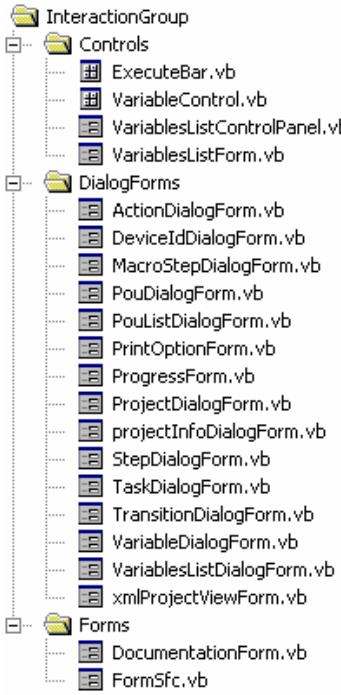
Dal punto di vista progettuale il tool è stato realizzato seguendo il paradigma object oriented. Dalla fase di analisi del modello software di riferimento definito dallo standard IEC 61131-3 è stata identificata una struttura ad oggetti di tipo gerarchico come descritta nel capitolo 1. La fase di progettazione 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:

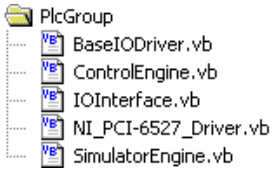
- *ProjectInfoGroup:* contiene le classi finalizzate alla memorizzazione delle informazioni relative al progetto;
- *TypeGroup:* contiene le classi che implementano i tipi di dati e le unità organizzative di programma (POU, Program Organization Units);
- *InstencesGroup:* contiene le classi che implementano 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

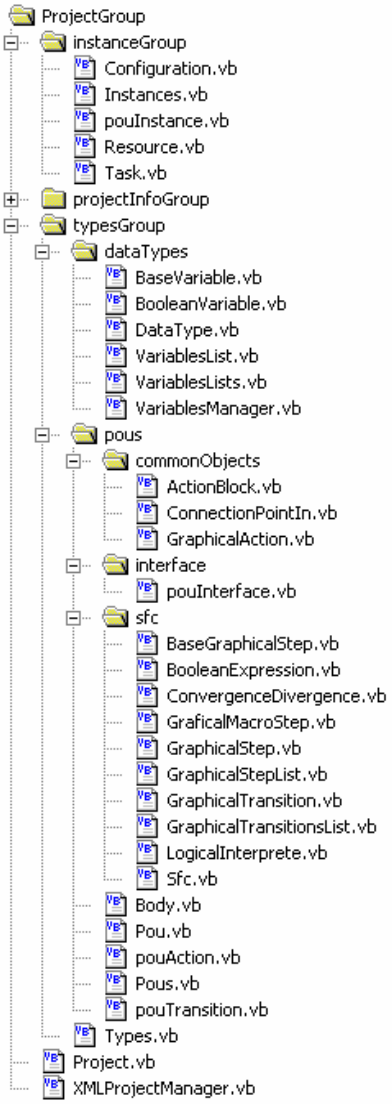


Figura 30. Suddivisione delle classi in gruppi

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.

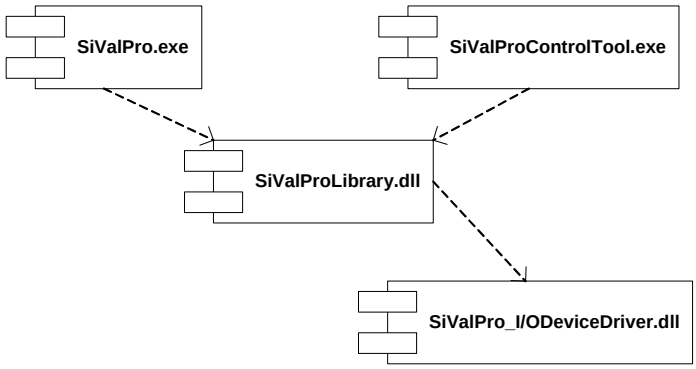


Figura 31. Component diagram

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, variabili, ecc.). In modo analogo vengono eseguiti l'import e l'export in formato xml del progetto.
La scelta di utilizzare un approccio di questo tipo è dovuta in particolare all'intento di realizzare uno strumento software dalla struttura altamente modulare e facilmente espandibile, considerando che non dovesse necessariamente rispondere a particolari requisiti di tipo hard- real time.

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*.

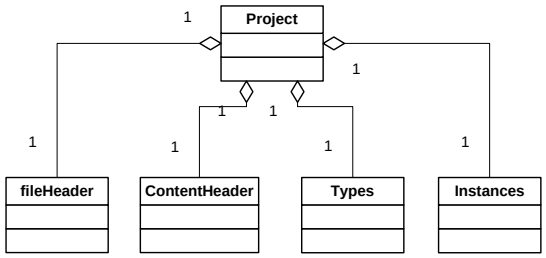


Figura 32. Class diagram 1

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* [16,17], che implementa una lista dinamica e fornisce i metodi per l'aggiunta, la rimozione e la ricerca in lista delle istanze dalla classe *Pou* .

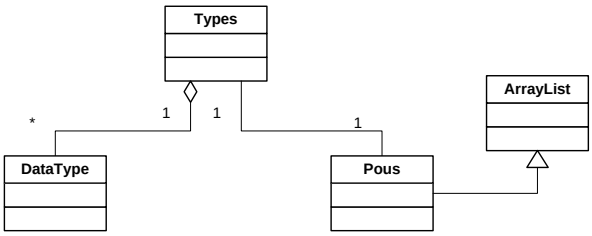


Figura 33. Class diagram 2

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.

L'oggetto *m_sfc* contenuto nella classe body è un'istanza della classe *sfc* e rappresenta il programma scritto nell'omonimo linguaggio.

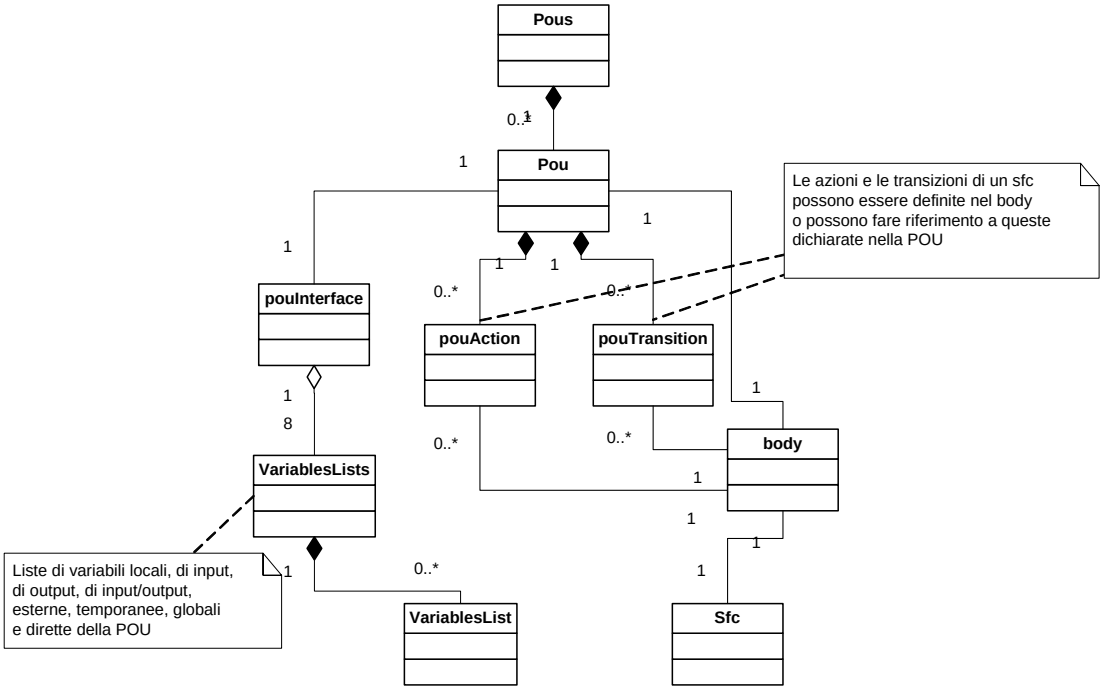
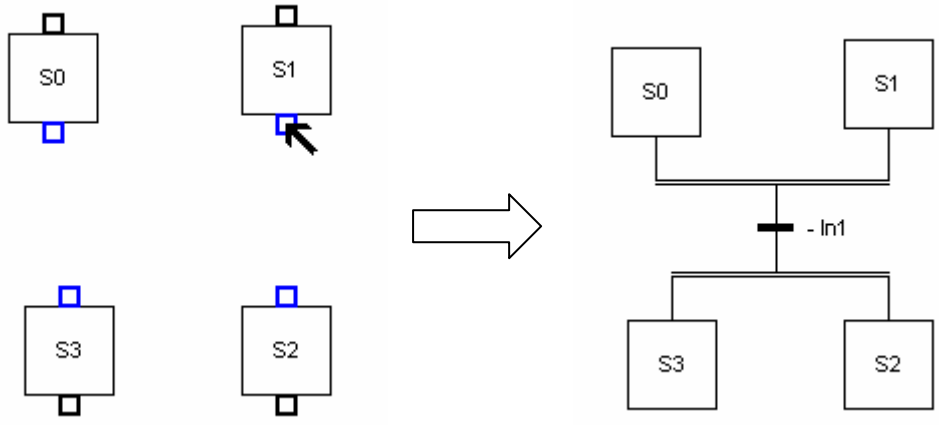


Figura 34. Class diagram 3

Implementazione dei programmi in linguaggio SFC

La classe *sfc* dispone degli attributi e dei metodi per l'editing dei programmi, per l'esecuzione degli stessi in fase di simulazione e controllo e per il relativo monitoraggio dell'evoluzione dello stato. Le caratteristiche di implementazione di questo linguaggio saranno descritte seguendo un approccio *down-top*, partendo dall'implementazione delle classi per gli oggetti definiti dal linguaggio. Questi ultimi sono: fase (classe *GraphicalStep*), macrofase (classe *GraphicalMacrostep*), blocco di azioni (classe *ActionBlock*), azione (classe *GraphicalAction*) e transizione (classe *GraphicalTransition*). Per semplificare l'editing dei programmi le divergenze e le convergenze sono gestite dalla classe *GraphicalTransition*. All'atto della definizione di una transizione l'utente stabilisce semplicemente le fasi precedenti e quelle successive ed automaticamente vengono tracciate graficamente le eventuali divergenze e/o convergenze.



a) ***b)***

Figura 35. Creazione automatica di divergenze e convergenze: a) selezione delle fasi; b) creazione delle divergenze e convergenze

Le classi *GraphicalStep* e *GraphicalMacroStep* ereditano la classe astratta *BaseGraphicalStep* e ne ridefiniscono i metodi virtuali. *BaseGraphicalStep* contiene gli attributi e i metodi che ne definiscono il comportamento comune. Insieme alla classe *GraphicalTransition* ed alla classe *GraphicalAction* 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 *BaseGraphicalStep* e la classe *GraphicalTransition* contengono l'attributo booleano *m_drawstate* che stabilisce se devono essere disegnati i rispettivi indicatori di stato: per la fase si riferisce allo stato attivo o meno, mentre per la transizione alla condizione vero o falsa.

Lo stato di una fase è indicato dall'attributo di tipo booleano *m_active*. I metodi per l'attivazione della fase nella classe *BaseGraphicalStep* sono virtuali. Le due classi che la ereditano ne definiscono implementazioni diverse: *GraphicalStep* imposta l'attributo *m_active* su true e memorizza l'istante di attivazione leggendolo dal timer di sistema, ed eventualmente ridisegna l'indicatore di stato se richiesto. Il codice è il seguente:

```
Public Overrides Sub Active()
   Dim StateChanged As Boolean = Not m_Active
   m_Active = True
   m_TimeActivation = Now  'Memorizza l'istante di attivazione
   m_PreActive = False
   'Disegna l'indicatore di stato se richiesto
   If StateChanged And DrawState Then
      DrawStepState(False)
   End If
End Sub
```

L'istante di attivazione (istruzione *m_TimeActivation = Now*) è necessario ai fini dell'esecuzione delle azioni temporizzate come si vedrà successivamente. La classe *GraphicalMacroStep* lo ridefinisce semplicemente impostando su true l'attributo *m_active*:

```
Public Overrides Sub Active()
   m_Active = True
End Sub
```

Una macrofase infatti, nell'implementazione realizzata dell'algoritmo di esecuzione dell'sfc, viene attivata solo quando termina l'esecuzione del suo body. Nel frattempo rimane solo 'preattiva' (attributo *m_preactive = true*).
L'attributo m_preactive è presente anche nella fase. In questo caso però ha un'altra finalità come si vedrà in seguito.
La classe *GraphicalTransition* contiene un istanza della classe *BooleanExpression* (*m_Condition*). Questo oggetto rappresenta l'espressione booleana che determina la condizione della transizione. L'espressione viene definita dall'utente all'atto della creazione della transizione e può essere modificata successivamente.

L'interprete di espressioni booleane
La classe *BooleanExpression* implementa un interprete di espressioni booleane. Il metodo *SetExpression* riceve come parametro una stringa contenente l'espressione ed effettua una chiamata al metodo *parse*, passandogli la stessa stringa. Questo metodo effettua il parsing dell'espressione, controllandone la correttezza sintattica e semantica. In caso di errore fornisce la posizione nella stringa dei caratteri inaspettati. Se il controllo va a buon termine viene creata una struttura ad albero i cui nodi rappresentano gli operatori logici e le variabili dell'espressione. I nodi sono le istanze delle classi, di cui si riporta il codice, che rappresentano rispettivamente

l'operatore OR, l'operatore AND, un riferimento ad una variabile in memoria ed un confronto temporale tra il tempo trascorso dall'istante di attivazione di una fase ed un intervallo di tempo specificato:

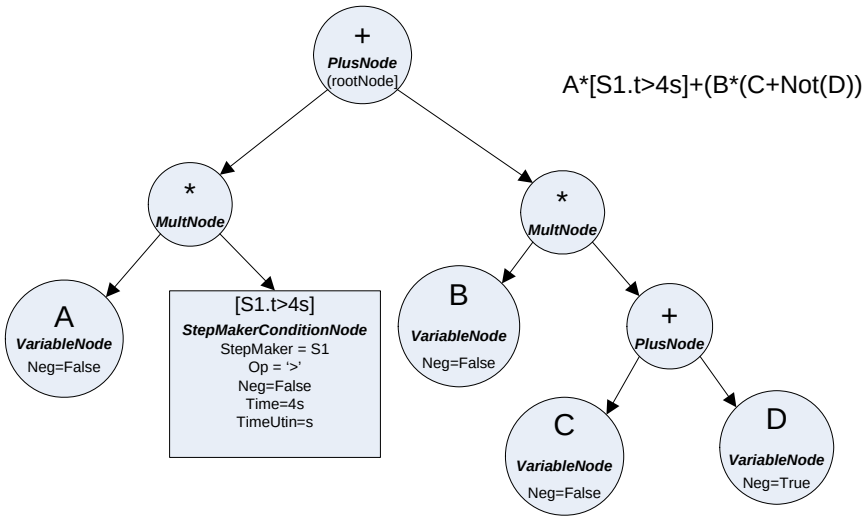
```
Public Class PlusNode
   Public NextNodes As ArrayList   'Lista dei nodi successivi
   Public Neg As Boolean        'Indica che l'operatore è preceduto da una negazione
End Class

Public Class MultNode
   Public NextNodes As ArrayList   'Lista dei nodi successivi
   Public Neg As Boolean        'Indica che l'operatore è preceduto da una negazione
End Class

Public Class VariableNode
   Public WithEvents Var As BaseVariable  ''Riferimento ad una variabile
   Public Neg As Boolean         'Indica che la variabile è preceduta da una negazione
   Public Event VarNameChanged()  'Evento generato al cambio del nome di Var
End Class

Public Class StepMakerConditionNode
   'Esempio sintassi: [101.T>10s]
   Public StepMaker As BaseGraphicalStep  'Riferimento ad una fase
   Public Op As Char              'Memorizza l'operatore di confronto '<' oppure '>'
   Public Neg As Boolean          'Indica che è preceduto da una negazione
   Public Time As TimeSpan        'Intervallo di tempo
   Public TimeUnit As String      'Memorizza l'unità di tempo scelta
End Class
```

Nella figura successiva vi è un esempio della struttura creata per una espressione.



I nodi che contengono le istanze della classe *VariableNode* puntano direttamente alla variabile allocata in memoria attraverso l'oggetto *Var*, di tipo *BaseVariable*. I nodi contenenti istanze della classe *StepMakerConditionNode* puntano direttamente alla fase allocata in memoria attraverso l'oggetto *StepMaker*, di tipo *BaseGraphicalStep*. L'albero con i relativi nodi viene creato immediatamente dopo l'immissione dell'espressione da parte dell'utente. In questo modo il valore dell'espressione può essere immediatamente calcolato in fase di simulazione senza la necessità di passare per una fase di compilazione. Inoltre essendo già stata effettuata la traduzione dei nomi in indirizzi fisici di memoria le operazioni di calcolo risultano estremamente più veloci. In figura è riportata la classe *BooleanExpression*.

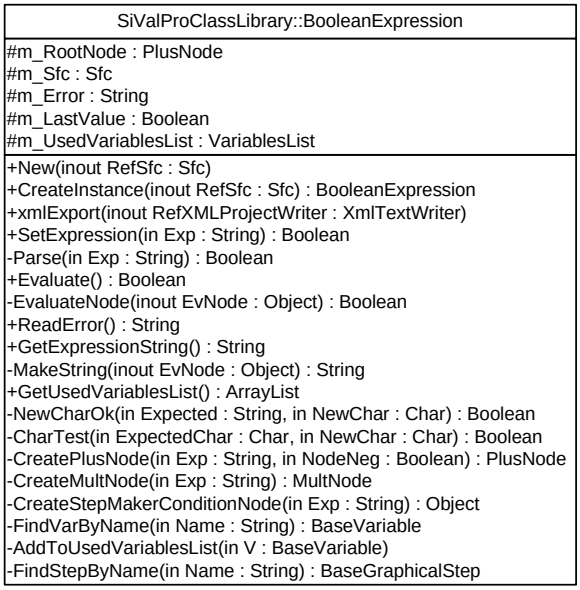


Figura 36. La classe BooleanExpression

Si nota il metodo *Evaluate()* che effettua il calcolo dell'espressione e restituisce i valori *true* o *false*. I metodi privati Parse(), PlusNode() e CreateMultNode, CreateStepMakerConditionNode() sono utilizzati dal metodo SetExpression per costruire la struttura ad albero dell'espressione e per verificarne la correttezza. Infine il metodo GetUsedVariablesList() restituisce la lista dei riferimenti alle variabili utilizzate nell'espressione.

Implementazione delle azioni

Le azioni di un SFC sono implementate dalla classe *GraphicalAction*. Oltre agli attributi ed ai metodi finalizzati al disegno contiene quelli finalizzati all'esecuzione delle azioni. L'attributo *m_quelifier* esprime il qualificatore dell'azione e può assumere uno dei seguenti valori definiti dallo standard: "N", "S", "R", "L", "D", "P", "SD", "DS" ed "SL". L'attributo *m_Time* di tipo *TimeSpan* memorizza l'intervallo di tempo necessario per l'esecuzione delle azioni temporizzate. Inoltre due attributi di tipo *BooleanVariable* contengono rispettivamente la variabile controllata dall'azione (*m_var*) e la variabile indicatore (*m_Indicator*). I metodi per l'esecuzione delle azioni sono diversi a seconda che si tratti di un'azione di tipo impulsivo o meno. Il metodo *ExecuteIfIsNotPulseAction()* esegue le azioni non impulsive e viene eseguito in seguito all'attivazione della fase che contiene le azioni. Il metodo *ExecuteIfIsPulseAction()* esegue le azioni impulsive e viene chiamato contestualmente alla 'preattivazione' della fase. Il metodo *StopIfIsNotPulseAction()* termina le azioni non impulsive contestualmente alla disattivazione della fase. Nel caso di azioni temporizzate, lo scadere di un timer (*m_ActionTimer*), eseguito in un diverso thread, determina l'esecuzione del metodo *OnEndTimer()* che si occupa di iniziare o terminare l'esecuzione dell'azione. Tutti i metodi contengono il costrutto di selezione di tipo

```
Select Case m_Qualifier
        Case ……..
       ……..
        Case ……..
       ……..
       ……..
End Select
```

che a seconda del valore del qualificatore ne 'seleziona operazioni diverse. L'inizio dell'azione è determinato dall'istruzione *m_Var.SetValue(True)* che imposta su *true* la variabile controllata. L'azione è terminata dall'istruzione *m_Var.SetValue(False*). L'implementazione dei metodi è riportata di seguito con gli opportuni commenti.

```
Public Sub ExecuteIfIsNotPulseAction()
  'Esegue l'azione se è qualificata come N, L ,SL, D, DS o SD
  If Not IsNothing(m_Var) Then 'La variabile può essere stata eliminata
    Try
      Select Case m_Qualifier
        Case "N"   'N
          m_Var.SetValue(True)
```

```
                    'Variabile indicatore
                    If Not IsNothing(m_Indicator) Then
                        m_Indicator.SetValue(True)
                    End If
                Case "L", "SL"    'Time limited, Set and time limited
                    m_Var.SetValue(True)
                    'Variabile indicatore
                    If Not IsNothing(m_Indicator) Then
                        m_Indicator.SetValue(True)
                    End If
                    m_ActionTimer = New Timer(m_TimerDelegate, m_Qualifier,_
                    _CInt(m_Time.TotalMilliseconds), -1)
                Case "D", "SD", "DS"  'Time delay,  Set and delay, Delayed and Set
                    m_ActionTimer = New Timer(m_TimerDelegate, m_Qualifier,_
                    _CInt(m_Time.TotalMilliseconds), -1)
            End Select
        Catch ex As System.Exception
        End Try
    End If
End Sub

Public Sub ExecuteIfIsPulseAction()
    'Esegue l'azione se è qualificata come S, R o P
    If Not IsNothing(m_Var) Then 'La variabile può essere stata eliminata
        Try
            Select Case m_Qualifier
                Case "S"    'Set
                    m_Var.SetValue(True)
                Case "R"    'Reset
                    m_Var.SetValue(False)
                Case "P"    'Pulse
                    m_Var.SetValue(True)
            End Select
        Catch ex As System.Exception
        End Try
    End If
End Sub

Public Sub StopIfIsNotPulseAction()
    'Termina l'azione se è qualificata come N, L , D o DS
    If Not IsNothing(m_Var) Then            'La variabile può essere stata eliminata
        Try
            Select Case m_Qualifier
                Case "N"
                    m_Var.SetValue(False)
                    'Variabile indicatore
                    If Not IsNothing(m_Indicator) Then
                        m_Indicator.SetValue(False)
                    End If
                Case "L", "D", "DS"  'Time limited, Time delay, Delayed and Set
                    m_Var.SetValue(False)
                    If Not IsNothing(m_Indicator) Then
                        m_Indicator.SetValue(False)
                    End If
                    'Distrugge il timer se non è stato ancora distrutto
                    If Not IsNothing(m_ActionTimer) Then
                        m_ActionTimer.Dispose()
                    End If
            End Select
```

```
        Catch ex As System.Exception
        End Try
    End If
End Sub

Private Sub OnEndTimer(ByVal Qual As Object)
'Esegue o termina l'azione a seconda del valore del qualificatore
    If Not IsNothing(m_Var) Then 'La variabile può essere stata eliminata
        Try
            Select Case m_Qualifier
                Case "L", "SL"    'Time limited, Set and time limited
                    m_Var.SetValue(False)
                    If Not IsNothing(m_Indicator) Then
                        m_Indicator.SetValue(False)
                    End If
                Case "D", "SD", "DS"  'Time delay, Set and delay, Delayed and Set
                    m_Var.SetValue(True)
                    If Not IsNothing(m_Indicator) Then
                        m_Indicator.SetValue(True)
                    End If
            End Select
            'Distrugge il timer se non è stato ancora distrutto
            If Not IsNothing(m_ActionTimer) Then
                m_ActionTimer.Dispose()
            End If
        Catch ex As System.Exception
        End Try
    End If
End Sub
```

Implementazione della struttura di un SFC

Le azioni di una fase sono contenute in una lista dinamica (*m_ActionList*) di un oggetto di tipo *ActionBlock*. Un oggetto di questo tipo è contenuto in un oggetto *GraphicalStep* che rappresenta la fase contenente il blocco di azioni.

Tutti gli oggetti di tipo *GraphicalStep* e *GraphicalTransition* sono contenuti rispettivamente in due liste dinamiche di un oggetto *sfc*, *m_GraphicalStepList*, istanza della classe *GraphicalStepList, e m_GraphicalTransition*, istanza della classe *m_GraphicalTransitionList*. Queste due classi ereditano la classe *ArrayList* e dispongono degli attributi e dei metodi per l'aggiunta, la rimozione e il disegno delle fasi e delle transizioni e dei metodi per l'implementazione dell'esecuzione dell'SFC.

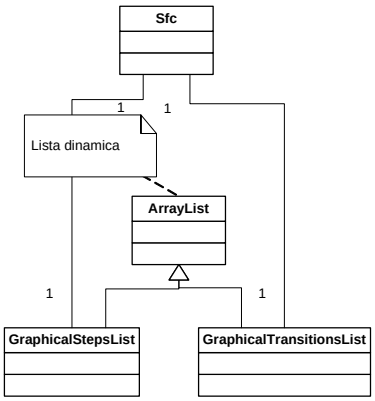


Figura 37. Class diagram 5

Un oggetto di tipo *GraphicalTransition* dispone inoltre degli oggetti *m_PreviousGraphicalStepsList* e *m_NextGraphicalStepsList*, entrambi istanze della classe *GraphicalStepList*. Queste liste contengono rispettivamente i riferimenti alle fasi precedenti alla transizione ed alle fasi successive. Il metodo che effettua il disegno della transizione legge le posizioni sul piano delle suddette fasi 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 transizione risulta superabile o meno e per l'attivazione e la disattivazione delle fasi.

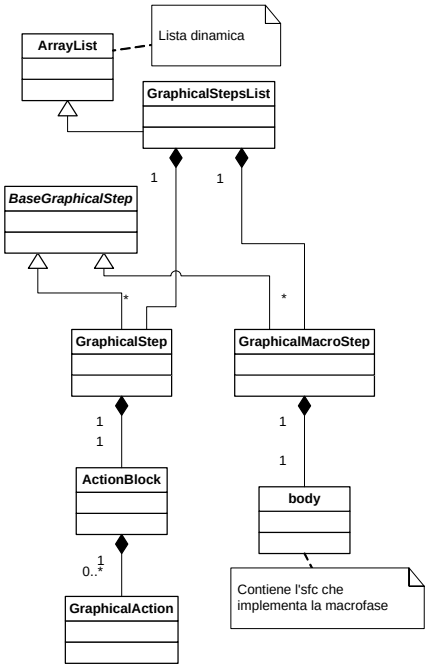


Figura 38. Class diagram 5

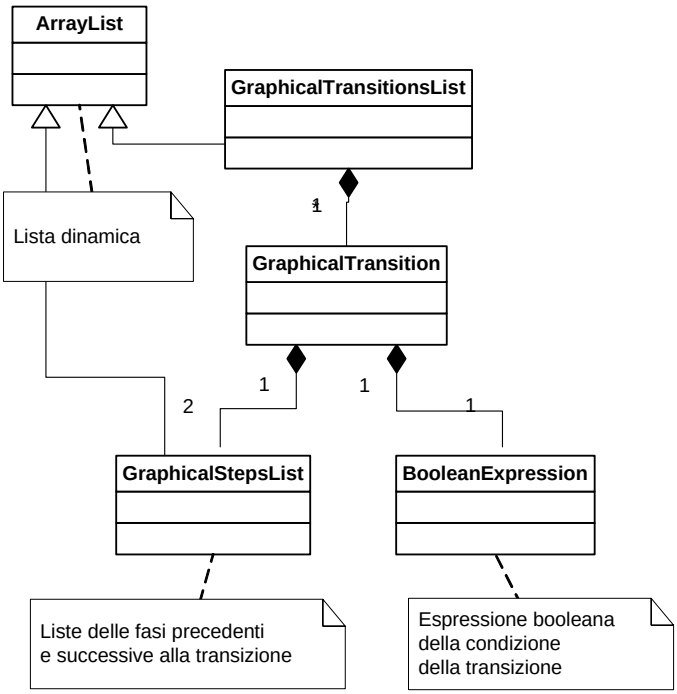


Figura 39. Class diagram 6

Algoritmo di esecuzione dell'SFC

L'algoritmo di esecuzione dell'SFC implementa la ricerca di stabilità [1]. E' un algoritmo di tipo ciclico in cui in ogni ciclo vengono effettuate in sequenza le seguenti operazioni:

- vengono determinate le transizioni superabili;
    - se almeno una transizione risulta superabile viene effettuato il ciclo di 'preattivazione' delle fasi, cioè per ogni transizione superabile vengono disattivate le fasi precedenti (l'attibuto *m_active* viene impostato su *false*) e 'preattivate' quelle successive (l'attibuto *m_preactive* viene impostato su *true*);
    - se nessuna transizione è superabile e le fasi preattive non sono state già attivate dopo l'ultimo ciclo di preattivazione, si procede all'attivazione di queste ultime (l'attibuto *m_active* viene impostato su *true*).;
- viene effettuato il ciclo di scansione per le macrofasi che risultano preattive (l'algoritmo è lo stesso, ma in questo caso viene eseguito per l'SFC del body della macrofase);

- vengono eseguite le azioni.

Al termine di ogni ciclo ogni fase dell'sfc può trovarsi in uno dei seguenti stati: preattiva, attiva o disattiva. Per essere attivata si deve verificare che nel ciclo precedente deve essere stata preattivata e che nel successivo nessuna transizione deve essere superabile. Quest'ultima condizione coincide con il raggiungimento della stabilità. In questo modo se dopo un ciclo una fase risulta preattiva e nel ciclo successivo vi sono transizioni superabili si verificano uno dei due casi:

- se una transizione in uscita dalla fase è superabile, la fase viene disattivata senza essere attivata;
- se nessuna transizione in uscita risulta superabile la fase resta preattiva, in attesa che il resto del programma raggiunga la stabilità.

Un algoritmo di questo tipo è particolarmente utile per monitorare lo stato delle fasi e il raggiungimento di una condizione stabile. Le funzioni di disegno della fase permettono di visualizzare sullo schermo, utilizzando colori diversi, lo stato in cui essa si trova ad ogni ciclo di scansione dell'sfc. In particolare nell'esecuzione a singoli passi scanditi dall'utente risulta molto chiara l'evoluzione dello stato dell'intero programma.

Le azioni vengono eseguite al termine di ogni ciclo attraverso la chiamata al metodo *m_GraphicalStepsList.ExecuteAction()* che determina l'esecuzione delle azioni delle fasi preattive o attive. Una volta preattivata una fase, devono essere eseguite solo le azioni impulsive ed in seguito all'attivazione anche quelle non impulsive. Queste operazioni sono effettuate dal metodo seguente della classe *GraphicalStep*.

```
Public Sub ExecuteActions()
    If Not m_preactive And Not m_Active Then
        If m_NotPulseActionExecuted Then
            'Termina le azioni non impulsive indipendenti da un temporizzatore
            StopNotPulseActions()
            m_NotPulseActionExecuted = False
        End If
        m_PulseActionExecuted = False
    Else
        'Esegue le azioni impulsive o non impulsive se non erano già eseguite
        If m_preactive And Not m_PulseActionExecuted Then
            ExecutePulseActions()
            m_PulseActionExecuted = True
        Else
            If m_active And Not m_NotPulseActionExecuted Then
                'Esegue le azioni non impulsive
                ExecuteNotPulseActions()
                m_NotPulseActionExecuted = True
            End If
        End If
    End If
```

```
End Sub
```

La classe *sfc* dispone dei seguenti metodi:

- *ExecuteInit()*
- *ExecuteScanCycle()*
- *Reset()*

Il metodo *ExecuteInit()* effettua le operazioni di inizializzazione attraverso la chiamata del metodo *m_GraphicalStepsList.ExecuteInit()*. Quest'ultimo preattiva tutte le fasi iniziali (attributo *m_initial = true*). Il metodo *ExecuteScanCycle*() esegue un singolo ciclo dell'algoritmo di esecuzione dell'sfc. Il codice è riportato in seguito.

```
Public Function ExecuteScanCycle() As Boolean
   'Genera l'evento di inizio scansione
   RaiseEvent StartScan()
      'Effettua la scansione dell'SFC
      'Monitor sull'SFC e su GraphToDraw per disegnare lo stato
      Try
         If Monitor.TryEnter(Me, 100) Then
            'Inizio ciclo di scansione
            m_ConditionChanged = m_GraphicalTransitionsList.ExecuteScanCycle
            If m_ConditionChanged Then
               'Se la condizione è cambiata preattiva e depreattiva le fasi
               m_GraphicalTransitionsList.ExecutePreactivationCycle()
               m_ActivatedPreactivedSteps = False
               'Disegna lo stato sul form se richiesto
            Else
               'se la condizione non è cambiata la prima volta attiva le fasi (non macrofasi)
                'preattive
               'Se viene attivata almeno una fase finale memorizza a restituisce True
               If Not m_ActivatedPreactivedSteps Then
                  ExecuteScanCycle = m_GraphicalStepsList.ActivePreactivedSteps()
                  m_ActivatedPreactivedSteps = True
               End If
            End If
            'Ciclo di scansione per le macrofasi
            m_GraphicalStepsList.ExecuteMacroStepScanCycle()
            'Esecuzione delle azioni
            m_GraphicalStepsList.ExecuteAction()
            'Fine ciclo di scansione
            Monitor.Exit(Me)
         End If
      Catch ex As System.Exception
         Monitor.Exit(Me)
      End Try
   End If
   'Genera l'evento di fine scansione
   RaiseEvent EndScan()
End Function
```

Infine il metodo *Reset()* attraverso in metodi *m_GraphicalStepsList.ResetState()* e

m_GraphicalTransitionsList.ResetState() disattiva tutte le fasi, termina tutte le azioni in corso e imposta l'attributo *m_superable* di tutte le transizioni su *false*. Anche i metodi *ExecuteInit()* e *Reset()* effettuano un blocco sull'sfc per eseguire le operazioni.

Nel codice riportato precedentemente si nota l'utilizzo dell'oggetto *monitor* [16,17]. Lo scopo è quello di bloccare ad uso esclusivo l'intero sfc per garantire che durante ogni singolo ciclo di esecuzione il programma non sia modificato dall'utente. Ciò permette di editare il programma anche in corso di esecuzione, evitando accessi simultanei agli oggetti condivisi dalle funzioni di esecuzione del programma e da quelle di editing. Le modifiche possono essere infatti effettuate tra un ciclo ed il successivo, nel quale arco di tempo gli oggetti devono essere acquisiti ad uso esclusivo dalle suddette funzioni di editing. I tentativi di acquisizione dell'oggetto sfc sono temporizzati, cioè se dopo un determinato tempo l'oggetto non risulta disponibile il ciclo viene saltato per evitare di bloccare il thread che esegue il programma a causa di un eventuale modifica di quest'ultimo da parte dell'utente. Il tentativo di acquisire l'sfc viene così ripetuto nella successiva chiamata.

L'intero algoritmo di evoluzione dell'sfc viene eseguito mediante una serie di collaborazioni tra gli oggetti contenuti in esso. Ogni oggetto coinvolto nell'esecuzione dispone dei metodi che effettuano le operazioni definite su di esso e che a loro volta chiamano i metodi degli oggetti contenuti. Il sequence diagram della pagina successivaè relativo ad uno scenario in cui si raggiunge una condizione stabile.

Le chiamate ai metodi *ExecuteInit()*, *ExecuteScanCycle()* e *Reset()* di un oggetto sfc sono effettuate da parte dell'oggetto body che lo contiene. A sua volta la chiamata al metodo del body viene effettuata dal relativo oggetto pou.

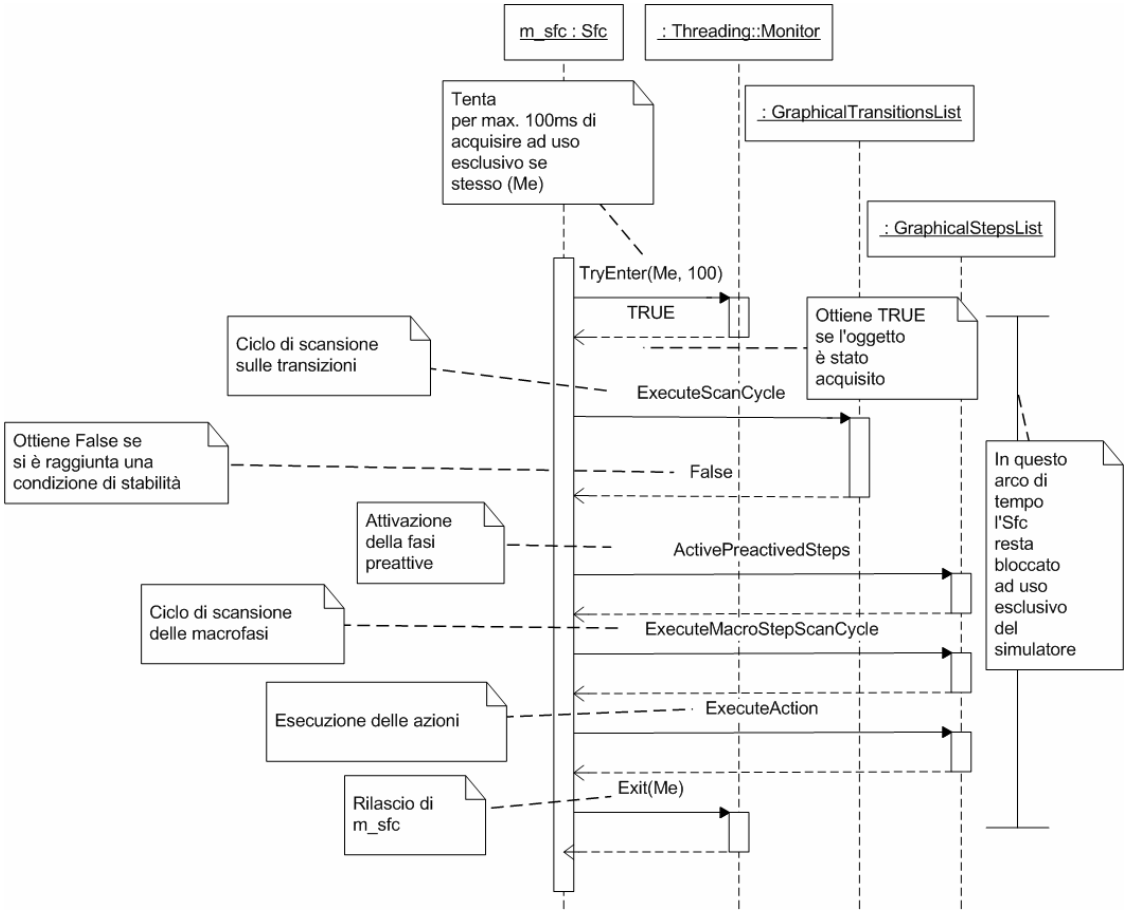


Figura 40. Sequence diagram 1

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[4] 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).

[4] Anche in questo caso, come nel modello software dello standard IEC 61131-3, una risorsa rappresenta un dispositivo in grado di eseguire programmi.

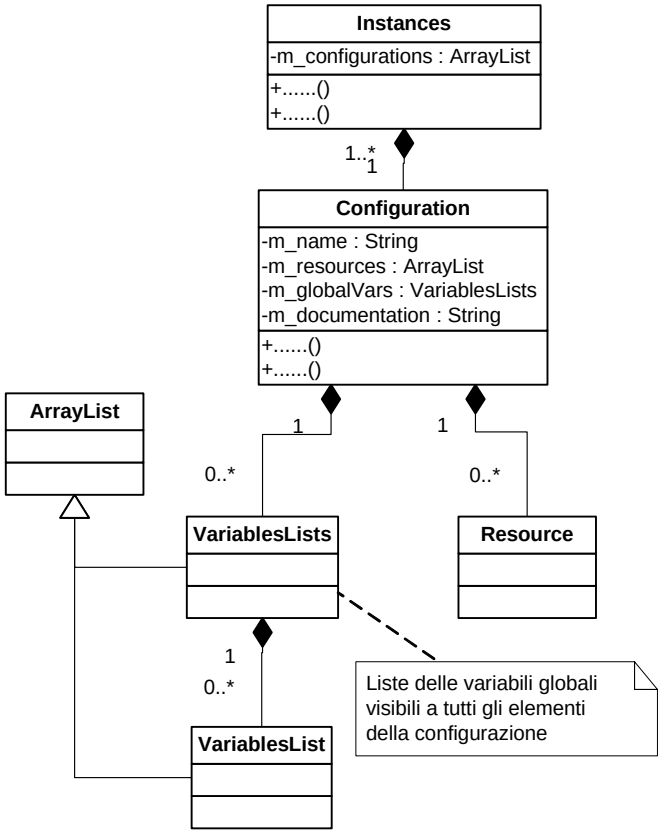


Figura 41. Class diagram 7

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 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.

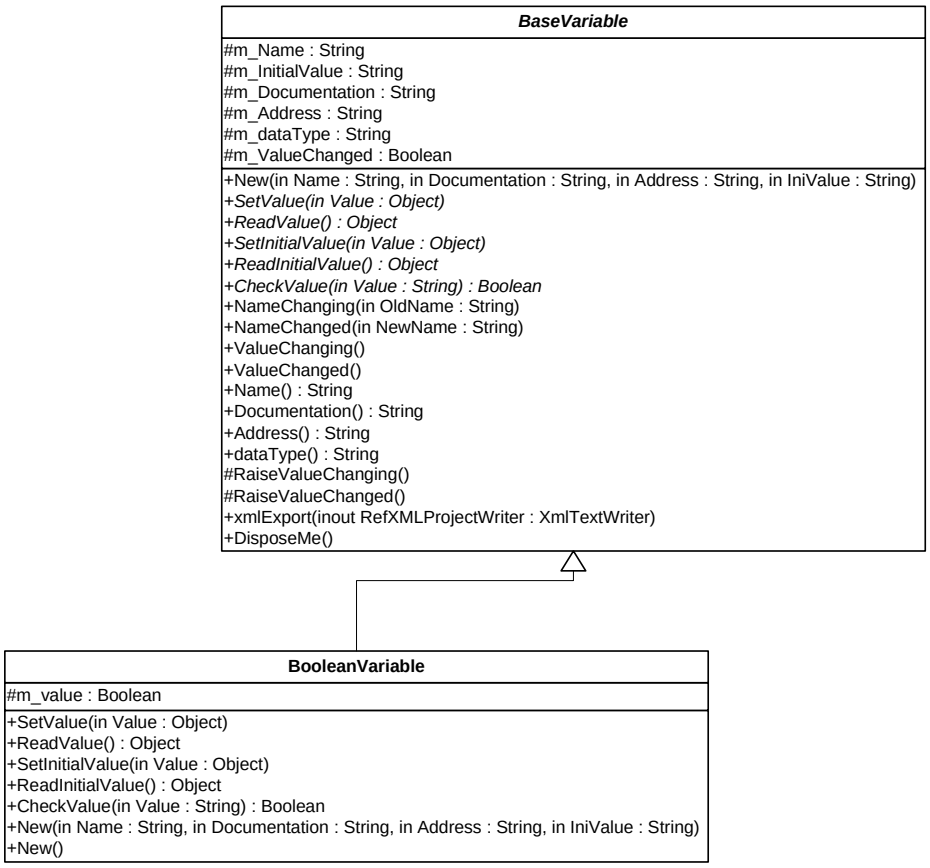


Figura 42. Class diagram 8

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.

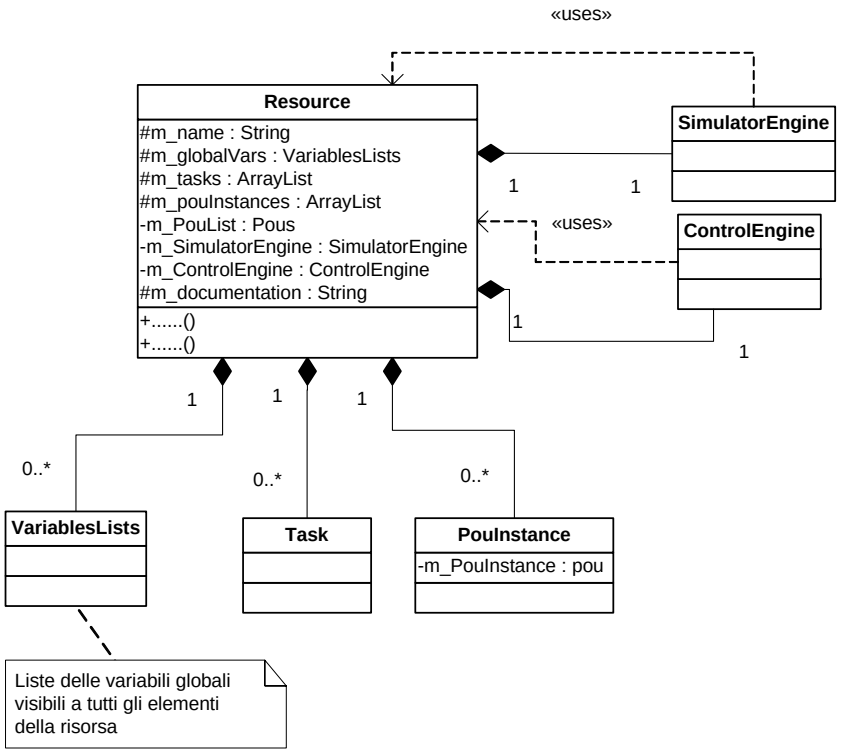


Figura 43. Class diagram 9

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.

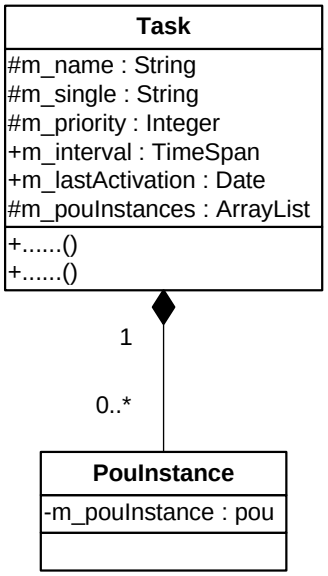


Figura 44. Class diagram 10

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. Il sequence diagram della pagina seguente mostra la creazione di un'istanza (alcuni oggetti sono stati omessi per semplicità).

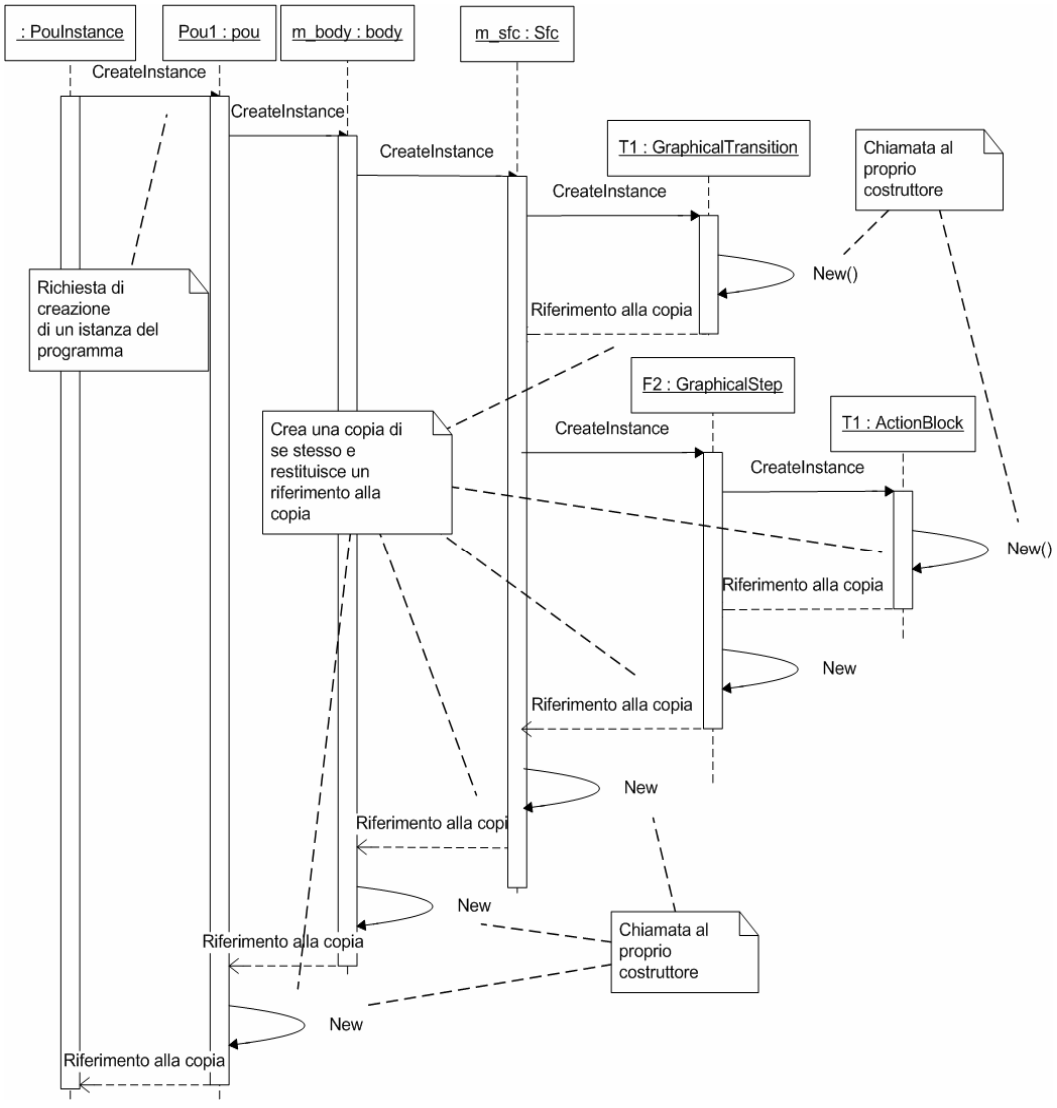


Figura 45. Sequence diagram 2

3.3 Il simulatore di controllo

In questo paragrafo vengono illustrate le funzioni, con le relative implementazioni, che consentono di simulare il controllo di un processo.

Ogni oggetto di tipo resource contiene un'istanza della classe *EngineSimulator*. Questa classe implementa le funzioni di controllo di simulazione, gli algoritmi di controllo dell'esecuzione e le funzioni di schedulazione dei task. Implementa sia un multitasking di tipo preemptive che non preemptive.

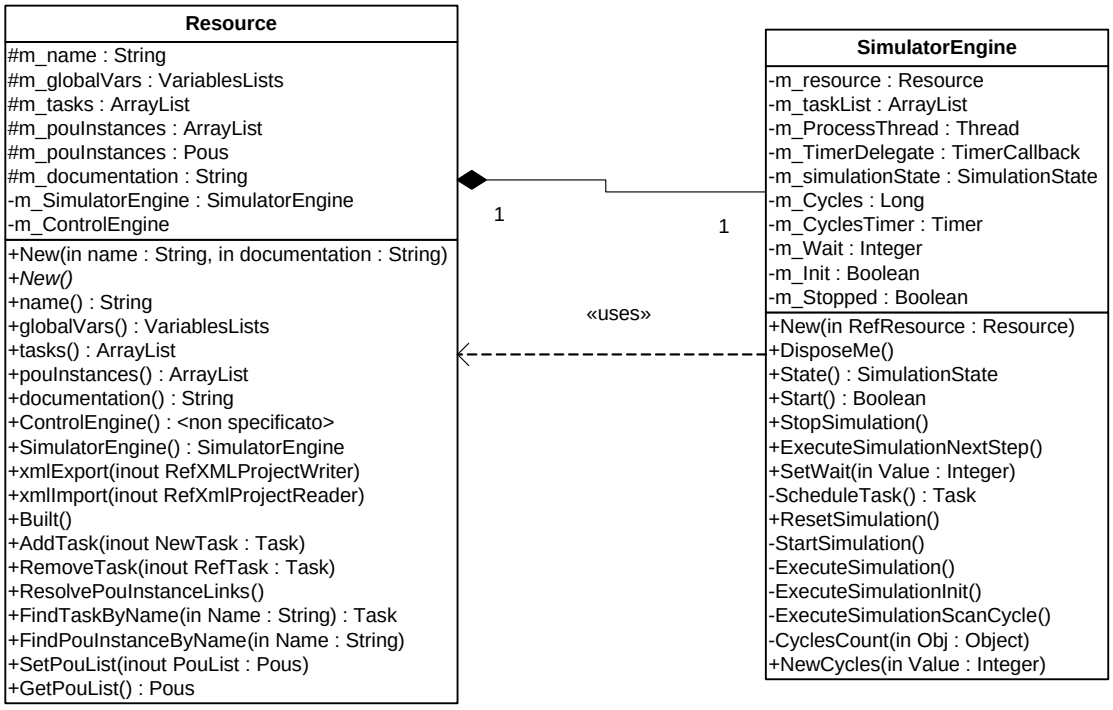


Figura 46. Class diagram 11

Gestione dell'esecuzione della simulazione

I metodi pubblici *Start()*, *StopSimulation()* ed *ExecuteNextSimulationStep()* consentono rispettivamente di avviare l'esecuzione, di arrestarla e di eseguire un singolo passo di esecuzione. La proprietà[5] *SimulationState* fornisce lo stato della simulazione (*Run* o *Stop*). Il metodo *SetSimulationWait()* consente di variare la priorità di esecuzione della simulazione rispetto agli altri thread del processo principale in esecuzione. Questa funzione è utile per concedere più risorse di sistema

[5] Per una descrizione relativa all'utilizzo delle proprietà in ambiente .Net Framework si veda [15,16,17]

ai thread che eseguono ad esempio le funzioni di editing, se la relativa esecuzione risulta particolarmente lenta. Infine l'evento pubblico *NewCicles* fornisce il numero di cicli effettuati nell'ultimo secondo di tempo e viene generato con lo stesso periodo.

Quando viene creato un oggetto *resource* il costruttore crea un'istanza della classe *SimulatorEngine* passando al relativo costruttore un riferimento alla risorsa. Il codice è il seguente:

```
Public Sub New(ByVal name As String, ByVal documentation As String)
    m_name = name
    m_documentation = documentation
    m_tasks = New ArrayList
    m_globalVars = New VariablesLists
    m_pouInstances = New ArrayList
    m_SimulatorEngine = New SimulatorEngine(Me)
    m_ControlEngine = New ControlEngine(Me)
End Sub
```

Il riferimento alla risorsa viene utilizzato dall'istanza di *SimulatorEngine* per referenziare la lista dei task e delle istanze delle POU senza task. L'esecuzione di queste ultime deve avvenire con intervallo ciclico nullo e la relativa priorità deve essere la minima. Per ottenere ciò all'interno del costruttore della classe *SimulatorEngine*, come si nota dal codice riportato, crea un task con gli attributi *m_interval* con valore 0 e *m_priority* con valore 65535.

```
Public Sub New(ByVal RefResource As Resource)
    m_TimerDelegate = New TimerCallback(AddressOf CyclesCount)
    m_Wait = 0
    m_Jumps = 0
    m_resource = RefResource
    m_taskList = m_resource.tasks
    m_TaskThreadsKilled = True
    'Configura il task per i pou della risorsa
    m_ResoursePouInstanceTask = New Task("", 65636, New TimeSpan(0),_
_m_resource.pouInstances)
    m_TaskExecuted = m_ResoursePouInstanceTask
End Sub
```

L'esecuzione della simulazione viene gestita da un thread principale che viene creato nel relativo istante di avvio.

Il thread esegue il metodo privato *StartSimulation*(). Quest'ultimo effettua l'inizializzazione dei programmi e delle variabili chiamando il metodo *ExecuteSimulationInit()* (se non è già stata effettuata in precedenza) e successivamente determina l'inizio del ciclo di simulazione con la chiamata al metodo *ExecuteSimulation( )* .

Il seguente sequence diagram mostra le creazione del thread di simulazione.

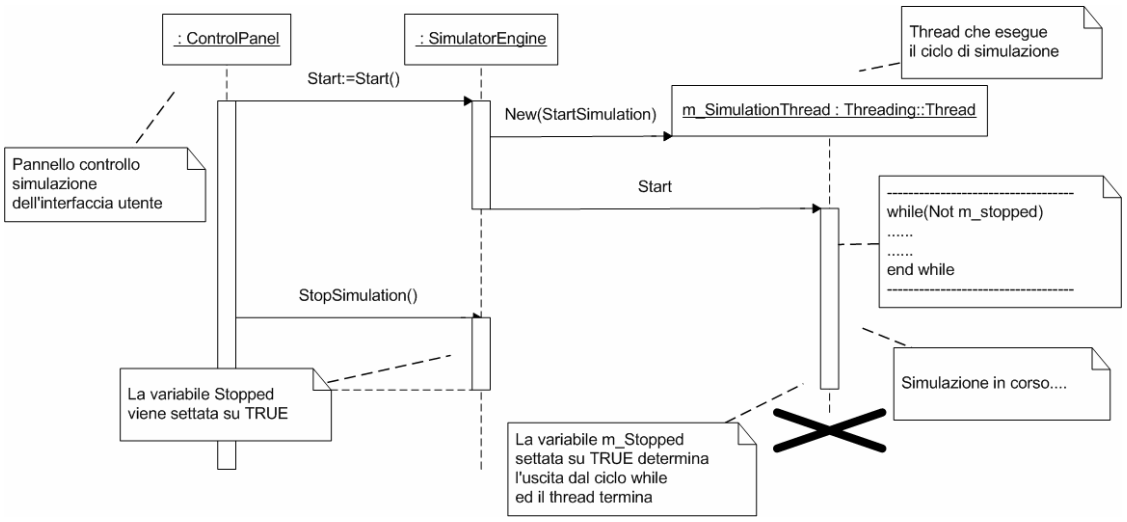


Figura 47. Sequence diagram 3

Per ogni ciclo di esecuzione, come nel ciclo di inizializzazione e di reset, gli oggetti utilizzati vengono acquisiti in modo esclusivo. Nel sequence diagram della pagina seguente relativo alla chiamata del metodo *ExecuteSimulationInit()* si nota l'arco di tempo in cui la risorsa rimane bloccata ad uso esclusivo del thread che esegue il metodo.

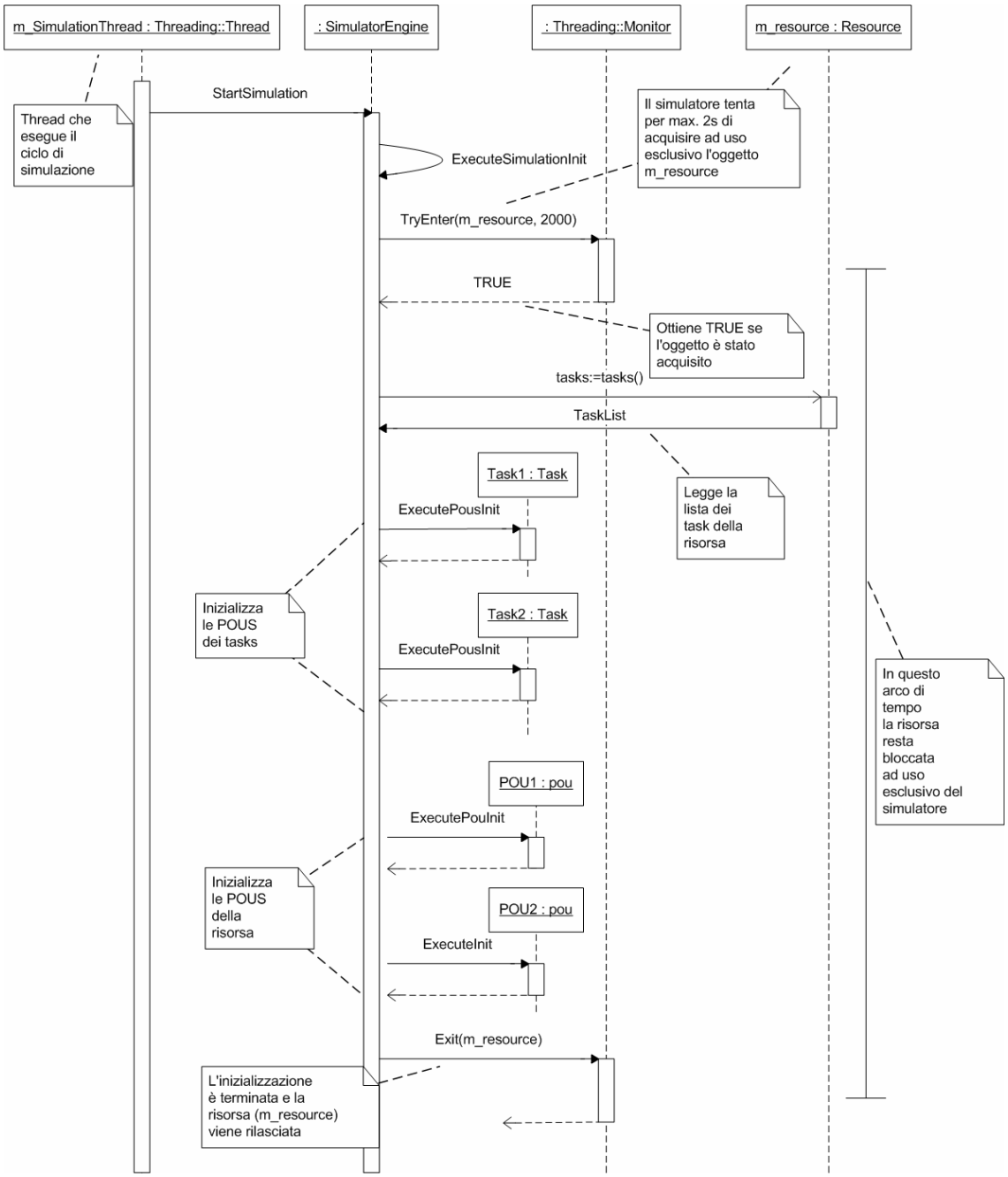


Figura 48. Sequence diagram 4

Il codice del metodo *ExecuteSimulation()* è riportto in seguito con i relativi commenti.

```
1   Private Sub ExecuteSimulation()
2
3       'Ciclo di simulazione
4       While Not (m_Stopped)
5           Try
6               'Seleziona tra l'algoritmo Preemptive e NotPreemptive
```

```
7            If m_PreemptiveScheduling Then
8               If Monitor.TryEnter(m_ resource.tasks, 100) Then
9                  If Monitor.TryEnter(m_resource, 100) Then
10                    ExecutePreemptiveSimulationScanCycle()
11                 End If
12                 Monitor.Exit(m_resource)
13              End If
14              Monitor.Exit(m_resource.pouList)
15           Else
16              If Monitor.TryEnter(m_resource.pouList, 100) Then
17                 If Monitor.TryEnter(m_resource, 100) Then
18                    'Controlla se sono terminati i thread dell'algoritmo preemptive
19                    If Not m_TaskThreadsKilled Then
20                       KillsTaskThreads()
21                    End If
22                    ExecuteNotPreemptiveSimulationScanCycle()
23                    Monitor.Exit(m_resource)
24                 End If
25                 Monitor.Exit(m_resource.pouList)
26              End If
27           End If
28           m_Cycles = m_Cycles + 1
29           If m_Wait > 0 Then
30              m_SimulationThread.SpinWait(m_Wait)
31           End If
32        Catch ex As System.Exception
33           Monitor.Exit(m_resource)
34           Monitor.Exit(m_resource.tasks)
35        End Try
36     End While
37     m_Stopped = False
38  End Sub
```

Il ciclo while che inizia alla riga 4 è il ciclo principale. Si arresta quando la variabile booleana *m_stopped* assume valore *True*. Il blocco *try* che inizia alla riga successiva gestisce eventuali eccezioni intercettabili a livello di sistema [16,17]. Il suo scopo è quello di evitare che, in caso di un eventuale blocco del thread che esegue il ciclo di simulazione, le risorse acquisite ad uso esclusivo non restino bloccate indefinitivamente. Infatti in caso di eccezione l'esecuzione riprende dall'istruzione *Catch ex As System.Exception* (riga 32) e successivamente vengono rilasciate le risorse occupate (righe 33-34). L'acquisizione ad uso esclusivo degli oggetti *m_resource* (riga 8) ed *resource.tasks* (lista dei task della risorsa, riga 9) è necessaria per garantire che durante un singolo ciclo di esecuzione non vengano modificati dall'utente. Gli oggetti sono modificabili tra la fine di un ciclo e l'inizio di un altro.
All'inizio di ogni ciclo viene selezionato il tipo di multitasking richiesto (preemptive o meno, riga 7). Anche questa scelta è modificabile tra due cicli di esecuzione. In questo modo è possibile passare da un tipo di multitasking all'altro anche in corso di

simulazione. I due rami dell'istruzione *if then else* dunque chiamano rispettivamente i metodi *ExecutePreemptiveSimulationScanCycle* e *ExecuteNotPreemptive-SimulationScanCycle*. Nel secondo caso viene effettuato un controllo per determinare se gli eventuali thread associati ai task ed utilizzati dal multitasking preemptive sono terminati. Ciò è necessario nel momento in cui si passa in corso di simulazione dal multitasking preemptive a quello non preemptive. Il metodo *KillsTaskThreads()* riportato in seguito termina i suddetti thread.

```
Private Sub KillsTaskThreads()
    'Termina gli eventuali thread in sospeso dell'algoritmo preemptive
    For Each T As Task In m_resource.tasks
        T.ResumeThread()
        T.Join()
    Next T
    m_ResoursePouInstanceTask.ResumeThread()
    m_ResoursePouInstanceTask.Join()
    m_TaskThreadsKilled = True
End Sub
```

Per terminare i thread il metodo 'risveglia' tutti i thread in sospeso (istruzioni *ResumeThread()*, e mette in attesa il thread principale del termina di ognuno (istruzioni *Join()*).

La schedulazione dei task

Il metodo ScheduleTask() restituisce il task da eseguire selezionandolo dalla lista dei task. L'implementazione è la seguente:

```
1   Private Function ScheduleTask() As Task
2       'Seleziona il task da eseguire
3       Try
4           Dim LastTaskPrioriry As Integer = Integer.MaxValue
5           If Monitor.TryEnter(m_taskList, 500) Then
6               'Memorizza l'istante di tempo attuale
7               m_NowTime = Now
8               For Each T As Task In m_taskList
9                'Cerca se ci sono task pronti o sospesi seleziona 10
                  'quello con priorità più alta(0 è la massima)
11                    If (m_NowTime.Subtract    _(T.m_LastActivation).TotalMilliseconds >_
                                    _T.m_interval.TotalMilliseconds And T.Ready)_ Or_
                                    _T.ExecutionThread.IsAlive Then
12                        If T.priority < LastTaskPrioriry Then
13                            ScheduleTask = T
14                            LastTaskPrioriry = T.priority
15                        End If
16                    End If
17                Next T
```

```
18              Monitor.Exit(m_taskList)
19          End If
20      Catch ex As System.Exception
21          Monitor.Exit(m_taskList)
22      End Try
23  End Function
```

Dopo aver bloccato la lista dei tasks (riga 5) memorizza l'istante di tempo leggendolo dal timer di sistema (riga 7). Successivamente per ogni task in lista (ciclo for della riga 8) verifica (riga 10) se risulta terminato l'intervallo ciclico di esecuzione del task e se il task risulta pronto (attributo *m_ready*) oppure se è stato precedentemente sospeso (istruzione *T.ExecutionThread.IsAlive*); questa situazione si può verificare solo nel momento in cui si stia eseguendo un multitasking preemptive). In caso una delle due condizioni sia verificata confronta (dalla riga 12) se il task dispone di una priorità più alta dei task che hanno superato il test della riga 10 in precedenza. In caso affermativo viene memorizzato il riferimento al task e viene memorizzata la relativa priorità per i successivi confronti (righe 13 e 14). Al termine del ciclo for (termina a riga 17) rilascia la lista dei task. Potrebbe verificarsi il caso in cui nessun task disponga delle condizioni per essere eseguito. In questo caso il metodo restituisce un riferimento nullo (*nothing*).

Implementazione del multitasking preemptive

Il multitasking preemptive viene gestito dal metodo *ExecutePreemptive-SimulationScanCycle()* riportato in seguito. Il metodo viene chiamato ciclicamente dal metodo *ExecuteSimulation()*. L'esecuzione dei task viene effettuata all'interno dei relativi thread (definiti negli oggetti *task*), che vengono sospesi e successivamente risvegliati, mediante le chiamate ai metodi *suspsend()* e *run()* della classe *task*, per concedere l'esecuzione ad un eventuale task a priorità maggiore. Il thread che esegue il metodo *ExecutePreemptiveSimulationScanCycle()*, mentre viene eseguito un task, chiama ciclicamente il metodo *ScheduleTask()* per verificare l'eventuale presenza di un task pronto con priorità maggiore di quello in esecuzione. In caso positivo sospende il thread del task in esecuzione ed avvia quello del task a priorità maggiore.

```
1   Private Sub ExecutePreemptiveSimulationScanCycle()
2
3       Dim TaskToExecute As Task
4       'Seleziona il task da eseguire
5       TaskToExecute = ScheduleTask()
6       'Se ottiene un task
7       If Not IsNothing(TaskToExecute) Then
8           'Esegue il task se è a priorità maggiore di m_TaskExecuted
9           If TaskToExecute.priority < m_TaskExecutedPrioriry Or Not_
10                  _m_TaskExecuted.ExecutionThread.IsAlive Then
11              If m_TaskExecuted.ExecutionThread.IsAlive Then
12                  m_TaskExecuted.Suspend()
13              End If
14              'Esegue il task selezionato
15              TaskToExecute.Run()
16              'Memorizza il task in esecuzione
17              m_TaskExecuted = TaskToExecute
18              m_TaskExecutedPrioriry = m_TaskExecuted.priority
19          End If
20      Else
21          'Se non riceve nessun task e m_TaskExecuted è terminato esegue i pou senza
task
22          TaskToExecute = m_ResoursePouInstanceTask
23          'Esegue il task
24          TaskToExecute.Run()
25      End If
25  End Sub
```

Alla riga 5 viene selezionato il task da eseguire. Se non viene restituito un riferimento nullo verifica se il task ottenuto dispone di una priorità maggiore di quello in esecuzione oppure se l'ultimo task eseguito è terminato (dalla riga 9). In caso affermativo se l'ultimo task da eseguire non è terminato lo sospende (riga 12). Quindi esegue il task (riga 15) e ne *memorizza un* riferimento e la relativa priorità (righe 17 e 18). Se dal metodo *ScheduleTask()* non riceve alcun task (nessun task richiede di essere eseguito) allora esegue il task relativo alle istanze delle POU prive di task nella risorsa (riga 22 e 24).

Questo tipo di multitasking naturalmente soffre di carico elaborativo relativamente elevato: due thread in esecuzione, elevata frequenza di schedulazione dei task e cambi di contesto.

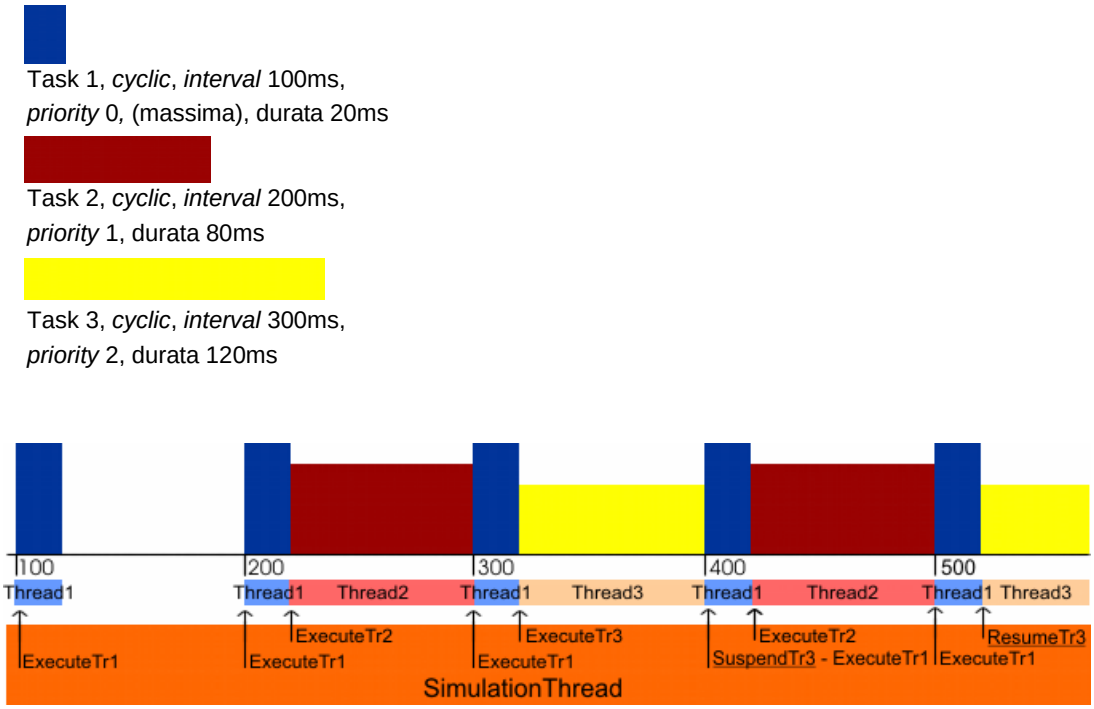


Figura 49. Implementazione del multitasking preemptive

Implementazione del multitasking non preemptive

L'algoritmo che implementa il multitasking non preempitive è più semplice. Esso non utilizza thread di ogni task, ma l'esecuzione è interamente affidata al thread principale di simulazione. La selezione del task da eseguire viene effettuata solo al termine dell'esecuzione di ogni singolo task, quindi viene eseguito il task selezionato ed il ciclo riprende. Il codice è il seguente:

```
Private Sub ExecuteNotPreemptiveSimulationScanCycle()
    Dim TaskToExecute As Task
    'Seleziona il task da eseguire
    TaskToExecute = ScheduleTask()
    'Esegue il task selezionato
    If Not IsNothing(TaskToExecute) Then
        TaskToExecute.ExecutePous()
    Else
        'Se non lo trova esegue le istanze dei pou senza task
        For Each P As pouInstance In m_resource.pouInstances
            P.pou.Execute()
        Next P
    End If
End Sub
```

Naturalmente in questo modo un task pronto con priorità più alta di quello in esecuzione deve attendere il termine dell'esecuzione di quest'ultimo.

Rispetto al multitasking preemptive il carico elaborativo è inferiore: in esecuzione vi è un unico thread, la selezione dei task da eseguire avviene con frequenza molto più bassa e non deve essere effettuato nessun cambio di contesto.

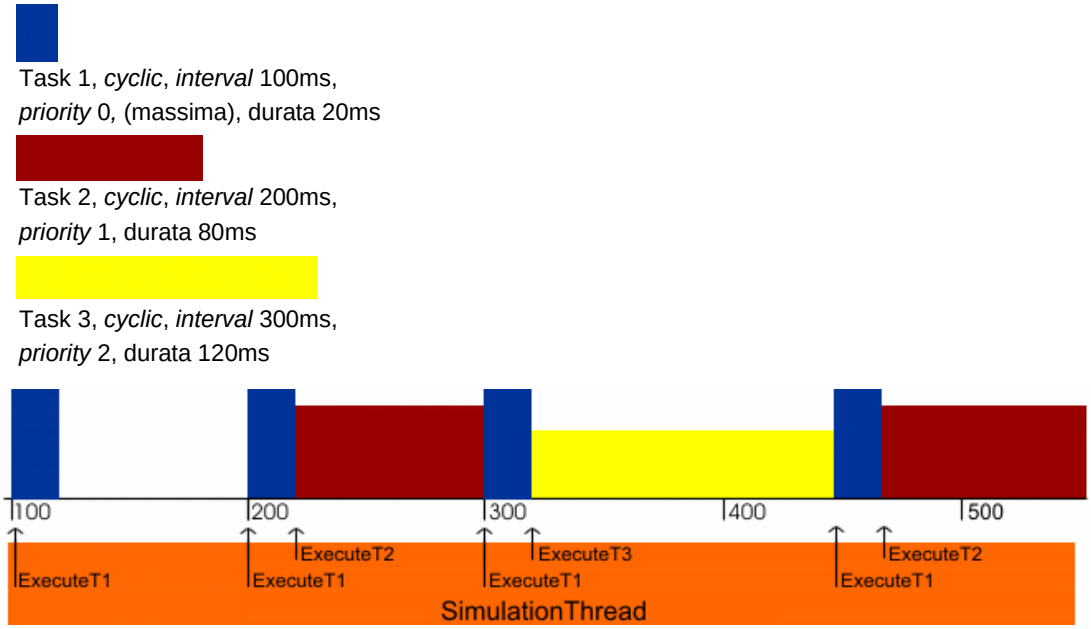


Figura 50. Implementazione del multitasking non preemptive

Esecuzione dei task

I cicli di esecuzione dei task sono effettuati attraverso le chiamate al metodo *ExecutePous()*. Quest'ultimo, nel caso del multitasking preemptive viene chiamato dall'interno del task stesso dal relativo thread. Si susseguono una successione di chiamate ai metodi di esecuzione delle singole unità di programma del task. Per ognuna di esse l'esecuzione prosegue attraverso le chiamate ai metodi illustrati nella parte relativa alle POU di questo capitolo. Ciò è illustrato nel seguente sequence diagram.

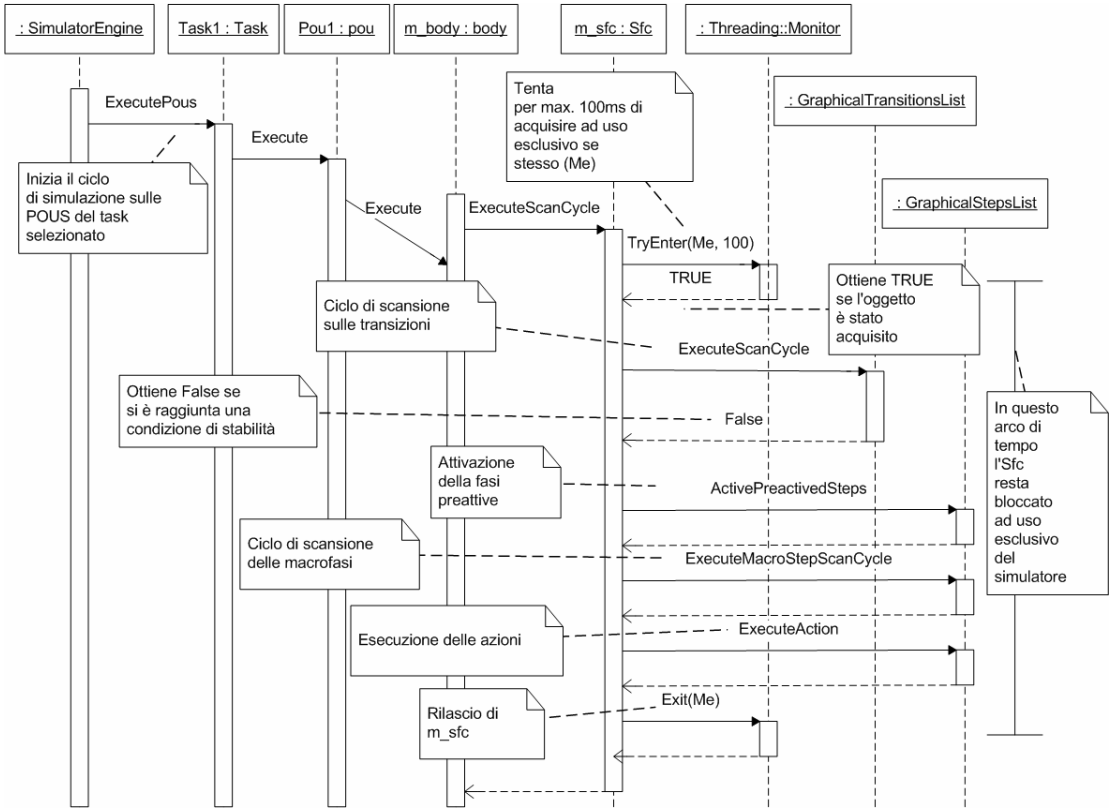


Figura 51. Sequence diagram 5

Gestione delle risorse condivise

Quando un programma utilizza una risorsa condivisa tenta di accedervi ad uso esclusivo. In questo caso le uniche risorse condivise da più programmi in esecuzione sono gli oggetti che rappresentano le variabili globali definite a livello di risorsa. In questa situazione, quando in simulazione viene eseguito il multitasking preemptive, si può verificare una sospensione di un task che ha acquisito ad uso esclusivo una variabile. Se il task in esecuzione, con priorità maggiore del task sospeso, cerca di acquisire la variabile si blocca. Questa situazione si potrebbe verificare anche in modo da determinare più task sospesi ed in attesa dell'oggetto condiviso. Il problema viene gestito mediante una tabella di allocazione delle risorse (classe *ResourceTable*). Ogni volta che un programma acquisisce ad uso esclusivo un oggetto, viene memorizzato nella tabella un riferimento all'oggetto ed al thread responsabile dell'esecuzione del task che contiene il programma. Quando un task resta bloccato in attesa che un oggetto si liberi genera un evento che viene intercettato da un istanza della classe ResourceTable. Questo oggetto interroga la

tabella ed individua il thread sospeso responsabile del blocco. Quindi genera un evento intercettato dal metodo *EventVarLocked* della classe *EngineSimulator*. Insieme all'evento viene comunicato un riferimento all'oggetto bloccato ed un riferimento al thread relativo al task in attesa. Il suddetto metodo effettua una sorta di *inversione di priorità*: mentre il task a priorità maggiore risulta bloccato, risveglia il thread a priorità minore relativo al task sospeso ed attende che termini. In questo modo viene sbloccato l'oggetto condiviso ed il task a priorità maggiore riprende l'esecuzione. Il metodo *EventVarLocked* è il seguente:

```
Private Sub EventVarLocked(ByRef T As Thread) Handles m_ResourceTab.VarLocked
   Try
      If T.IsAlive Then
         T.Resume()
         T.Join()
      End If
   Catch ex As System.Exception
   End Try
End Sub
```

L'istruzione *T.Resume* risveglia il thread che esegue il task responsabile del blocco e l'istruzione *T.Join()* attende che termini.

3.4 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* [16,17]). 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
Il modulo di controllo

Il tool consente di eseguire il controllo di processi reali. Esso dispone di un interfaccia verso dispositivi di input/output che possono essere collegati con i sensori e gli attuatori di un impianto di automazione. L'evoluzione dello stato dei programmi, dei valori delle variabili, dell'esecuzione dei task, ecc. possono essere monitorati con gli stessi strumenti grafici come per la simulazione. Allo stato attuale SiValPro, operando solo su variabili di tipo booleane, si interfaccia con dispositivi forniti di ingressi e uscite di tipo digitale. Attraverso l'utilizzo di questo modulo è possibile testare direttamente il software prodotto sul sistema reale. Ciò consente di utilizzare il tool anche per la generazione rapida di prototipi.

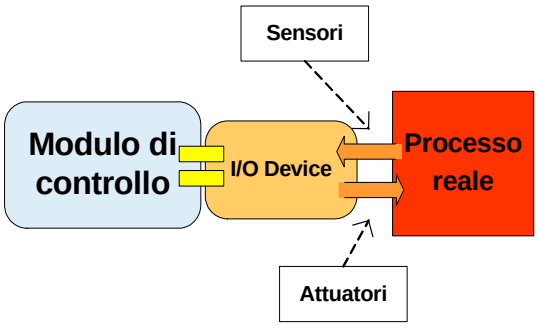


Figura 52. Controllo di un processo

È necessario considerare che il modulo, essendo stato progettato per essere eseguito su sistemi non real-time, può essere utilizzato per il controllo di processi non critici

dal punto di vista delle specifiche temporali. Le prestazioni a run-time dell'algoritmo di controllo infatti sono influenzati da diversi fattori dipendenti dal sistema operativo, in particolare nei tempi di risposta. Ciò dunque ne consente l'applicazione nel caso di sistemi soft real-time[6] [9,20].

4.1 Architettura e caratteristiche del modulo di controllo

L'approccio object-oriented seguito per lo sviluppo del tool ha consentito di produrre codice riusabile. Il modulo di controllo infatti utilizza tutte le classi prodotte appartenenti al *ProjectGroup* e parte delle classi dell'*InteractionGroup*. In più sono state appositamente realizzate le classi *ControlEngine*, *IOInterface* e *BaseIODriver* e sono state aggiunte al *PLCGroup*.

Il modulo e la relativa interfaccia grafica costituiscono un'applicazione indipendente. I progetti sono importati dal formato xml[7] sempre previa validazione del parser secondo le regole definite dal relativo schema. Dopo la validazione viene creata la struttura degli oggetti definiti dagli elementi contenuti nel progetto da importare (programmi, istanze, task, variabili, ecc.) con i rispettivi attributi.

Dalle valutazioni delle prestazioni mostrate a run-time dal modulo[8] si è rilevato un degrado dei tempi di risposta proporzionale al numero di thread in esecuzione. In particolare si è mostrata netta la differenza determinata dalla presenza di un solo thread in esecuzione rispetto a due. Ciò ha determinato la scelta di non consentire in fase di controllo l'utilizzo del multitasking preemptive come in fase di simulazione, in quanto realizzato mediante l'utilizzo di più thread.

Implementazione delle funzioni di controllo

Ogni risorsa, oltre contenere un'istanza della classe *SimulatorEngine*, ne contiene una della classe *ControlEngine*.

[6] Un sistema è detto hard real-time se l'eventuale perdita di un dato determina il fallimento dell'algoritmo che ne effettua il controllo. Per i sistemi soft real-time un'eventuale perdita non ne compromette definitivamente l'esito, ma il controllo può essere immediatamente recuperato acquisendo le informazioni successive [9,20].
[7] Si veda Capitolo 2

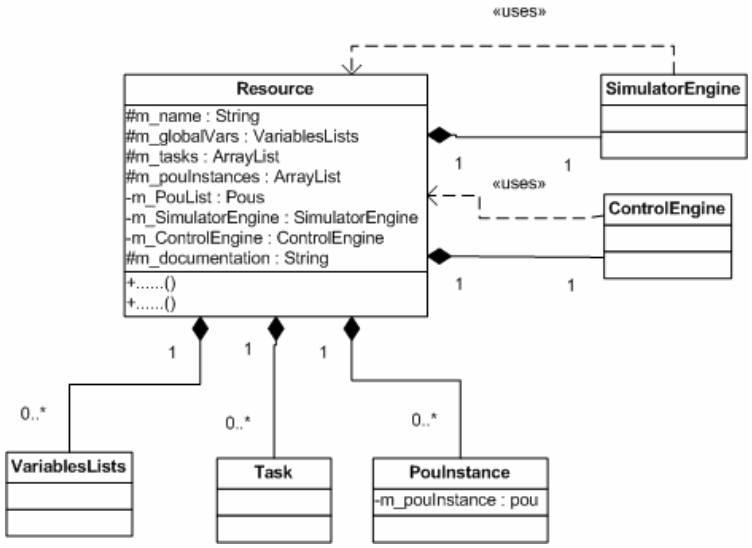


Figura 53. Class diagram 12

Questa classe implementa le funzioni per la gestione dell'esecuzione del controllo del processo reale. L'algoritmo ciclico di controllo è lo stesso utilizzato nella simulazione, con l'aggiunta delle chiamate ai metodi per la comunicazione con in dispositivi di I/O. I metodi *Start()* e *StopControl()* determinano rispettivamente l'avvio e l'arresto del processo di controllo. Il metodo *Reset()* resetta lo stato dei programmi e delle variabili. La classe *ControlEngine* dispone di una lista dinamica atta a memorizzare gli eventuali errori che si verificano a run-time internamente al ciclo di controllo. I tipi di errori sono definiti dalla seguente enumerazione:

```
Public Enum Errors
   NothingIOInterface
   ReadError
   WriteError
   InternalCycleError
   InitError
   ResettingErrors
   BuiltingErrors
End Enum
```

I membri dell'enumerazione vanno così interpretati:

NothingIOInterface	identifica un errore relativo alla creazione di un istanza dell'interfaccia;
ReadError	identifica un errore in fase di lettura dei dati dagli ingressi;

[8] La valutazione è stata effettuata misurando il tempo necessario ad effettuare un determinato numero di cicli in

WriteError	identifica un errore in fase di scrittura dei dati sulle uscite;
InternalCycleError	identifica un errore dovuto ad un'eccezione intercettata dal blocco *try…catch* interno al ciclo di controllo;
InitError	identifica un errore verificatosi durante l'inizializzazione e che ne ha impedito il completamento;
ResettingErrors	identifica un errore verificatosi durante l'operazione di reset e che ne ha impedito il completamento;
BuiltingErrors	identifica un errore in fase di creazione delle istanze dei programmi.

In seguito saranno descritte le soluzioni adottate per la comunicazione con i dispositivi di I/O. Successivamente, alla luce di quanto sarà detto, si riprenderà la descrizione dell'algoritmo di controllo.

Interfaccia verso i dispositivi di I/O

La classe *ControlEngine* contiene l'oggetto *m_IOInterface*. Esso costituisce un istanza della classe *IOInterface*, che implementa l'interfaccia di comunicazione verso i dispositivi di I/O. Questa classe utilizza un'istanza di un driver specifico di dispositivo. Un oggetto di tipo *IOInterface* riceve un riferimento alla collezione di liste delle variabili utilizzate dai programmi in esecuzione ed attraverso l'attributo *m_address* (di tipo *string*) effettua il mapping delle variabili con gli ingressi e le uscite fisiche. Come indicato dallo standard per far riferimento alle locazioni di memoria destinate all'input/output[2,5], l'assegnazione avviene dapprima in base al primo carattere dell'attributo *m_address*: il carattere 'I' referenzia un ingresso, mentre il carattere 'O' referenzia un'uscita. I caratteri successivi sono di tipo numerico ed indicano la locazione di memoria di ingresso o di uscita. I valori dipendono dall'architettura dello specifico dispositivo. La convenzione adottata in questo luogo identifica con i caratteri precedenti al punto(.) il numero del canale e con quelli successivi il relativo bit. Un canale è composto da 8 bit[9]. Le associazioni vengono memorizzate attraverso due array la cui dimensione può essere variata anche a run-time [16]. Le strutture utilizzate sono le seguenti.

```
Private Structure LinkVariableLine
```

diverse condizioni.

[9] Ad esempio l'indirizzo 'I2.4' identifica il quarto bit del secondo canale di ingresso.

```
    Dim Var As BaseVariable
    Dim LineNumber As Integer
End Structure
'Liste che contengono i riferimenti diretti alle variabili e la linea di I o O
Private m_DigitalInputVariablesList() As LinkVariableLine
Private m_DigitalOutputVariablesList() As LinkVariableLine
```

La struttura *LinkVariableLine* rappresenta un record che associa un riferimento ad un oggetto variabile ad una linea di ingresso o di uscita[10]. Le associazioni relative alle linee di input sono memorizzate nell'array *m_DigitalInputVariablesList()*, mentre quelle relative alle linee di output nell'array *m_DigitalOutputVariablesList()*.

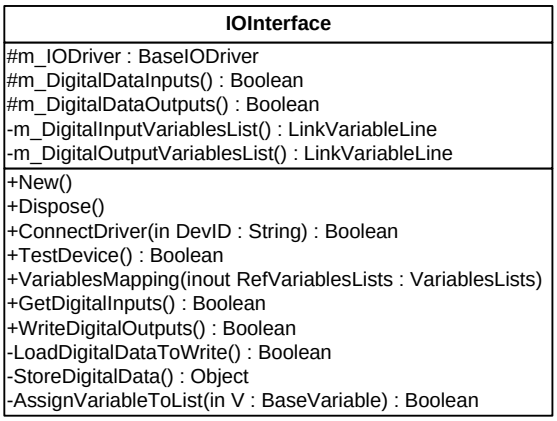


Figura 54. La classe *IOInterface*

Le operazioni di lettura e scrittura utilizzano gli array di variabili booleane *m_DigitalDataInputs()* e *m_DigitalDataOutputs()*. La chiamata al metodo *GetDigitalInputs()* determina la lettura degli ingressi. La lettura viene effettuata mediante la chiamata al metodo *GetDigitalInputs* dell'istanza di un driver (*m_IODriver*), passando come parametro il riferimento all'array in cui memorizzare i valori. Successivamente con il metodo *StoreDigitalData()* i valori vengono copiati nelle variabili risolvendo le associazioni mediante l'array di record *m_DigitalInputVariablesList()*. L'implementazione dei due metodi è la seguente:

```
Public Function GetDigitalInputs() As Boolean
    'Legge gli input attraverso il driver
    Try
```

[10] Le linee di ingresso o di uscita sono numerate in sequenza crescente a partire da 0. Il valore numerico contenuto nell'indirizzo corrisponde al numero di linea secondo la seguente uguaglianza: NL = P*7+S, dove NL è il numero della linea, P il valore numerico nell'indirizzo precedente al punto ed S quello successivo

```
        m_IODriver.GetDigitalInputs(m_DigitalDataInputs)
        'Aggiorna le variabili
        StoreDigitalData()
    Catch ex As System.Exception
        MsgBox(ex.Message)
    End Try
End Function


Private Function StoreDigitalData()
    'Aggiorna le variabili di input
    Dim i As Integer
      'Per ogni elemento in m_DigitalInputVariablesList legge il valore da assegnare alla
      variabile dal numero di linea specificato in LineNumber
      For i = 0 To m_DigitalInputVariablesList.GetLength(0) - 1
                m_DigitalInputVariablesList(i).Var.SetValue(m_DigitalDataInputs_
                _(m_DigitalInputVariablesList(i).LineNumber))
      Next i
End Function
```

In modo analogo viene eseguita la scritture delle uscite. I metodi utilizzati sono i seguenti:

```
Public Function WriteDigitalOutputs() As Boolean
    'Aggiorna la matrice di bit m_DigitalDataOutputs
    LoadDigitalDataToWrite()
    'Scrive gli output attraverso il driver
    Try
        m_IODriver.WriteDigitalOutputs(m_DigitalDataOutputs)
    Catch ex As System.Exception
        MsgBox(ex.Message)
    End Try
End Function


Private Function LoadDigitalDataToWrite() As Boolean
    'Riempie la matrice di bit m_DigitalDataOutputs leggendo i valori delle variabili di output
    Dim i As Integer
      'Per ogni elemento in m_DigitalOutputVariablesList assegna il valore della variabile al
      bit specificato in LineNumber
    For i = 0 To m_DigitalOutputVariablesList.GetLength(0) - 1
        m_DigitalDataOutputs(m_DigitalOutputVariablesList(i).LineNumber) =_
      _ -m_DigitalOutputVariablesList(i).Var.ReadValue
    Next i
End Function
```

L'oggetto *m_IODriver* utilizzato costituisce un riferimento ad un istanza di una classe che implementa un driver per una specifica periferica. *m_IODriver* è dichiarato come tipo di riferimento della classe astratta *BaseIODriver*. Una classe che implementa un driver per uno specifico dispositivo deve ereditare tale classe e ridefinirne i metodi virtuali per la lettura, la scrittura ed il test. A run-time, all'atto

della creazione di un'istanza del driver, quest'ultima viene referenziata da *m_IODriver*, utilizzato dalla classe *IOInterface*. In tal modo, poiché l'oggetto driver referenziato da *m_IODriver* dispone dei metodi virtuali ridefiniti della classe *BaseIODriver*, l'interfaccia implementata dalla classe *IOInterface* resta indipendente da uno specifico driver.

Per effettuare i test del tool è stato implementato un driver per schede di I/O di tipo digitale della National Instruments [21,22,23]. Nel seguente class diagram si nota la ridefinizione dei metodi virtuali della classe *BaseIODriver*.

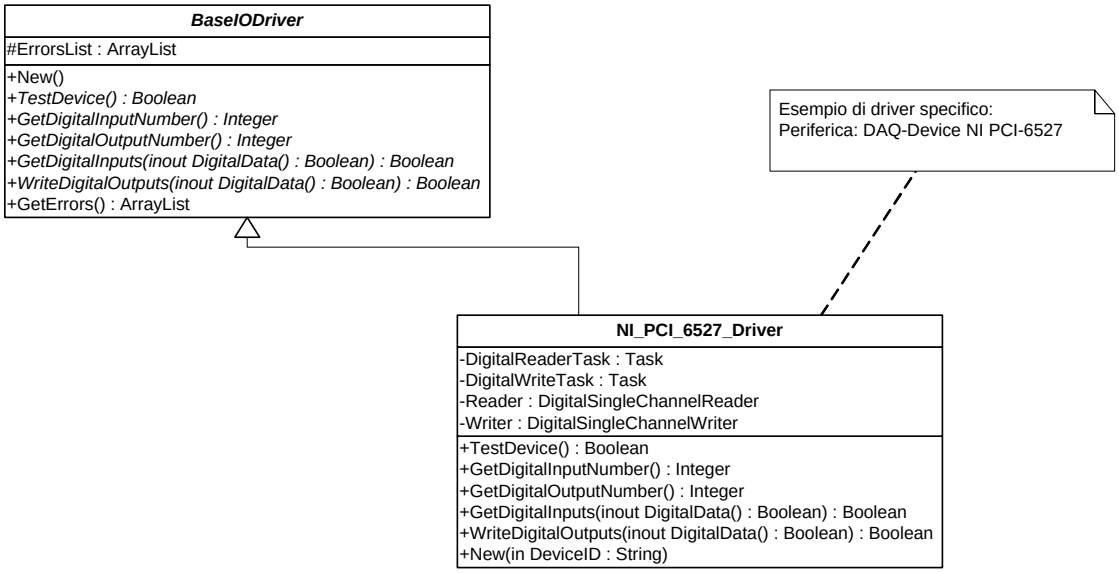


Figura 55. Class diagram 13

L'algoritmo di controllo

Il seguente class diagram mostra le relazioni esistenti tra le classi del modulo di controllo menzionate in questo capitolo.

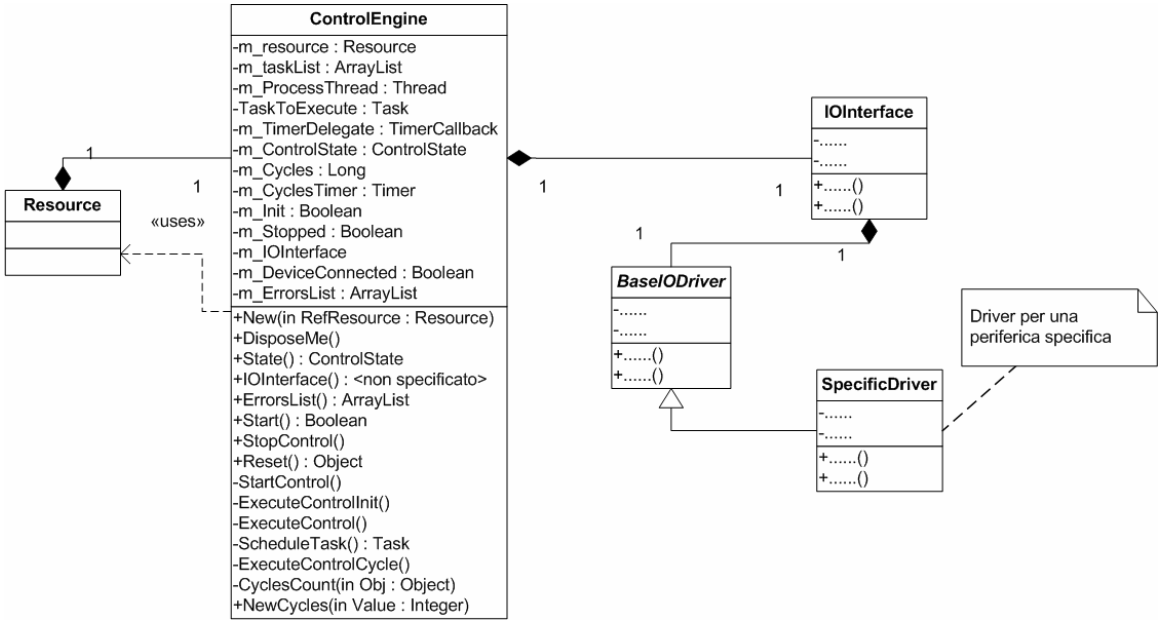


Figura 56. Class diagram 14

L'esecuzione del controllo inizia con la chiamata al metodo *Start()*. Esso inizializza dapprima le variabili di supporto, quindi crea un thread che esegue l'algoritmo di controllo ed un thread che gestisce un timer per il conteggio dei cicli. Successivamente avvia l'esecuzione di entrambi i thread. Il primo thread esegue il metodo *StartControl()* che effettua l'inizializzazione e successivamente determina l'avvio del ciclo di controllo attraverso la chiamata al metodo *ExecuteControl()*. L'implementazione è la seguente:

```
1   Private Sub ExecuteControl()
2     'Ciclo di controllo
3     While Not (m_Stopped)
4       Try
5         'Legge i dati dagli ingressi fisici
6         If Not m_IOInterface.GetDigitalInputs() Then
7           'Aggiunge l'errore alla lista
8           m_ErrorsList.Add(Errors.ReadError)
9         End If
10          'Ciclo di esecuzione
11          ExecuteControlCycle()
12          'Scrive i dati sugli ingressi fisici
13          If Not m_IOInterface.WriteDigitalOutputs() Then
14            'Aggiunge l'errore alla lista
15            m_ErrorsList.Add(Errors.WriteError)
16          End If
17          m_Cycles = m_Cycles + 1
18        Catch ex As System.Exception
19          'Aggiunge l'errore alla lista
```

```
20              m_ErrorsList.Add(Errors.InternalCycleError)
21          End Try
22      End While
23  End Sub
```

Il ciclo *while* (riga 3) continua fin che la variabile *m_Stopped* non viene impostata su *false* dal metodo *StopControl(),* successivamente ad una richiesta da parte dell'utente. Per ogni ciclo vengono effettuate le seguenti operazioni:

- lettura degli ingressi del dispositivo fisico (riga 6);
- esecuzione di un singolo ciclo di controllo (riga 11);
- scrittura degli ingressi del dispositivo fisico(riga 13).

Se una delle operazioni di lettura o scrittura fallisce i metodi corrispondenti (*IOInterface.GetDigitalInputs()* e *m_IOInterface.WriteDigitalOutputs()*) restituiscono il valore *False*. L'evento viene registrato aggiungendo la descrizione dell'errore nella lista *m_ErrorsList*. La lista può essere consultata attraverso la proprietà in sola lettura *ErrorsList*[11].

L'esecuzione di ogni singolo ciclo di controllo viene effettuata dal metodo *ExecuteControlCycle()*. L'implementazione è simile al quella del metodo *ExecuteNotPreemptiveSimulationScanCycle()* relativa al ciclo di simulazione con multitasking non preemptive, come si nota dal codice riportato in seguito.

```
1   Private Sub ExecuteControlCycle()
2       'Seleziona il task da eseguire
3       TaskToExecute = ScheduleTask()
4       'Esegue il task selezionato
5       If Not IsNothing(TaskToExecute) Then
6           TaskToExecute.ExecutePousInstance()
7       Else
8           'Se non lo trova esegue i pou senza task
9           For Each P As pouInstance In m_resource.pouInstances
10              P.ExecutePouInstance()
11          Next P
12      End If
13  End Sub
```

La differenza fondamentale rispetto al metodo menzionato sta nelle chiamate ai metodi *ExecutePousInstance()* (righe 6 e 10) piuttosto che ai metodi *ExecutePous()* nel caso della simulazione. Il metodo *ExecutePousInstance()* esegue le istanze delle POU, mentre il metodo *ExecutePous()* esegue direttamente le POU create dall'utente. In fase di simulazione l'esecuzione delle POU create dall'utente è necessaria per permetterne contestualmente la modifica. Il modulo di controllo, non

[11] Per una descrizione relativa all'utilizzo delle proprietà in ambiente .Net Framework si veda [15,16,17]

essendo prevista la modifica dei programmi, esegue le relative istanze.

Le istanze vengono create attraverso il metodo *built()* della classe *resource*, di cui si riporta il codice:

```
Public Function Built() As Boolean
   'Genera le instanze dei pou
   Built = True
   Try
      'Pou con task
      For Each T As Task In m_tasks
         For Each PI As pouInstance In T.pouInstances
            PI.CreateInstance()
         Next PI
      Next T
      'Pou senza task
      For Each PI As pouInstance In m_pouInstances
         PI.CreateInstance()
      Next pi
   Catch ex As System.Exception
      Built = False
   End Try
End Function
```

Si notano le chiamate ai metodi *CreateInstance()* per ogni oggetto *pouInstace* di ogni task e per ogni oggetto *pouInstace* della risorsa (quelli privi di task). Tali metodi realizzano la creazione di una copia degli oggetti *m_pou* contenuta in ogni oggetto *pouInstace*, restituendone un riferimento che viene assegnato al tipo di riferimento *m_pouIstance*. Il codice del metodo *CreateInstance()* è il seguente:

```
Public Sub CreateInstance()
   'Crea l'istanza del POU
   m_pouInstance = m_pou.CreateInstance
End Sub
```

La creazione delle istanze delle POU è stata descritta in dettaglio nel capitolo 3, nel paragrafo "*Le istanze delle unità organizzative di programma*".

Per concludere si può notare l'assenza degli oggetti *monitor* utilizzati invece dalle funzioni di simulazione per ottenere ad uso esclusivo l'utilizzo della risorsa e della lista delle istanze delle POU. Ciò in quanto il modulo di controllo non dispone di funzioni di editing, quindi non vi è nessuna condivisione dei suddetti oggetti. Questa assenza determina un miglioramento delle prestazioni nell'esecuzione dell'algoritmo di controllo.

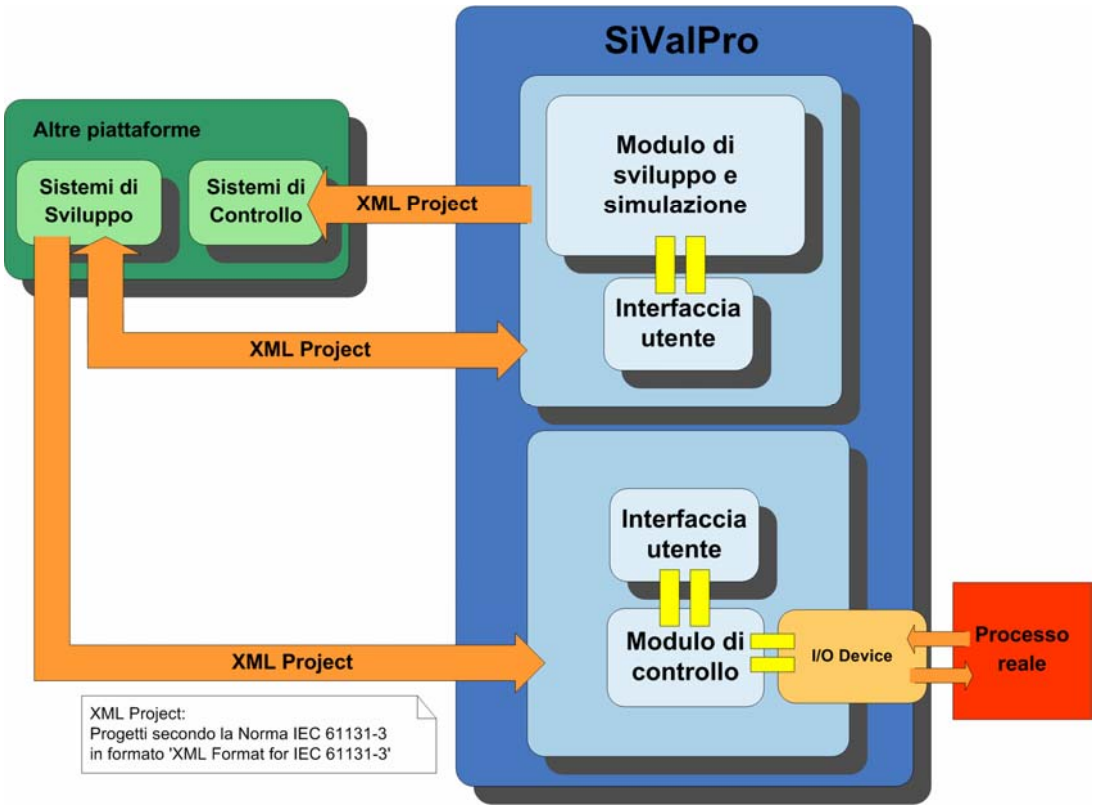
Conclusioni e sviluppi futuri

SiValPro è un tool di sviluppo per progetti d'automazione secondo le direttive dettate dallo Standard 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 consente il riuso del software prodotto in caso di migrazione tra sistemi di case produttrici diverse.

Il seguente diagramma rappresenta il possibile inserimento del tool nel contesto dei sistemi sviluppo di software d'automazione.



Il formato di interscambio adottato è stato rilasciato ufficialmente il 28 aprile 2005. Attualmente il tool risulta il primo applicativo esistente ad averlo adottato. Una volta che tale formato sia stato recepito dalle case produttrici di controllori a logica programmabile, tutto il software prodotto con SiValPro potrà essere utilizzato dai vari sistemi di sviluppo proprietari, per poi poter essere eseguito sulle varie piattaforme commerciali. Analogamente un progetto d'automazione prodotto con un sistema di sviluppo proprietario potrà essere validato utilizzando il simulatore e/o il modulo di controllo di SiValPro.

La versione attuale del tool presenta alcune limitazioni, tra cui:

- 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.

Alcune scelte come l'approccio object-oriented, la progettazione di un'architettura indipendente da uno specifico linguaggio dello Standard e l'attenzione alla modularità del software indirizzano gli sviluppi futuri verso il superamento di queste limitazioni.

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] R. David, "GRAFCET: a powerful tool for specification of logic controllers", IEEE Transaction on Control Systems Technology, vol. 3, n. 3, 1995.

[7] Liu J. W. S., "Real Time Systems", Prentice Hall, 2000.

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

[9] http://www.w3c.org/1999/xhtml

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

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

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

[13] http://msdn.microsoft.com/

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

[15] Microsoft, 2003, "Visual Studio 2003 – Manuale dello sviluppatore"

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

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

[18] P. Ancilotti.; M.Boari , 1997, "Principi di programmazione concorrente" UTET Libreria

[19] P. Ancilotti.; M.Boari , 2004, "Sistemi Operativi'", McGraw-Hill

[20] http://www.embedded.it/

[21] National Instruments, 2004, "NI-DAQ™mx Help"

[22] National Instruments, 2001, "NI-DAQ™ - Release Note"

[23] http://www.ni.com

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

[25] http://www.iec.ch