UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

tesi di laurea

Progetto e sviluppo di un simulatore Ladder e FBD per il tool UniSim

Anno Accademico 2006-2007

relatore
Ing. Gianmaria De Tommasi

candidato
Enrico Granata
matr. 534/002065

Ringrazio i miei genitori per avermi consentito di intraprendere e proseguire gli studi, per aver avuto fiducia in me e per avermi sempre seguito ed incoraggiato.
Ringrazio altresì la professoressa Ada Lettieri per il preziosissimo aiuto da lei ricevuto lungo tutto il percorso dei miei studi.
Ringrazio infine il mio relatore, ing. Gianmaria De Tommasi, ed il mio tutore di tirocinio, prof. Alfredo Pironti, per la serietà ed il garbo con i quali hanno seguito e coordinato lo sviluppo di questo mio lavoro di tesi.

Indice

Introduzione

L'ambito di questa tesi è l'automazione industriale, ed in particolare i sistemi PLC (Programmable Logic Controller) compatibili con lo standard IEC 61131 sezione 3 "Programming Languages".

L'ambiente UniSim è un simulatore didattico in uso presso la Facoltà di Ingegneria dell'Università di Napoli Federico II, in particolare nell'ambito del corso di "Tecnologie dei sistemi di automazione". Esso è un ambiente di sviluppo e simulazione per programmi nei linguaggi dell'automazione definiti dallo standard IEC.

Il lavoro sviluppato in questa sede consiste nell'estensione del tool UniSim tramite l'aggiunta degli ambienti di editing e di simulazione per i linguaggi Functional Block Diagram e Ladder Diagram, di una funzionalità per la traduzione del linguaggio Sequential Functional Chart nel linguaggio Ladder Diagram. Inoltre, si è esteso il sistema dei tipi di dati dell'ambiente UniSim aggiungendo il nuovo tipo REAL per supportare l'aritmetica in virgola mobile, e si sono riviste le funzionalità aritmetiche preesistenti del linguaggio Sequential Functional Chart per consentire l'uso di questo nuovo tipo. Infine, si è prevista la possibilità per l'utilizzatore di creare ed utilizzare POUs di tipo function, l'equivalente nella terminologia dello standard 61331 di una funzione. In particolare si supporta la creazione di funzioni, il passaggio ad esse di parametri, e la loro valutazione come parte di blocchi funzionali nei due linguaggi FBD e LD, tutto ciò per via grafica e in maniera indistinguibile per l'utilizzatore dalle funzioni predefinite dell'ambiente UniSim.

Questa tesi si propone di descrivere le tecnologie utilizzate nello svolgimento di tale lavoro di sviluppo software, le tecniche e le scelte progettuali adottate. In particolare, si darà enfasi alle strutture dati scelte, agli algoritmi usati per lavorare con esse, e ad informali valutazioni sulla correttezza dei ragionamenti svolti. Si

daranno, ove sia opportuno, brevi cenni alla complessità computazionale, senza appesantire la trattazione con dimostrazioni e calcoli matematici. Si intende porre in evidenza l'utilizzo delle tecniche di progettazione ad oggetti, e in particolare l'uso dei costrutti di interfaccia al fine di migliorare l'estensibilità del tool UniSim. Ove ciò abbia dato luogo alla necessità di refactoring, il lavoro necessario, le sue motivazioni e i benefici che si ritiene siano stati apportati saranno discussi.

In particolare, la tesi è strutturata in 5 capitoli.

Il primo capitolo, "Introduzione al tool UniSim", descrive il contesto in cui si inquadra il tool stesso e l'architettura del tool. Esso descrive come sia stata concepita l'architettura di UniSim a layer. Si focalizza poi sul componente centrale: la Class Library descrivendone il modello a oggetti e i principali componenti. Il capitolo, inoltre, motiva e descrive le scelte di refactoring fatte in questo lavoro, ed in particolare la migrazione di UniSim verso il framework .net versione 2.0, rispetto alle versioni 1.0 ed 1.1 utilizzate in precedenza.

Il secondo capitolo, "Specifica delle nuove funzionalità", descrive i requisiti con cui si è iniziato lo sviluppo. In particolare esso si sofferma sulla descrizione dei linguaggi FBD e LD, basate entrambe sullo standard. Il capitolo inoltre descrive la realizzazione del nuovo tipo di dati REAL e brevemente descrive il ragionamento con cui si sono introdotte le POUs function nell'ambiente. Infine, esso introduce il convertitore SFC → Ladder.

I capitoli terzo e quarto trattano rispettivamente del supporto LD ed FBD introdotto in UniSim, in entrambi i casi partendo da una *overview* del linguaggio da gestire, che in parte riprende quanto detto nel secondo capitolo. In seguito essi analizzano e descrivono le strutture dati con cui si sceglie di rappresentare il linguaggio, in un caso tramite una classe Rung, nell'altro tramite un parse tree generale, enfatizzando come sia possibile convertire ogni costrutto del linguaggio nella struttura dati selezionata e come quindi essa sia appropriata per tale rappresentazione. In particolare, il capitolo terzo conduce una dettagliata disamina dell'algoritmo RtL, introducendo brevemente anche un precedente algoritmo risultato poi scorretto di nome LtR, di cui l'RtL risulta, e ciò viene chiaramente evidenziato, un discendente simmetrico.

Il capitolo quinto, infine, "Traduzione del Sequential Functional Chart in Ladder", descrive le varie fasi del processo di traduzione di una POU SFC in una POU equialente LD, introducendo sia le scelte fatte per tradurre i concetti propri dell'SFC in quelli propri dell'LD, e sia gli algoritmi usati e le possibilità di

estensione di questi al fine di completare tale traduttore.

Capitolo 1
Introduzione al tool UniSim

UniSim è un ambiente di sviluppo per i linguaggi dell'automazione, così come definiti dallo standard IEC 61131-3 [1], sviluppato nel corso di diversi progetti di tesi, dagli studenti della Facoltà di Ingegneria dell'Università degli Studi di Napoli Federico II, ed adottato dalla medesima quale strumento didattico nel corso di "Tecnologie dei Sistemi di Automazione".

Esso supporta i tre linguaggi grafici dello standard: Sequential Functional Chart (SFC), Ladder Diagram (LD), Functional Block Diagram (FBD).

Usando UniSim è possibile, quindi, scrivere e testare (attraverso simulazione puramente software o interfacciandosi con un sistema reale attraverso schede hardware standard) programmi in uno o più di tali linguaggi. UniSim consente anche la gestione di processi reali non critici e l'interfacciamento con altri ambienti software che adottino il formato di file *XML Format for IEC 61131-3* [2]. Tale formato standard, sviluppato dal comitato vendor independent PLCOpen, è concepito per consentire l'interscambio di progetti o anche solo di singole POU tra diversi ambienti di programmazione. Sin dalla sua prima versione, UniSim ha adottato tale formato [3].

Il seguito del capitolo descriverà succintamente l'architettura del tool, evidenziando in particolare come essa sia stata concepita inizialmente, ed estesa nel corso del tempo, per consentire una sempre migliore estensibilità e facilitare, così, quei problemi di integrazione che sorgono sovente quando persone diverse dagli autori originari, si trovano a lavorare su un progetto software di complessità non banale.

1.1 L'architettura di UniSim

Se è concepibile che un piccolo strumento di poche centinaia di righe di codice e nato per uno scopo ben preciso e limitato, possa essere sviluppato senza un preciso progetto a guidare l'implementatore, ciò è impensabile per un progetto di media complessità, già inizialmente pensato per crescere e mutare. UniSim è ad oggi un programma di oltre 50 kLOC, la sua prima versione risale ad almeno 3 anni fa, ed è stato usato da svariate centinaia di persone. È pertanto un progetto importante ed ambizioso, e non è pensabile nel breve termine che esso venga messo da parte e sostituito da altri strumenti software.

Essendo lo sviluppo di UniSim affidato a tesisti che svolgono sia la funzione di progettista che quella di programmatore, è fondamentale per essi conoscerne l'architettura. Ciò facilita sia la risoluzione di bug nell'esistente, sia l'aggiunta di funzionalità, sia l'estensione dell'architettura stessa ove questo sia ritenuto necessario.

Questo capitolo si propone di descrivere, seppure a grandi linee, l'architettura di UniSim così come oggi si presenta e la *rationale* che ha portato a concepirla così invece che in uno qualsiasi degli altri infiniti modi possibili.

La scelta della tecnologia Microsoft .net

UniSim è stato sviluppato utilizzando il linguaggio Microsoft Visual Basic .net, uno dei linguaggi previsti dalla tecnologia Microsoft .net

Tale tecnologia, rilasciata in versione 1.0 nel 2002 insieme al relativo ambiente di sviluppo, Microsoft Visual Studio.net, è stata la risposta di Microsoft all'invenzione di Java da parte della Sun Microsystems.

A differenza di Java, linguaggio unico ma compilato in un bytecode indipendente dalla piattaforma, ed eseguibile da ogni SO reale su cui si possa far girare una Java Virtual Machine (una CPU software che esegue, appunto, il bytecode emesso dal compilatore Java), il .net si inserisce quale minimo comun denominatore tra diversi linguaggi grazie alla astrazione fornita dal MSIL (Microsoft Intermediate Language) e dalla BCL (Base Class Library) [4,5,26,27].

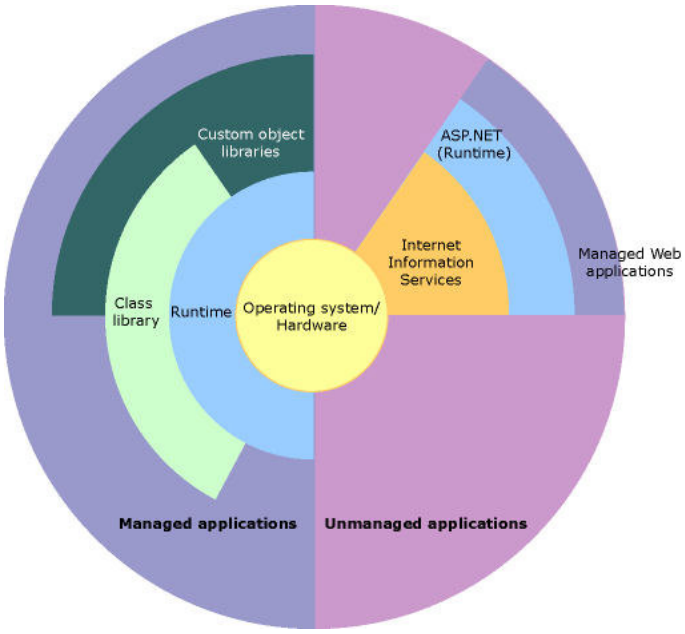

Figura 1. Visione d'insieme dell'ambiente .net

Le applicazioni, compilate in MSIL (ed eventualmente tradotte in codice macchina del sistema target da un compilatore JIT, Just in time), si dicono *managed*, gestite, in quanto fanno uso della gestione automatica della memoria con garbage collector generazionale, fornite dal framework.

Il punto centrale di questa tecnologia è che il MSIL è un target di compilazione concepito per poter tradurre linguaggi molto diversi tra loro, tant'è che anche linguaggi quali il COBOL (opportunamente esteso per supportare il paradigma di programmazione OOP) possono essere compilati verso tale "architettura". I linguaggi previsti per default da Microsoft sono il Visual Basic .net (una revisione del precedente ambiente di sviluppo Visual Basic della Microsoft stessa), il C# (un nuovo linguaggio standardizzato dalla ISO, e sviluppato appositamente per .net [28]) e il C++ con Managed Extensions (un insieme di parole chiave che aggiungono al C++ standard il supporto per i concetti del .net), ma nulla vieta di produrre compilatori per altri linguaggi, e lo stesso framework include classi di utilità per generare codice MSIL da un proprio programma.

Oltre ad un linguaggio target comune, ciò che unifica i diversi linguaggi sotto l'ombrello del .net, è l'uso di un sistema di tipi comuni (CTS), che assicura, per esempio, che un intero con segno definito da un programma Visual Basic abbia la stessa rappresentazione e dimensione di un intero con segno definito da un programma C#, e l'adozione di una libreria di runtime comune (la Base Class

Library). Lo standard per la tecnologia .net [29] prevede diversi livelli di compatibilità in base a quali librerie si sceglie di implementare e possibilità di aggiungere nuovi componenti nativi della implementazione (un esempio è fornito dalla stessa Microsoft, che include la libreria Windows Forms per la programmazione nell'ambiente a finestre Windows).

Teoricamente, nulla osta ad un porting della tecnologia su ambienti diversi dal SO Windows di Microsoft, analogamente a quanto avvenuto per Java, che non è limitato al SO Solaris di Sun, ma ad oggi l'unico sforzo in tal senso, ancora non completo, è di un consorzio indipendente, finanziato però da Novell Inc, e il prodotto (open source sotto licenza GPL) risultante è detto Mono [30]. Esso gira su piattaforme Linux e Mac OS X (sia PowerPC che Intel). Siccome ad oggi, il Mono non è sufficientemente completo per supportare UniSim (come riportato dal sito web dell'organizzazione che lo produce [6]) nel seguito della discussione si farà esclusivo riferimento al .net Framework di Microsoft Corporation.

Di tale Framework, la Microsoft rilascia periodicamente versioni aggiornate. Ad oggi, in parallelo al rilascio del SO Windows Vista è stata rilasciata la versione 3.0, aggiornata per far uso delle nuove tecnologie incluse in tale release del sistema operativo. La stessa Microsoft ha attualmente nello stato di beta il Visual Studio 2008 che si aggancia al .net 3.5, tra le cui principali funzioni vi è l'uso di un insieme di costrutti per implementare query à la SQL all'interno della sintassi dei linguaggi C# e VB.net [7].

UniSim ha sempre fatto uso delle versione 1.0 ed 1.1 del framework (con la 1.1 sostanzialmente analoga alla 1.0 fatte salve correzioni di bug). In questo lavoro di tesi si è scelto di portare il tool sulla versione 2.0 del framework, una scelta delicata ed importante in quanto non reversibile. I motivi di tale scelta sono molteplici e spaziano dalle migliorie prestazionali apportate alla compilazione JIT su piattaforma x86, ai nuovi costrutti di programmazione resi possibili da tale versione, in particolare l'uso di *generics*, ovvero di tecniche tipiche della programmazione generica. Pur meno potenti e generali che in C++, i costrutti di generics del .net 2.0, applicati in particolare alle collezioni, hanno permesso di ridurre drasticamente l'uso di late binding[1] in UniSim e pertanto di migliorare le prestazioni ed aumentare la sicurezza sui tipi (molti controlli vengono ora svolti a

[1] Il late binding fa parte naturalmente dell'ambiente .net ed è funzionalmente analogo all'uso dei metodi virtual in C++. Il concetto di late binding qui riferito è diverso, e si avvicina molto al supporto offerto dal linguaggio Objective C per la risoluzione a runtime delle chiamate indipendentemente dalle interfacce dei tipi definite a tempo di compilazione [25]

tempo di compilazione e gli errori relativi riportati senza bisogno di test).

La suddivisione in componenti

UniSim è stato concepito come un'architettura a due livelli: un livello sottostante "server", implementato da una DLL, ed un livello sovrastante costituito dalle applicazioni "client" che sfruttano i servizi della libreria, chiamata UniSimClassLibrary. Questa concezione permette di immaginare diversi applicativi basati sulla stessa interfaccia e su comuni funzionalità. Essa ha consentito di produrre due applicativi client: UniSim DevApp e UniSim ControlApp.

La DevApp è il vero e proprio ambiente di sviluppo, in cui l'utente scrive, corregge, esegue il debug ed il testing software dei sui progetti d'automazione. D'altro canto, la ControlApp (non ulteriormente sviluppata in questo elaborato) consente la simulazione hardware-in-the-loop e il controllo di processi reali, facendo da interfaccia tra il motore di simulazione di UniSim e una scheda di I/O digitale.

Ovviamente, una parte non trascurabile della semplicità di cooperazione tra strumenti diversi è l'uso da parte di tutti del formato *XML Formats for IEC 61131-3*, già citato in precedenza, da cui poi ogni tool sarà in grado di estrarre tutte e sole le informazioni necessarie al proprio compito.

Sono ovviamente immaginabili e possibili altri tools basati sulla tecnologia di UniSim, un esempio dei quali potrebbe essere costituito da un tool che estrae le informazioni di documentazione dei vari elementi di un progetto UniSim e li raccoglie in un formato adatto alla consultazione, analogamente al tool javadoc.

Partendo da questa visione d'insieme, si può approfondire la concezione della ClassLibrary che è il vero nucleo del prodotto UniSim, essendo la DevApp e la ControlApp sostanzialmente dei *thin-client* di questa ultima

L'architettura della UniSimClassLibrary

La ClassLibrary mima sostanzialmente il modello architetturale descritto dallo standard IEC 61131-3, fatte salve le aggiunte necessarie all'implementazione:

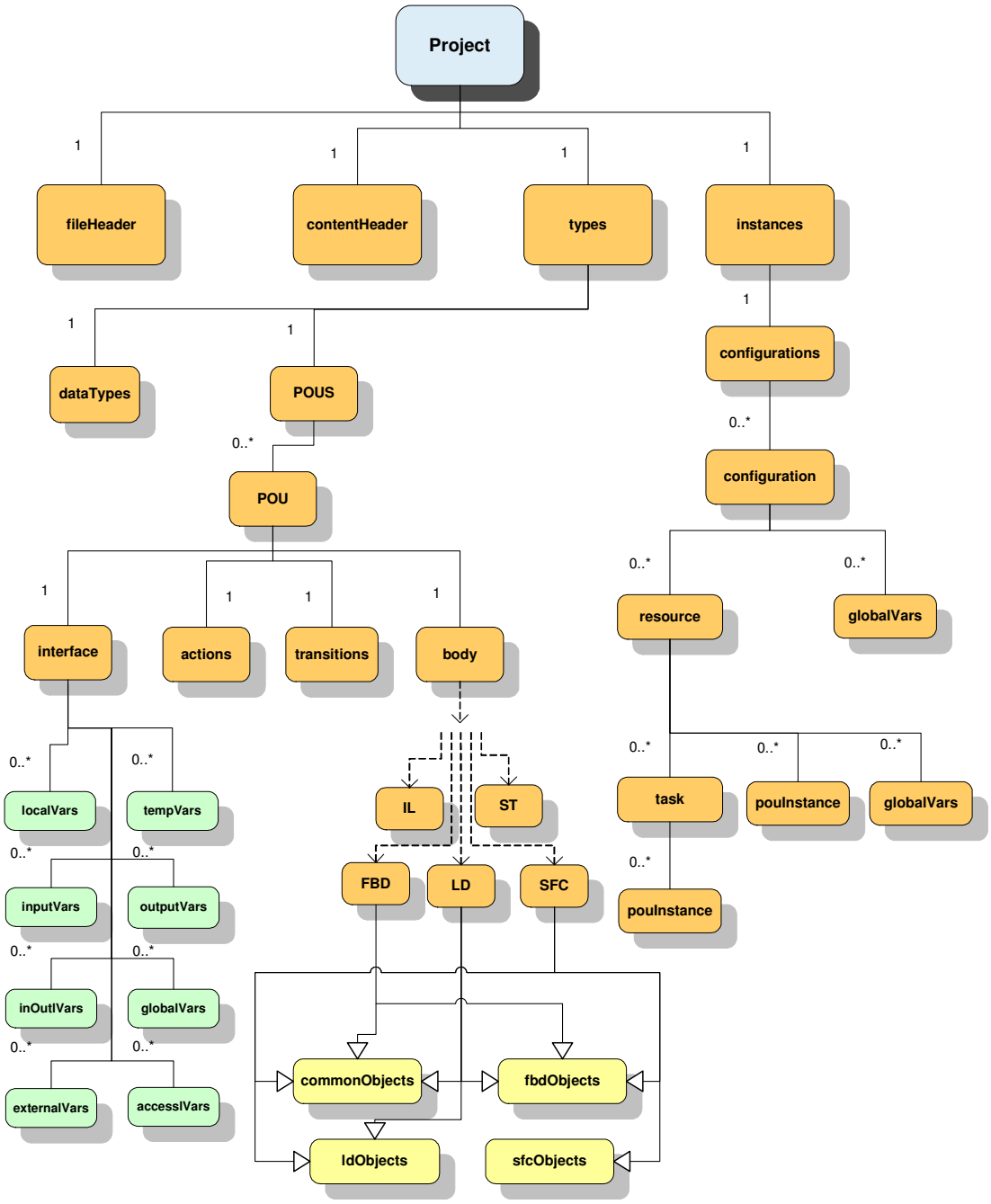

Figura 2. La struttura software descritta dallo standard IEC e adottata da UniSim

Sebbene non sia possibile "staccare" singole parti di questa struttura per potervi lavorare in maniera totalmente separata ed indipendente da tutte le altre, il punto nevralgico di questo elaborato di tesi è sicuramente il body, che è l'elemento architetturale che descrive il corpo di una POU (Program Organization Unit, nella terminologia dello standard). Lo standard prevede che un programma sia diviso in più blocchi, ognuno dei quali appunto detto POU. Tali POU possono essere di tre

tipi diversi, per ognuna delle quali si riporta la descrizione fattane dal glossario dello standard [1,8]:

- Program: l'insieme di tutti gli elementi e costrutti di programmazione necessari per realizzare le funzioni di processamento segnali richieste per il controllo di un dispositivo o di un processo da parte di un PLC [23]
- Function: una POU che, quando eseguita, ritorna esattamente un elemento dato e la cui invocazione può essere usata nei linguaggi testuali quale operando di una espressione
- Function Block: una istanza di un elemento di programmazione di un PLC che consiste della descrizione di una struttura dati divisa tra elementi dato di input, output, input/output e locali ed un insieme di operazioni da doversi effettuare su tali valori quando l'istanza stessa viene invocata

e sono ancora suddivise in una interfaccia e un corpo. L'interfaccia di una POU ne specifica le variabili, con particolare enfasi sulla classificazione delle stesse in base alla visibilità e al tipo di uso consentito. Il corpo della POU è la implementazione, scritta in uno dei cinque linguaggi dello standard (tre dei quali, quelli grafici, previsti in UniSim).

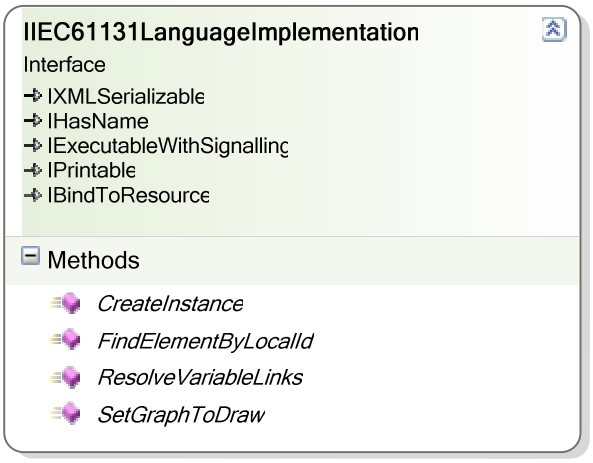

Figura 3. L'interfaccia IIEC61131LanguageImplementation

L'introduzione all'interno dell'ambiente UniSim di un nuovo linguaggio di programmazione, si basa sulla implementazione dell'interfaccia IIEC61131LanguageImplementation. Tale interfaccia descrive i comportamenti fondamentali che un qualsiasi editor ed interprete per un linguaggio standard deve

implementare per poter essere gestito da UniSim. Ci si aspetta inoltre che ogni linguaggio definisca elementi di programmazione suoi propri (ad esempio, le fasi nell'SFC, le bobine in LD, e i blocchi in FBD). Per tale motivo, la convenzione seguita nello sviluppo di UniSim è di creare una cartella all'interno dell'albero sorgente in cui inserire tutti i file di codice relativi alla definizione di un linguaggio.

Con questo elaborato si è proceduto ad una attività di *refactoring* di UniSim, in parte anche necessaria per perseguire la politica progettuale di eliminare il late binding. Infatti, sono state introdotte numerose interfacce nell'architettura del tool, ognuna relativa ad un ridotto insieme di comportamenti collegati, eventualmente usando l'ereditarietà multipla per creare interfacce più potenti. Le classi implementano quelle interfacce che sono più aderenti alla propria natura ed ai propri comportamenti o che è più ovvio ed utile per i client che esse adottino. In particolare, questo schema progettuale si basa sull'idea di suddividere in aree semantiche i comportamenti di ogni classe, e poter usare riferimenti non già alla classe in quanto tale ma all'interfaccia che descrive l'area semantica di interesse. Ciò consente una grande flessibilità nell'implementazione e supporta un paradigma di progettazione in cui una volta che un algoritmo è stato codificato a partire da una certa interfaccia, tutti gli oggetti presenti e futuri che implementeranno l'interfaccia potranno usufruire degli algoritmi ad essa associati in modo naturale ed elegante.

Mancando il .net di un sistema di templates analogo a quello del C++, in cui i riferimenti agli argomenti di un template vengono risolti a tempo di compilazione sul tipo effettivo degli argomenti (il .net richiede che l'argomento di un *generic* sia specificato come tipo System.Object o che venga specificato dall'implementatore del generic che si desiderano parametri con vincoli speciali e consente di invocare solo quei membri che saranno sicuramente ammissibili per il tipo formale dell'argomento), questo modus operandi consente una enorme flessibilità, forse la massima possibile nell'interno del CLR di Microsoft.

Il meccanismo di serializzazione XML

Come già specificato, UniSim fa uso del formato standard *XML Formats for IEC 61131-3*, standardizzato tramite il meccanismo degli *XML Schema* dal comitato internazionale PLCOpen [2]. L'XML è stato concepito come versione

semplificata e più pratica all'uso dell'SGML, formati di definizione per documenti di tipo markup [9]. In tali documenti la sintassi è specificata attraverso elementi di demarcazione detti tag, ovvero stringhe incluse tra parentesi angolari < e >, all'interno delle quali si può trovare un nome e zero o più attributi formati da coppie del tipo *chiave*="*valore*" (con i marcatori " e " necessari). I tag di un markup si dividono in apertura e chiusura. Gli attributi vanno posti all'interno del tag di apertura, mentre il tag di chiusura è dal marcatore di chiusura </, dal nome del tag e dal segno >. Tutto ciò che si trova tra i tag di apertura e di chiusura è il contenuto del tag, e può essere formato da testo, da altri tag o da un insieme delle due cose. È ammissibile in un tag senza contenuto fondere l'apertura e la chiusura usando il simbolo di chiusura /> a fine del tag di apertura, potendosi indicare anche gli eventuali attributi.

Il più famoso linguaggio definito in termini dell'SGML è sicuramente l'HTML, Hypertext Markup Language, utilizzato per formattare i documenti del World Wide Web. Pur essendo stato definito in termini di un markup rigoroso come l'SGML e pur essendo enorme l'insieme di documenti scritti in HTML oggi fruibili tramite Internet e non solo, il linguaggio soffre di due principali problemi:

1. L'HTML, così come interpretato dai programmi di lettura, detti browsers, è meno rigoroso di quanto sin qui descritto e consente varie omissioni e forzature della sintassi. Sebbene il risultato di queste variazioni dipenda dall'implementazione, cioè dello specifico browser, ormai è universalmente accettabile in HTML l'omissione, ad esempio, dei simboli " nella definizione del valore degli attributi, e la mancata chiusura di alcuni tag la cui terminazione possa essere ricavata dalla chiusura di tag di livello superiore o da opportune euristiche contestuali (molto diffuso nella corrente programmazione HTML è l'uso del tag <br> per indicare il ritorno a capo forzato senza un opportuno tag </br> oppure non usando la forma coincisa <br/> o del tag per indicare un nuovo paragrafo senza il di chiusura del precedente paragrafo)
2. L'HTML è un linguaggio per la formattazione visiva, nel senso che i suoi tag corrispondono a ciò che l'utente, con buona approssimazione, vedrà sullo schermo del suo computer (fatti salvi i casi limite di browser testuali quali il lynx). I sistemi di processamento automatico hanno bisogno di una formattazione semantica per poter facilmente processare i dati, ma ciò in HTML non è possibile se non a costo di forzare il linguaggio, oppure solo

nelle versioni più recenti usando il supporto per i Cascading Style Sheets (CSS) e separando in parte formattazione da semantica [10]

Per ovviare a questi problemi, il W3C (World Wide Web Consortium) ha introdotto alla fine degli anni '90 il linguaggio XML, che consente di definire dei propri schemi di markup di livello semantico ed applicarli a documenti. Il formato XML è semplice da usare, e più rigoroso dell'HTML evitando quindi a priori il primo dei problemi discussi: un documento XML è ben formato o non ben formato, ed è richiesto ai programmi di rifiutare i documenti non ben formati. Diverso è il problema della validità di un documento XML: chiunque può scrivere un documento XML ben formato anche inventando per la singola occasione un formato di markup conveniente. Per garantire però che documenti prodotti da diverse persone, con diversi strumenti, e in tempi diversi, siano tutti coerenti con l'intenzione, il W3C ha inventato due strumenti: il DTD e l'XML Schema. Il DTD è un formato di documento ad hoc, che stabilisce a grandi linee il contenuto di documenti XML. L'XML Schema è esso stesso un documento XML che stabilisce in maggior dettaglio il contenuto di ogni singolo tag, potendosi spingere anche a definire il tipo degli attributi, e il conteggio esatto dei sottotag di ogni tag, distinguendoli anche per tipo. Essendo la tecnologia più rigorosa e più aggiornata, il PLCOpen ha scelto di adottare per definire il proprio formato l'XML Schema.

Tale scelta si è rivelata fortunata, in quanto il .net Framework ha abbracciato le tecnologie basate su XML e quindi include un lettore XML con validazione, ovvero in grado di segnalare se un documento XML è o meno valido rispetto allo schema che esso dichiara di voler seguire[2].

L'approccio usato da UniSim nel caricare e salvare i documenti XML (procedimento noto come serializzazione: trasferire un insieme di oggetti con collegamenti i più vari tra loro in una serie, appunto, di byte) è basato su un pattern di delega: ogni oggetto serializzabile in XML sarà composto da una parte sua propria e da altri oggetti serializzabili in XML. L'oggetto crea il tag ad esso dedicato ad interno del formato usato, vi scrive gli attributi o i contenuti necessari e poi delega ai suoi oggetti contenuti il salvataggio della parte loro propria. In fase di caricamento, ogni oggetto riceve un riferimento al lettore di XML *puntato* sul tag di propria competenza e carica i suoi elementi. Si segue la convenzione che ogni

[2] A differenza della *well formedness*, è possibile per un applicativo ignorare la non validità rispetto alla DTD o allo schema. UniSim sfrutta, infatti, questa opzione per poter tentare di aprire documenti non validi (ad esempio per facilitare la recovery). Nulla può essere invece fatto con documenti malformati usando i componenti del framework

oggetto sappia cosa potrà essere contenuto all'interno del proprio tag, e che quando incontra un tag che appartiene ad un oggetto da esso contenuto, deleghi a quest'ultimo la propria creazione, passando ad esso il lettore XML puntato sul tag corretto. Ricorrendo questo meccanismo, si vede che si è correttamente salvato su disco un progetto XML e se ne è ricreata un'immagine in memoria. UniSim usa un meccanismo in due tempi per la scrittura su disco: prima crea in memoria il documento XML vero e proprio e poi lo trascrive su disco quale file di testo semplice. Usando questo meccanismo è sempre possibile, invocando gli appositi metodi, conoscere la trasposizione in XML di ogni oggetto all'interno di un progetto UniSim, incluso il progetto stesso.

Il principale problema di tale meccanismo, ma ciò è comune a quasi tutti i sistemi di salvataggio e caricamento di documenti, è che non vi sia perfetta corrispondenza tra quanto salvato e quanto caricato (ad esempio, alcuni attributi che non vengono salvati e che andrebbero ricalcolati non vengono ricalcolati) e ciò possa causare errori solo quando si apre un documento da disco invece che quando lo si crea ex novo. A ciò, oltre ad una solida struttura ad oggetti, può ovviare solo la corretta pratica nella programmazione, l'attenzione del programmatore e un'analisi approfondita dello stato degli oggetti e della correlazione tra lo stato di ogni oggetto, le azioni dell'utente, le modifiche dello stato, e le interazioni tra gli oggetti.

La UniSim DevApp

Come già indicato in precedenza, il principale client del componente software sin qui descritto è la UniSim DevApp, l'applicazione che consente all'utilizzatore di creare, modificare e simulare progetti d'automazione.

La DevApp è sostanzialmente composta di un Form di tipo MDI (si tratta della funzionalità fornita dall'ambiente grafico Windows di poter avere una finestra al cui interno "vivono" altre finestre e che Windows stesso fa cooperare secondo una sua propria logica [4]) che ospita al suo interno un pannello grafico con gli strumenti di controllo per la simulazione, un pannello laterale con l'elenco ad albero degli elementi del progetto divisi tra POUs e risorsa, e una barra di menu ed una toolbar con i comandi propri dell'applicativo, in analogia con le linee guida di stile Microsoft per le applicazioni Windows. Un altro importante componente incluso nella DevApp è la finestra che consente all'utente di gestire le preferenze

per l'ambiente UniSim. La gestione di queste, però, è demandata alla UniSim ClassLibrary e in gran parte le preferenze non hanno influenza sulla DevApp ma sulla ClassLibrary.

L'utente, anche grazie all'uso del sistema MDI fornito da Windows, ha l'illusione di star operando con un unico componente software, mentre in realtà si passa in maniera trasparente dalla DevApp alla ClassLibrary e di nuovo alla DevApp, sfruttando appieno i canoni dell'orientamento agli oggetti.

1.2 Conclusioni

Come per ogni modello architetturale, non è possibile a priori e in astratto, dire se esso sia veramente il più valido per assolvere lo scopo prefissato, né tantomeno che esso lo sarà anche per assolvere gli scopi di domani. Il lavoro del progettista software è di definire, con metodologia ma anche, e forse soprattutto, con intelligenza competenza ed esperienza, un modello software che gli implementatori possano tradurre in un programma efficiente, funzionale, usabile e quanto più possibile estendibile e manutenibile. Solo il tempo può dire se e in che misura il lavoro del progettista è valido o se risulta necessaria una più o meno complessiva revisione architetturale. Il software è un'entità dinamica, in costante evoluzione, e le architetture spesso cambiano insieme ai requisiti in tempo reale. Sta al bravo progettista prevedere questi cambiamenti, organizzare sin da subito il suo lavoro in modo che possa evolvere, intuirne i punti di possibile estensione e, spesso, guardare anche più oltre del proprio committente nell'immaginare ciò che oggi appare cristallizzato e domani non lo sarà più.

Una buona architettura, pertanto, più che quella perfetta di per sé, è quella che più facilmente si adatta al cambiamento [31]. La speranza è che questa sin qui descritta sia una buona architettura.

Capitolo 2
Specifica delle nuove funzionalità

Questo capitolo si propone di specificare e descrivere le nuove funzionalità aggiunte allo strumento UniSim che sono l'oggetto di questo elaborato di tesi. Tali funzionalità sono state concepite per integrarsi nell'ambiente preesistente, sia a livello di consistenza dello schema progettuale e di codifica, sia a livello dell'interfaccia grafica verso l'utente. Ciò non ha impedito, ovviamente, di apportare quei miglioramenti che si sono ritenuti necessari in entrambi gli ambiti.
In primo luogo, le nuove funzionalità hanno necessitato di strumenti grafici che consentissero all'utente di interagire con esse, e questi ultimi sono stati sviluppati tenendo conto delle linee guida date dalla GUI esistente. Ove si è ritenuto, però, che questa interfaccia esponesse uno schema di interazione non naturale e prolisso, si è proceduto a modificarla, e a rendere consistente il nuovo sviluppo con tali modifiche. Si è fatto uso di Forms già sviluppati per funzioni che essi assolvono e di cui la nuova interfaccia ha bisogno, e si è concepita la nuova parte dell'interfaccia in modo estensibile e manutenibile, in modo che sviluppatori futuri che lo ritengano possano con pari facilità far uso di queste funzionalità.

Per quanto riguarda la progettazione della parte non grafica, si sono seguite le linee guida della Object Oriented Programming, ed in particolare il principio già esposto nel Capitolo 1, della definizione di un ricco insieme di interfacce (supportate nativamente dall'ambiente .net, a differenza del linguaggio C++ in cui l'interfaccia si crea de facto usando i metodi virtuali puri [11]). Lo sforzo di refactoring per inserire anche il codice preesistente all'interno del nuovo modello progettuale, è

stato ripagato dalla riduzione del late binding basato su reflection, e dal conseguente incremento prestazionale, oltre che da una maggiore chiarezza e manutenibilità del codice.

2.1 Aggiunta di nuovi linguaggi

UniSim nasce come tool di sviluppo per il linguaggio Sequential Functional Chart, e tale sostanzialmente rimane per un certo periodo di tempo. Nondimeno, lo standard IEC 61131-3 include ben cinque linguaggi[3] e l'architettura di UniSim non è di ostacolo all'aggiunta di supporto per linguaggi diversi dall'SFC. Anzi, l'autore originale del software, nell'elaborato di tesi in cui ne descrive la prima versione [3], indica chiaramente quella dell'aggiunta degli altri linguaggi grafici come una delle strade principali per l'estensione futura del tool.

Questo lavoro di tesi aggiunge ad UniSim una implementazione del linguaggio LD e del linguaggio FBD, completando in gran parte il progetto di estensione allora previsto. L'attuale versione del tool include infatti i tre linguaggi grafici dello standard (gli altri due, IL e ST, sono linguaggi di tipo testuale simili all'Assembler ed al Pascal) e consente, se desiderato, il passaggio dal SFC al LD.

Il linguaggio a contatti (LD)

Il linguaggio a contatti nasce agli albori dell'automazione basata su tecnologie informatiche per semplificare il passaggio degli esperti in tecnologie di tipo relè, in uso all'epoca, ai nuovi ambienti. Il Ladder si basa sui concetti di contatto, bobina e flusso di energia, attraverso un insieme di regole semplificative di quelle con cui la corrente elettrica scorre attraverso un circuito reale.

[3] Alcuni autori ritengono che i linguaggi definiti dallo standard siano quattro: IL, ST, FBD e LD. Essi pensano che l'SFC sia un formalismo basato sulle reti di Petri, in quanto dipende dagli altri quattro per la definizione di azioni e transizioni [8]. UniSim però implementa l'SFC in modo del tutto indipendente, per cui si seguirà in questo lavoro di tesi l'interpretazione secondo la quale tutti e cinque sono linguaggi veri e propri.

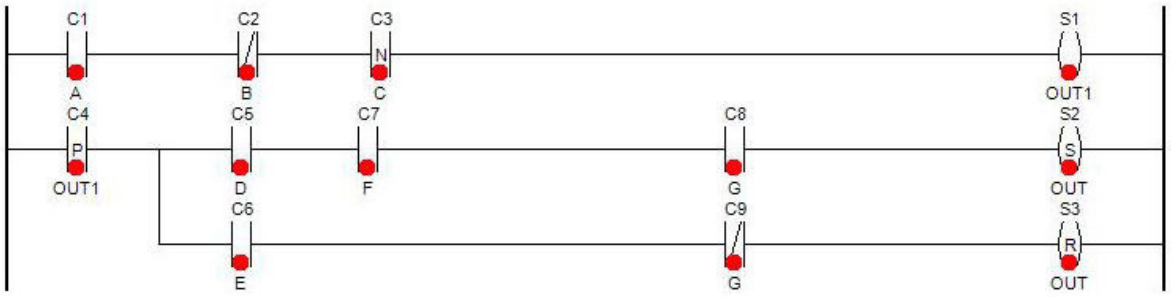

Figura 4. Un semplice programma in linguaggio a contatti prodotto con UniSim

Gli elementi fondamentali del linguaggio LD sono le alimentazioni destra e sinistra, rappresentate da due lunghe linee verticali, i contatti, rappresentati da due piccole linee verticali e parallele (con eventualmente una lettera di specifica del significato all'interno), e dalle bobine, rappresentate da due linee curve a mo' di parentesi (con anch'esse eventuali specifiche all'interno). I programmi Ladder sono divisi in *rung*, ognuno dei quali determinato dalla bobina alla sua destra (nell'esempio vi sono 3 rung: OUT1, S OUT e R OUT). Scopo di ogni rung è valutare una funzione booleana e scrivere il suo risultato sulla variabile associata alla bobina, in base sia al valore della funzione stessa sia al tipo di bobina. Lo standard IEC 61131-3 definisce i seguenti tipi di contatto e la semantica associata:

Tipo di contatto	Semantica
Normalmente aperto	Il valore associato al contatto è uguale a quello della variabile associata
Normalmente chiuso	Il valore associato al contatto è la NOT di quello della variabile associata
A fronte di salita	Il valore associato al contatto è alto se e solo se si rileva un fronte di salita sulla variabile associata
A fronte di discesa	Il valore associato al contatto è alto se e solo se si rileva un fronte di discesa sulla variabile associata

La valutazione di un rung procede attraverso percorsi da sinistra verso destra, in cui le biforcazioni corrispondono ad OR logici e le sequenze di contatti ad AND logici. Se un contatto lungo un percorso ha un valore falso, in base alla tabella di cui sopra, il percorso ha valore falso in toto. Se esiste almeno un percorso che ha valore logico alto, il rung ha valore logico alto. Le bobine, una per rung in UniSim, ricevono una

OR di tutti i percorsi che portano ad esse secondo la regola sinistra → destra e valutano come segue:

Tipo di bobina	Semantica
Normalmente aperta	La variabile associata alla bobina riceve il valore logico del rung
Normalmente chiusa	La variabile associata alla bobina riceve il NOT del valore logico del rung
A fronte di salita	La variabile associata alla bobina riceve il valore logico alto se e solo se il rung è passato da basso ad alto nel corso del ciclo di scansione, basso altrimenti
A fronte di discesa	La variabile associata alla bobina riceve il valore logico alto se e solo se il rung è passato da alto a basso nel corso del ciclo di scansione, basso altrimenti
Set o Latch	La variabile associata alla bobina viene impostata al valore logico alto se il valore logico del rung è alto, altrimenti non viene modificata
Reset o Unlatch	La variabile associata alla bobina viene impostata al valore logico basso se il valore logico del rung è alto, altrimenti non viene modificata

Lo standard definisce anche un tipo di bobina a ritenzione, che, basato su un supporto hardware, consente di mantenere il valore della variabile anche a seguito di interruzione dell'alimentazione al PLC. Ovviamente, per un ambiente di simulazione quale è UniSim, e non disponendo di un opportuno supporto hardware, questa classe di bobine è stata esclusa dall'implementazione, mentre vi è completo supporto per tutti gli altri tipi di bobina e contatto qui indicati, e tale supporto è compliant con lo standard IEC.

Il Functional Block Diagram (FBD)

Il diagramma a blocchi funzionali è un linguaggio molto usato, anche a livello descrittivo, per specificare il funzionamento di componenti elettronici o logici (può essere considerato un linguaggio di tipo funzionale la descrizione in termini di gate logiche di un circuito elettronico).

In particolare, lo standard IEC definisce come *Functional Block Diagram* "una o più reti di funzioni, blocchi funzionali, elementi dato, etichette ed elementi di connessione rappresentati per via grafica" [1,5].

L'implementazione UniSim del FBD non include i blocchi funzionali e le etichette (che vengono usate nello standard per definire i salti), ma include un ricco set di elementi funzione e grazie alla ricchezza di tipi di dato previsti in UniSim (booleani, interi e valori a virgola mobile) consente di valutare una ricca gamma di funzioni di interesse.

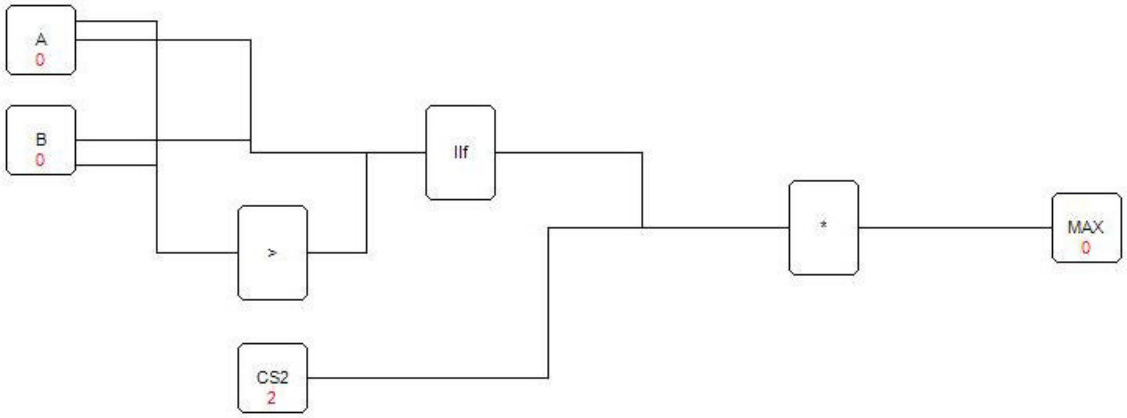

Figura 5. Un semplice programma in FBD prodotto con UniSim

Gli elementi fondamentali dell'FBD in UniSim sono le variabili (di ingresso o uscita), e i blocchi, ognuno dei quali rappresenta una funzione.

L'FBD viene eseguito scorrendo le uscite, e per ognuna eseguendo, secondo un algoritmo che sarà illustrato dettagliatamente in un successivo capitolo, la rete associata. Dal calcolo di tale rete si ottiene un valore che diviene, appunto, il valore dell'uscita nel successivo ciclo di scansione (in questo esempio, il programma riportato calcola il doppio del valore massimo tra A e B).

E' possibile in un singolo diagramma inserire più uscite e determinare un ordine di priorità tra esse sia automaticamente (con un algoritmo di tipo FIFO: l'ordine di inserimento delle uscite nel diagramma è l'ordine della loro esecuzione), sia manualmente associando alle diverse uscite, e quindi alle reti associate, una priorità.

A differenza dell'LD, basato su un flusso di corrente virtuale, l'FBD è basato su un flusso di segnale virtuale, in cui le varie connessioni sono viste semanticamente come elementi in grado di "trasportare" un valore dal loro terminale sinistro (d'ingresso) al loro terminale destro (d'uscita). In effetti, una variabile d'uscita può

essere vista come una funzione che ha un ingresso e nessuna uscita significativa (tant'è che le variabile d'uscita non hanno un punto di connessione destro) e il cui compito è scrivere l'ingresso nella variabile associata alla funzione. Così, sempre dovendosi intendere il tutto quale metafora e non linea guida per la progettazione o implementazione, una variabile d'ingresso può vedersi come una funzione che non valuta i suoi ingressi (e infatti non è preposta a riceverne mancando del punto di connessione sinistro), ma restituisce un valore significativo in uscita: il valore della variabile associata.

Esistono poi i veri e propri blocchi funzione, i quali hanno ingressi significativi ed uscite significative che, in genere, dipendono almeno in parte dai loro ingressi. A differenza della rappresentazione classica, ma non in violazione dello standard, i blocchi grafici di UniSim hanno un unico punto di ingresso a cui vanno collegati tutti gli input di una funzione, e l'ordine (ove questo risulti significativo) è dato dall'ordine di collegamento. Esiste, beninteso, una funzione del programma che consente di visionare ed eventualmente modificare questo ordine in caso di errori da parte dell'utente. Nel pieno rispetto dello standard, l'FBD di UniSim supporta sia connessioni affermate che connessioni negate (in cui il valore che effettivamente viene "visto" alla destinazione è il NOT logico del valore in ingresso).

L'editor e l'interprete FBD cooperano per consentire, in linea con lo spirito dell'intera concezione di UniSim quale strumento di sviluppo rapido (in analogia con gli ambienti Rapid Application Development), una valutazione delle reti *lazy*.

Nessun controllo formale sugli ingressi e le uscite dei blocchi viene effettuato a tempo di creazione della rete, ed è prevista la più ampia gamma di casting possibili (si può ad esempio collegare una uscita di tipo INT al risultato di un prodotto tra due valori REAL e UniSim eseguirà automaticamente l'adattamento del valore all'uscita).

2.2 Il tipo REAL

Nel momento della sua creazione, UniSim include il tipo di dati BOOL. Nel corso di due successivi lavori di tesi, viene aggiunto anche il tipo di dati INT, prima con la semantica di un contatore con il solo supporto di azioni Increment, Decrement e Reset ed in seguito con il pieno e completo supporto all'aritmetica, grazie ad un nuovo tipo di azione e di condizione di transizione in linguaggio SFC.

In questa sede, ci si è proposti di includere un terzo tipo di dati: REAL, un tipo di dati per numeri in virgola mobile secondo lo standard floating-point IEEE [13].
Il lavoro di estensione del sistema dei tipi di dato si è mosso lungo tre direzioni:

- Gestione delle variabili REAL: lettura e scrittura del loro valore, creazione di variabili REAL, loro salvataggio e caricamento da file
- Supporto per parametri di tipo REAL nel linguaggio LD e FBD, supporto per la conversione REAL ←→ INT
- Estensione della gestione delle condizioni ed azioni aritmetiche dell'SFC per consentire l'uso di variabili REAL

Aggiunta di un nuovo tipo di dati

Il sistema di variabili di UniSim si basa su una classe BaseVariable, che definisce metodi generali per la creazione di variabili e per la lettura e scrittura dei valori di ciascuna variabile

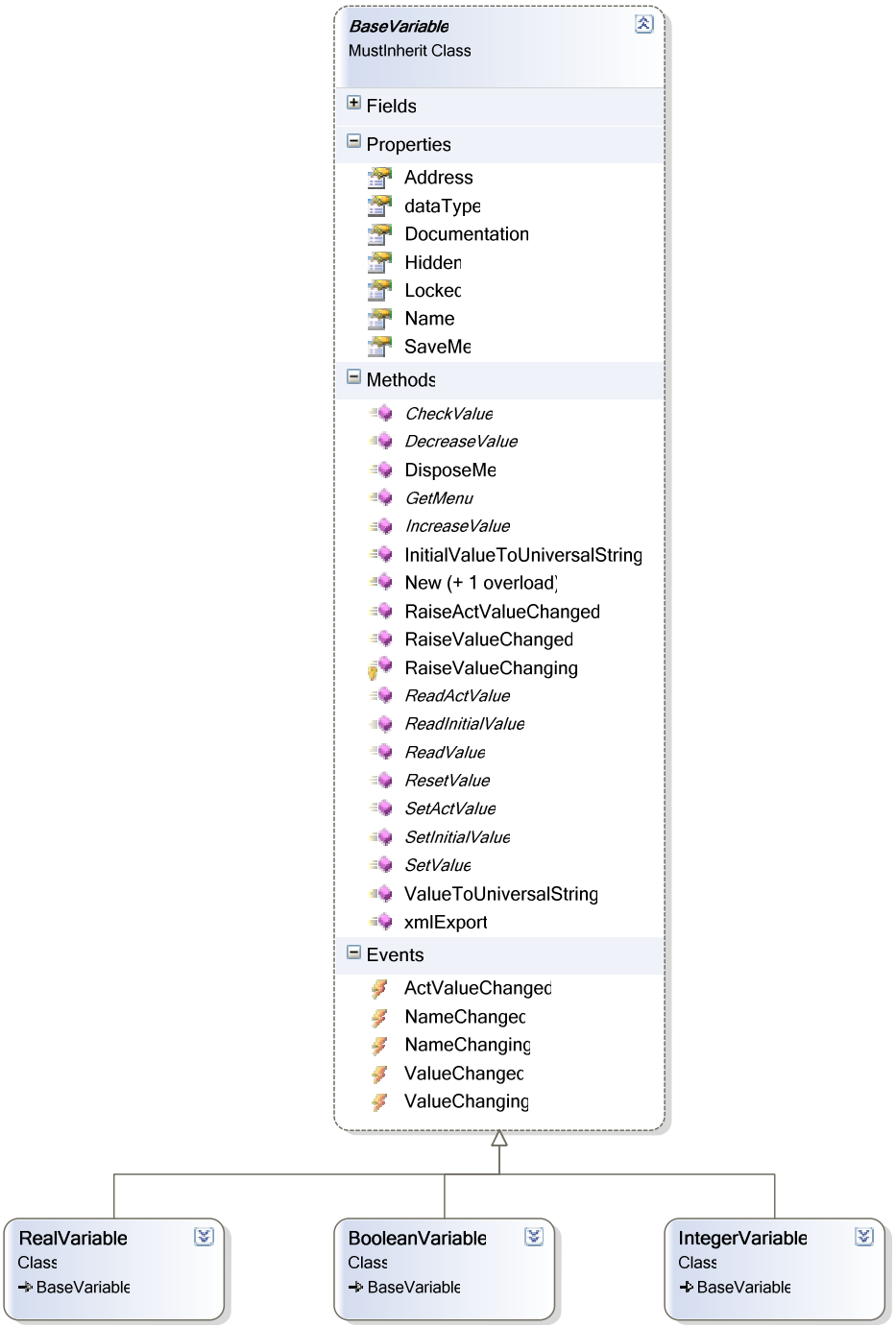

Figura 6. Diagramma delle classi per il sistema di gestione delle variabili

Al momento dell'aggiunta dei contatori all'ambiente si è deciso di seguire la via di associare a ciascun tipo di dato due valori: un valore booleano e un valore reale (*ActValue*). Secondo tale convenzione le variabili booleane hanno un valore reale ed un valore booleano ovviamente coincidenti, mentre le variabili intere avevano

valore reale intero e valore booleano True se e soltanto se esse avevano raggiunto la soglia fissata. Ciò violava però lo standard IEC, secondo il quale i contatori sono blocchi funzionali veri e propri e invece i confronti tra variabili numeriche vanno esplicitamente codificati. Dovendosi definire un terzo tipo di dati si è deciso di rivedere il sistema delle variabili per eliminare questa ambiguità semantica, resa ancora più evidente dall'introduzione di confronti ed operazioni aritmetiche nell'SFC. Non si è potuto rimuovere l'intero schema del doppio valore, in quanto troppi elementi di UniSim facevano riferimento ai metodi dell'uno o dell'altro tipo e si è deciso pertanto per una riduzione del sistema in modo che il valore booleano delle variabili non booleane sia sempre False. Ciò consente una più facile e più palese migrazione verso il nuovo schema (transizioni basate sull'uso di contatori falliranno sempre, e sarà pertanto evidente all'utente che il proprio programma va rivisto).

L'ambiente .net, ed il linguaggio VB.net, mettono a disposizione un vasto insieme di operatori per la conversione di ogni tipo di dato da e verso gli altri, ed in particolare per il passaggio da stringhe a numeri. Tali funzionalità soffrono però di una dipendenza dalla *locale* o *culture* nella terminologia Microsoft [4,5]. La *locale* specifica come vadano formattati diversi elementi dipendenti dalla lingua: giorni della settimana, date, orari ed appunto numeri. Mentre gli interi vengono trascritti nello stesso modo in tutto il mondo occidentale e non solo, per i numeri floating-point vi sono differenze nell'uso dei separatori (, in Europa e . negli USA ad esempio). Per evitare che i progetti UniSim possano dipendere dalla locale impostata dall'utente (un problema comune a un po' tutti i linguaggi ed ambienti di programmazione) si segue l'impostazione standard, analoga a quella della cosidetta locale POSIX C, che il separatore da usare sia sempre il punto fermo [12]. Per rispettare questo vincolo è necessario forzare gli operatori di conversione del sistema di programmazione ad usare la locale opportuna, che nella terminologia dell'ambiente è detta *Invariant Culture*. A tal fine, la classe BaseVariable include i due nuovi metodi ValueToUniversalString() e InitialValueToUniversalString() che convertono in stringa rispettivamente l'*ActValue* e l'InitialValue usando però la *culture* opportuna, che potrebbe o meno coincidere con quella in uso dal sistema operativo. Ovviamente, il valore booleano sarà sempre e solo true o false e pertanto non sarà influenzato dalla locale corrente (il .net fortunatamente non traduce i nomi true o false nella lingua dell'utente).

Una volta effettuate queste modifiche, e sostituite nei punti opportuni del codice le

chiamate precedenti con quelle nuove (lavoro effettuato con un estensivo testing oltre che usando gli strumenti di ricerca testuale forniti dall'ambiente Visual Studio), il lavoro principale è consistito nel riadattare l'interfaccia grafica e la logica applicativa già esistente per includere gli elementi necessari all'interazione con il nuovo tipo di dati.

Nel caso della finestra di creazione variabili l'unico lavoro necessario è stato inserire un nuovo valore in un elenco di tipi di dati ammessi, in quanto la finestra di creazione variabili non svolge in sé alcun lavoro di validazione dell'input utente e quindi non ha bisogno di conoscere le caratteristiche dei vari tipi di dati.

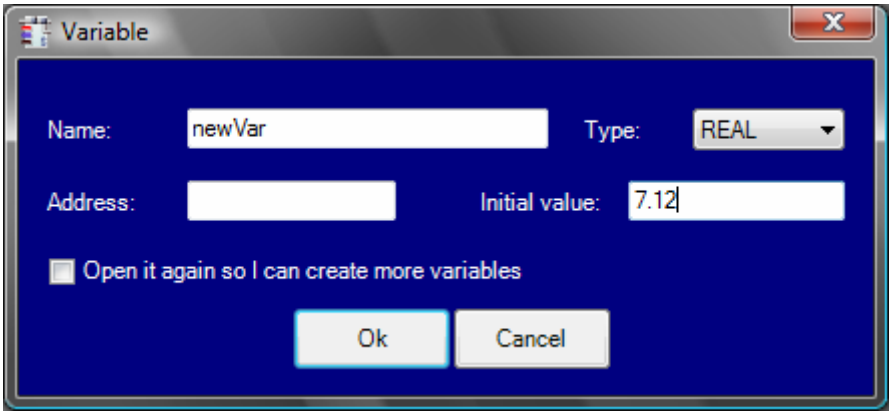

Figura 7. La finestra di creazione variabili

Più complesso è risultato il caso della finestra di Lista variabili, la quale si basa su una stretta interazione con i vari tipi di variabile e ha bisogno di conoscere più dettagliatamente la natura dei dati contenuti in ciascuna variabile. L'architettura di tale monitor è risultata suddivisa in due parti: un contenitore e un elemento contenuto il quale, esso soltanto, incapsula la logica di funzionamento di ciascun controllo di gestione e monitoraggio variabili. Gli interventi su di esso necessari sono consistiti nell'estendere il già esistente codice per la gestione delle variabili intere (nessun monitoraggio e comandi per le operazioni increment, decrement e reset) alle variabili REAL. Si è scelto di esporre una interfaccia analoga a quella del tipo INT dalla variabile verso il componente, salvo modificare gli incrementi e decrementi in ragione di 0.5 invece che 1 per ogni step.

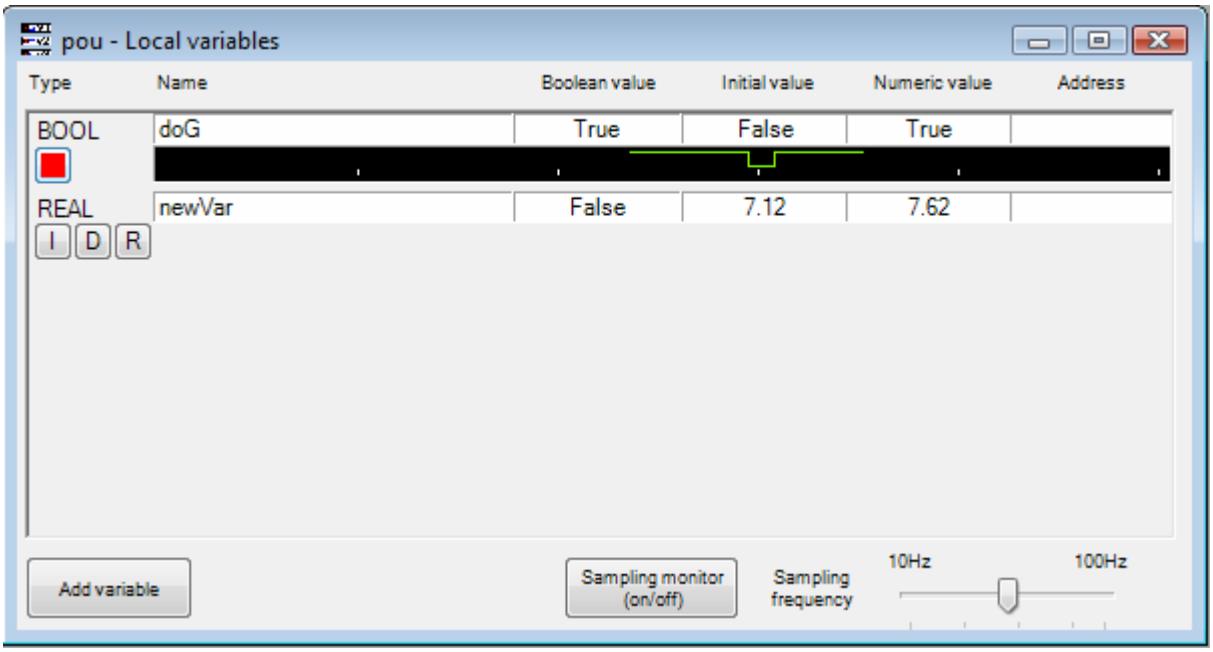

Figura 8. La finestra di elenco variabili

Una volta riadattata l'interfaccia grafica, si è proceduto rapidamente ad estendere la classe VariablesManager. Tale classe, la cui utilità è stata molto ampliata nel corso di questo elaborato, è un repository di informazioni sui tipi di dati di UniSim, ed è concepito per restituire su richiesta informazioni quali il valore iniziale predefinito di ogni tipo di dato di UniSim, e per validare un valore rispetto al tipo di dati previsto (una interrogazione che conferma potersi procedere con l'assegnamento oppure doversi generare un errore). Infatti, mentre le operazioni garantiscono un valore consistente alle variabili (non esiste un modo, ad esempio, per assegnare una stringa ad un numero in virgola mobile), le inizializzazioni, in quanto provengono da valori digitati manualmente dall'utente vanno validate contro possibili errori occasionali o intenzionali che siano. Garantite le precondizioni valide, e garantita la stabilità rispetto al tipo di ogni operazione, si garantisce infatti il sistema di sicurezza dei tipi per l'intero ambiente UniSim.

Estensione delle operazioni

Si è proceduto ad aggiungere il tipo di dati REAL quando già era completata la parte di interprete FBD e la gestione dei blocchi aritmetici in LD. Ci si è trovati pertanto, sia in questo caso, sia nel caso seguente della estensione dell'SFC, a dover adattare un software esistente ad un mutato insieme di requisiti piuttosto che a

produrre una soluzione originale ex novo. Ovviamente, tra i requisiti delle operazioni vi era quello del casting verso il tipo di dati più ampio necessario, ma non più ampio del necessario. Ad esempio, si desidera che una somma tra *n* variabili tutte di tipo INT dia una uscita di tipo INT, ma che se anche una sola di esse è di tipo REAL l'uscita sia di tipo REAL (salvo ovviamente downcasting in uscita). L'FBD (e il meccanismo di blocchi in LD) di UniSim sono da questo punto di vista permissivi, consentendo di intrecciare senza bisogno di passi intermedi di conversione le variabili dei vari tipi di dati. Ovviamente, il rispetto della semantica delle operazioni ricade sull'utente, in quanto, come verrà analizzato nei capitoli successivi, gli editor non cercano neanche di validare la semantica delle operazioni, fedeli ad un principio di valutazione *lazy* di ogni parte del programma.

La parte più delicata di questa estensione è stata sicuramente la parte SFC. A differenza del supporto LD e FBD, che interagisce non già con le variabili ma con i valori da queste assunti, il codice aritmetico e di confronto dell'SFC cerca di accedere direttamente alle variabili, e ovviamente partendo dall'assunzione che il tipo REAL non esista, effettua cast verso IntegerVariable in tutta sicurezza. Una volta aggiunto un terzo tipo di dati, i risultati sono ovviamente "catastrofici" ed è stato necessario rielaborare il codice per gestire il terzo tipo di dati, in particolare per le espressioni booleane si è lasciato che gli operatori di confronto overloaded del Visual Basic vengano automaticamente risolti sul tipo di dati più opportuno: Integer o Double che sia.

Per quanto riguarda gli operatori aritmetici vi erano due strade da seguire: gestire il tipo di dati del risultato e degli operandi in base ad una tabella di possibili casi (Integer/Integer, Integer/Double, …) oppure scegliere l'opzione meno restrittiva, Double in questo caso, e fare upcasting degli operandi al tipo Double. Si è scelta questa strada in base alle seguenti considerazioni:

- gli eventuali errori ed arrotondamenti introdotti con l'aritmetica in virgola mobile sono nei casi pratici molto piccoli, tanto che una eventuale riconversione in intero (nel caso il calcolo fosse tra valori interi) sarà corretta
- in caso di numeri interi, l'eventuale cancellazione catastrofica relativa a valori molto prossimi darà, riconvertita in intero, il valore zero, cioè il valore esatto
- in caso uno o entrambi i valori operandi siano in virgola mobile, il risultato dovrà necessariamente essere espresso in virgola mobile

- le operazioni aritmetiche sono di tipo impulsivo, quindi l'overhead prestazionale è ridotto

In base a queste motivazioni [13], si è scelto di procedere nel modo più semplice ma allo stesso tempo elegante possibile: in luogo di una lookup table che discriminasse i tipi di operandi e risultati, si è scelto il tipo meno restrittivo e si sono effettuati in maniera trasparente gli upcasting necessari (in effetti, nulla nella classe ArithmeticExpression lascia trasparire l'esistenza di un casting, che verrà effettuato automaticamente dal runtime se e quando necessario). Una eventuale riconversione in intero sarà anch'essa trasparente ed effettuata nel momento dell'assegnazione del risultato alla variabile di destinazione del calcolo.

2.3 Supporto per le POUs function

Come già indicato in precedenza, lo standard IEC su cui è basato UniSim prevede che vi siano tre tipi di unità organizzative di programma: *program*, *function* e *function block*. In questo elaborato si è aggiunto il supporto per la gestione delle POUs function e per la possibilità di usarle in maniera analoga alle funzioni predefinite fornite dall'ambiente. Tale supporto, che sarà approfondito nei capitoli relativi sia al LD che all'FBD, i due linguaggi che lo prevedono (è stato omesso l'SFC per ragioni di *compliance* allo standard IEC), utilizzando un blocco in grado di valutare una POU, secondo la seguente accezione:

1. passaggio degli ingressi alla POU
2. esecuzione di un ciclo di scansione della POU
3. lettura e restituzione al chiamante dell'uscita

Usando i meccanismi forniti dallo standard per separare ingressi da uscite (uscita nel caso di una funzione, in quanto lo standard prevede che le funzioni abbiano una ed una sola uscita, eventualmente strutturata ma unica) è facile procedere in questo modo, in quanto la lista di variabili d'ingresso fornisce tutti e soli gli ingressi (in tal senso può essere considerata la firma della function) e l'unica uscita, tra l'altro per convenzione chiamata sempre OUT in UniSim, è sicuramente presente nella lista delle variabili d'uscita. Sia nell'FBD che nel LD, viene utilizzato un unico blocco per valutare tutte le function, essendo unico l'algoritmo, e l'esecuzione della function vera e propria demandata alla capacità del singolo oggetto POU di eseguire se stesso.

```
Public Overrides Function Calculate() As Object
        m_pou.ExecuteInit()
        Dim vars As VariablesList = m_pou.PouInterface.inputVars
        vars.Reset()
        Dim i As Integer = 0
        Dim max As Integer = Math.Min(Me.Count, vars.Count)
        While i < max
                vars(i).SetActValue(Me(i).GetNodeValue())
                i += 1
        End While
        m_pou.ExecuteScanCycle()
        Return  _
        m_pou.PouInterface.outputVars.FindVariableByName("OUT").ReadActValue()
End Function
```

Questo codice non supporta la ricorsione nelle chiamate di funzione ma dato che lo standard non consente di scrivere funzioni ricorsive nei linguaggi grafici la limitazione è non solo positiva ma, anzi, andrebbe diversamente gestita se l'algoritmo in sé non la impedisse.

Ogni blocco ha un tipo univoco che viene stabilito al momento della sua creazione e che coincide con la funzione da esso espletata. Al fine di consentire che tutte le POUs vengano collegate all'unico tipo di blocco che esegue l'algoritmo descritto in precedenza si usa un caso di fallback nella routine che fa corrispondere i nomi dei blocchi alle implementazioni, e si stabilisce di effettuare la ricerca del nome che non può essere risolto tra le funzioni predefinite attraverso l'elenco di tutte le POUs presenti nel progetto:

```
Public Shared Function CreateBlock(ByVal BlockName As String, _
        ByVal plist As Pous) As BlockFBDTreeNode

        Select Case BlockName

                Case "AND"
                        Return New AndFBDBlock()

                Case "NAND"
                        Return New NandFBDBlock()

                […] omissis, un elenco di casi simili ai due precedenti

                Case Else
                        ' Promemoria: se abbiamo aggiunto qualche blocco lì e non qui...
                        If Array.IndexOf(ValidBlocks(plist), BlockName) >= 0 Then
                                ' ...cerca tra le POU per prima cosa...
                                If plist.FindpouByName(BlockName) IsNot Nothing Then _
```

```
                    Return New
                POUBoundFBDBlock(plist.FindpouByName(BlockName))
                ' ...poi dai una eccezione
                Throw New BlockNotBoundToClassException(BlockName)
            Else
                ' altrimenti ritorna Nothing, non dovremmo neanche passare
            questo BlockName
                Return Nothing
            End If

        End Select
    End Function
```

2.4 La traduzione SFC → Ladder

Il linguaggio SFC è inteso quale formalismo per la descrizione, la modellazione, e la programmazione di sistemi per loro natura sintetizzabili quali sistemi ad eventi discreti. Molti PLC non dispongono però di un supporto nativo per l'SFC, e necessitano di una fase di traduzione di questo verso altri linguaggi, il principale dei quali è il LD. L'SFC è definito, oltre che in termini di fasi e transizioni, anche tramite le opportune equazioni booleane che modellano i suoi meccanismi d'evoluzione. Essendo il Ladder il linguaggio d'elezione per la descrizione di sistemi basati su equazioni booleane, solitamente in letteratura è dedicato ampio spazio all'operazione di conversione di una POU in linguaggio SFC in una in linguaggio LD [23]. In questa release, il tool UniSim include un supporto basilare per la traduzione dal linguaggio SFC al LD. Pur essendo UniSim in grado di eseguire l'SFC, la traduzione in Ladder ha una valenza didattica di per sé, in quanto illustra a "basso livello" il meccanismo di funzionamento del linguaggio SFC. Il meccanismo usato da UniSim per effettuare la traduzione si basa su un modulo che scompone in diversi passaggi logici l'operazione. Tali passaggi sono sostanzialmente equivalenti a quanto l'utente fa per creare un programma in SFC: creazione delle fasi, impostazione delle fasi iniziali, aggiunta delle azioni, aggiunta delle transizioni. Le limitazioni del modulo di traduzione SFC2Ladder, che saranno dettagliatamente descritte e spiegate nel capitolo 5 di questo lavoro di tesi, sono sostanzialmente nella gestione delle azioni e delle transizioni. In particolare, al fine di semplificare la creazione della feature si è trascurato di gestire l'aritmetica, sia nelle azioni che nelle transizioni, lasciando questo lavoro ad un futuro

implementatore che potrà estendere il modulo di traduzione. Mancando per ora il LD di UniSim di un supporto per i timer, sono anche omesse le funzioni di traduzione delle condizioni temporali nelle transizioni. Nondimeno, come sarà successivamente analizzato, già così il traduttore è sufficientemente potente per un'ampia gamma di usi, come esemplificato dalle due figure seguenti

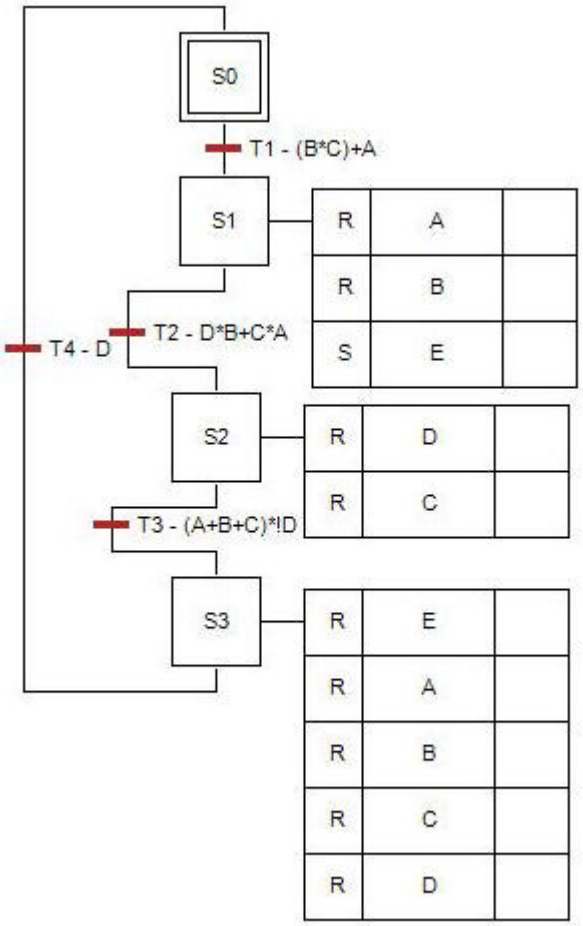

Figura 9. Una POU SFC scritta con UniSim

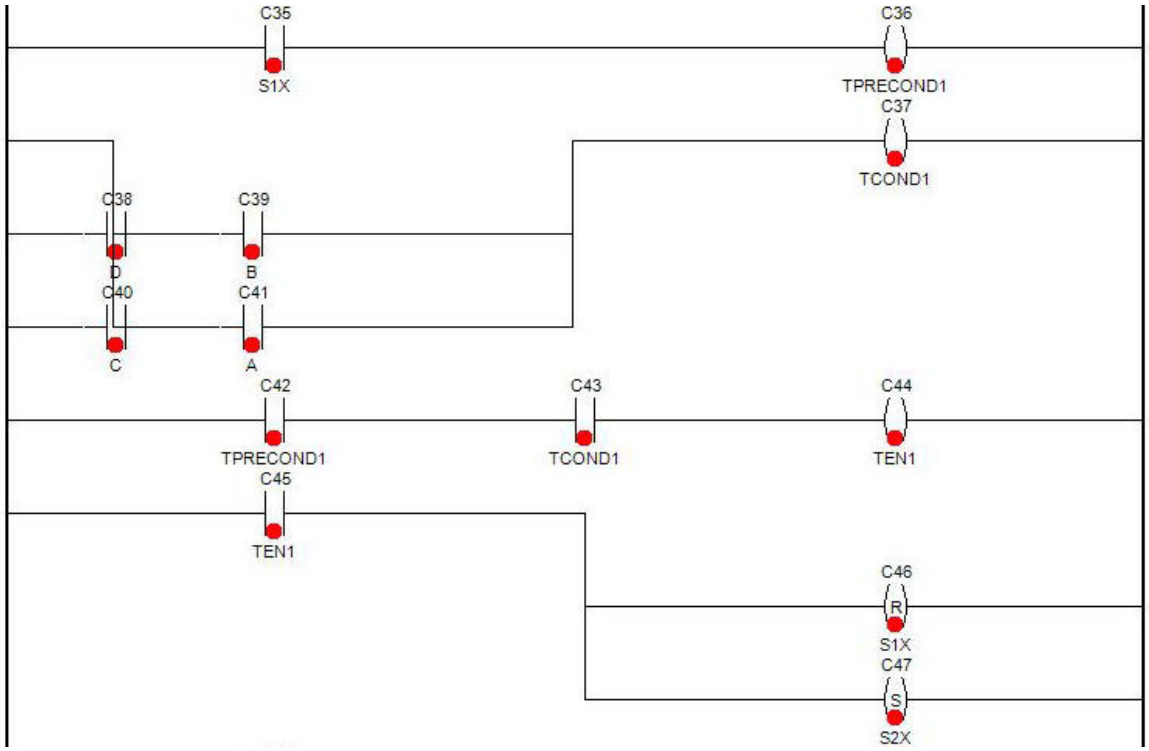

Figura 10. Una parte della traduzione in LD della POU di figura 6 (relativa alla transizione S1 → S2)

2.5 Conclusioni

Questo lavoro di tesi ha esteso molte parti dell'ambiente UniSim: il supporto per due nuovi linguaggi, il supporto per un nuovo tipo di dati e la possibilità di tradurre l'SFC in LD. Spunti di ulteriore innovazione sono già presenti in alcune di queste nuove funzionalità, principalmente nel traduttore SFC → Ladder, che è largamente incompleto, per quanto le basi algoritmiche siano sufficientemente solide da permettere una facile estensione. Esistono ancora diversi tipi di dato previsto dallo standard ma non implementati da UniSim, principalmente quelli temporali e strutturati, che sono di grande importanza nella programmazione, specialmente allorquando si volesse introdurre il linguaggio ST nell'ambiente. La definizione di blocchi funzionali, e l'estensione dell'SFC per supportare azioni definite da reti di tipo FBD sono sicuramente tra i principali *extension points* presenti in UniSim ad oggi. Ed ovviamente, infine, come già detto in precedenza, le specifiche sono la parte più mutevole di un qualsiasi prodotto software, e UniSim non farà sicuramente

eccezione. L'attesa è che i requisiti del programma possano cambiare enormemente nel corso del tempo, anche grazie all'utilizzo da parte di una ricca community di utenti, la quale sicuramente individuerà errori, imperfezioni, omissioni e, principalmente, nuove ed utili funzionalità.

Il fatto, infine, che UniSim sia open source sotto licenza GPL [14] è sicuramente un incentivo per chiunque, dotato e motivato, voglia entrare nel programma, capirlo e migliorarlo, e questo rende virtualmente illimitate le possibilità di evoluzione del tool [15].

Capitolo 3
L'interprete LD

Una delle principali attività di questo elaborato di tesi consiste nella creazione di un editor e di un simulatore per il linguaggio a contatti. Tale linguaggio, tra i cinque dello standard IEC, è il linguaggio d'elezione per programmi basati su operazioni logiche e manipolazioni di dati a livello dei singoli bit, pur non mancando della possibilità di svolgere altri tipi di operazione, grazie alla possibilità prevista dallo standard medesimo di incorporare delle reti funzionali (simili all'FBD) nelle POUs. Il lavoro svolto su UniSim è consistito in una prima fase in cui si è creato un editor ed un interprete in grado di generare ed eseguire programmi composti di sole bobine e contatti, di tutti i tipi previsti dallo standard (salvo quelle a ritenzione, inutili in un ambiente di simulazione). Terminata questa versione basilare del supporto LD, si è potuto aggiungere in maniera semplice ed elegante il supporto per i blocchi funzionali, anche basandosi in parte sul codice già previsto per l'implementazione del Functional Block Diagram, anch'essa parte del lavoro discusso in questo elaborato.

3.1 Presentazione del problema

Il problema che ci si accinge a risolvere, già brevemente presentato nel capitolo precedente, è l'interpretazione di un programma scritto in linguaggio a contatti. Per interpretazione devesi intendere la valutazione di ogni rung del programma, e la

scrittura corretta delle uscite nelle variabili apposite. Si vuole inoltre rispettare l'algoritmo a copia massiva degli ingressi e a ciclo di scansione, tipico e caratterizzante dei controllori a logica programmabile rispetto ai calcolatori di uso generale basati sull'algoritmo di von Neumann [16,17,20].

Lo standard IEC 61131-3 definisce i vari tipi di bobina e contatto previsti in un programma Ladder, la loro semantica, e l'interpretazione dei collegamenti. L'ordine di valutazione definito dallo standard, e rispettato da UniSim, è dall'alto verso il basso (salvo elementi di controllo di flusso che UniSim non prevede allo stato attuale). Lo standard definisce anche la rappresentazione grafica degli elementi facendo uso di caratteri, e UniSim in quanto strumento grafico raffigura tali elementi con un disegno rispondente alla natura e all'intenzione di quanto indicato dal comitato di standardizzazione.

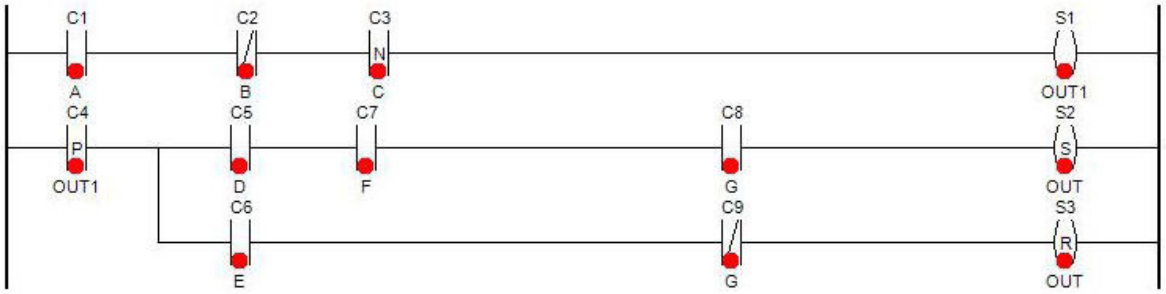

Figura 11. Un semplice programma in linguaggio a contatti prodotto con UniSim

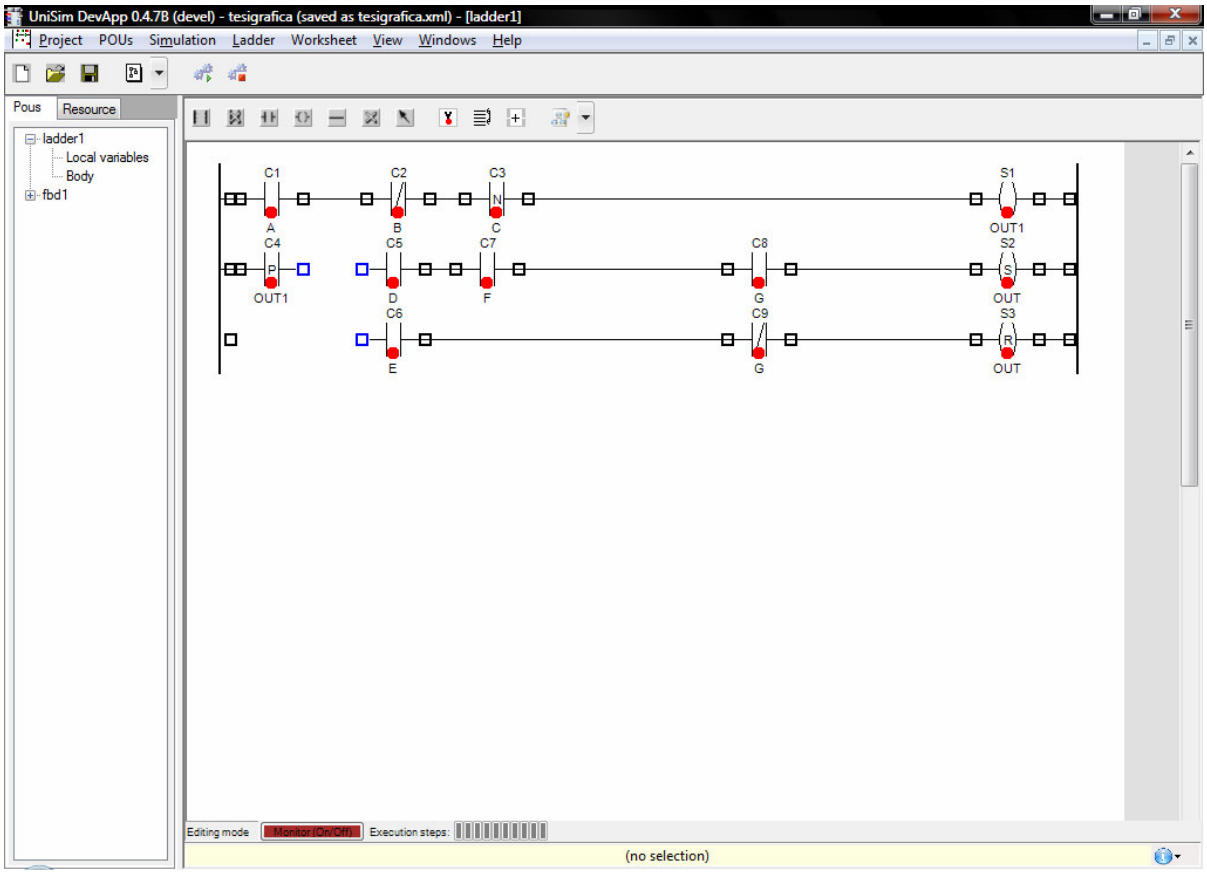

Figura 12. L'ambiente di editing Ladder di UniSim

L'editor Ladder di UniSim fornisce strumenti per la creazione di contatti e bobine dei vari tipi proposti dallo standard, e per la loro connessione attraverso dei rettangoli associati ad ogni elemento grafico. A partire dagli elementi disposti su schermo dall'utente, e dai collegamenti da questi creati, l'editor crea e mantiene aggiornate due collezioni: GraphicalContactList e GraphicalConnectionList, rispettivamente l'elenco di tutti i contatti e bobine (il modello a oggetti, derivato da un prototipo preesistente, non distingue i due tipi di elemento usando diverse classi, ma attraverso un sistema di qualificatori numerici e di tipo stringa) e l'elenco di tutte le connessioni tra questi elementi, e tra questi e le alimentazioni destre e sinistre.
Lo spazio grafico di lavoro è dato dalla regione tra le alimentazioni destre e sinistre, e le varie "righe" di programma (dette rung) sono definite dalle alimentazioni destre. Una regola prevista in UniSim e che ad ogni alimentazione destra possa essere collegata non più di una bobina, e l'alimentazione destra darà luogo ad un rung se e solo se vi è una bobina ad essa collegata (questa regola non prevede il caso in cui il

rung includa una rete FBD, che sarà discusso in una sezione seguente del capitolo).

3.2 Analisi del problema

La prima considerazione possibile in un diagramma Ladder è che lo scopo di ogni istruzione è la scrittura di un certo valore sulla uscita, e che tale valore dipende:

- dai valori correnti e precedenti degli ingressi
- dai tipi di contatto a cui gli ingressi sono associati
- dal tipo di bobina a cui è associata l'uscita
- dal valore corrente dell'uscita

Se si tralascia momentaneamente il problema dei riconoscimenti di fronte, è banale vedere che una serie di contatti si converte in una AND delle variabili associate ad ognuno, eventualmente con dei termini negati per i contatti normalmente chiusi, e come i paralleli di contatti siano da convertirsi in una OR delle variabili associate, ancora usando ove occorra funzioni NOT. Essendo tale schema applicabile in maniera ricorsiva, il problema di calcolare il valore di un rung privo di riconoscitori di fronte è quindi equivalente ad una funzione composta di AND, OR e NOT. Per quanto riguarda i riconoscimenti di fronte, una breve analisi ci porterà a vedere che lo stesso tipo di logica è applicabile anche a questo caso, assumendo l'uso di memoria addizionale in quantità costante per ogni contatto e/o bobina a fronte.
I ricercatori [16] evidenziano come anche i contatti a fronte siano descrivibili, usando apposite variabili ausiliarie, quali semplici funzioni booleane. Si riporta, ad esempio, la descrizione della funzione logica associata ad un fronte di salita OUT=RisingFront(X):

X_n	X_{n+1}	OUT
0	0	0
0	1	1
1	0	0
1	1	0

Tale funzione corrisponde all'espressione $Not(X_n)*X_{n+1}$. Inoltre, se X_n è alta, senza bisogno di valutare X_{n+1} si può porre OUT al valore basso (questa ottimizzazione

viene usata nel codice di UniSim). Considerazioni duali si applicano ai fronti di discesa.

Si può procedere nella gestione delle bobine, considerando associata ad ogni bobina una variable v e una funzione $f(x)$ a valori booleani e con x e v parametri booleani, tale che detto r il valore dell'espressione logica formata dai contatti, il problema della valutazione di un rung si riduce a $v = f(r)$. È evidente come la funzione f dipenda dal tipo di bobina e possa in generale dipendere anche dal valore precedente dal rung o dal valore attuale della variabile v. Essendo banale ricavare entrambi questi valori con al massimo l'uso di una quantità costante di tempo e memoria, si vede facilmente che tali aggiunte non complicano realmente il modello del problema che si sta sviluppando, ed anzi questa situazione sembra confermare l'ipotesi iniziale di questa analisi [20,32].

Una volta che sia definito un blocco funzionale che, ricevendo in ingresso una variabile, sia in grado di riconoscere se il valore della stessa è cambiato o meno dal precedente ciclo di scansione, e pertanto sia un rilevatore di fronti (salita o discesa non cambia la natura fondamentale del discorso), e assegnata l'uscita di questo dispositivo ad una variabile d'appoggio temporaneo, è palese che l'intero problema della valutazione di un rung è stato ricondotto ad una opportuna funzione composta di AND, OR e NOT su variabili booleane, e che ogni contatto contribuisce con un termine alla funzione finale, ovvero con una costante o con una singola variabile booleana, o con il negato di una singola variabile booleana.

Un buon testo di algoritmica, o di analisi del problema della costruzione di compilatori [18,19] metterà inoltre in evidenza come, a valle di queste riduzioni effettuate, il problema della valutazione di un rung sia equivalente ad un problema di valutazione di una espressione, e quindi alla creazione ed alla riduzione di un parse tree. Ad ulteriore semplificazione del problema, si vede facilmente che i parse tree derivanti dal LD non hanno la stessa generalità che potrebbero avere nel caso di un compilatore di un linguaggio di programmazione general-purpose in quanto l'insieme di operatori è limitato e non estendibile da parte dell'utente.

Per quanto concerne la struttura formale di un programma Ladder così come prodotto dall'editor, esso può essere visto come un grafo orientato, con però alcune limitazioni:

- una bobina non può essere collegata ad un'altra bobina
- un'alimentazione destra può essere collegata a nulla o ad una sola bobina
- un contatto può essere collegato ad altri contatti o ad una alimentazione

sinistra

- le alimentazioni sinistre non possono essere direttamente collegate ad una alimentazione destra

Data l'ampia gamma di algoritmi su grafi esistenti, usare questa rappresentazione è sicuramente vantaggioso. L'analogia è semplificata dal fatto che l'editor, è importante ribadirlo, fornisce una lista di contatti (vertici) e di connessioni (archi). Esso, però, non fa nulla per limitare la gamma di grafi producibili a quelli teoricamente accettabili, fedele al principio di non valutare la correttezza semantica di quanto prodotto prima del momento dell'esecuzione, momento in cui la valutazione dello stato del grafo viene demandata ad un componente *client*: l'interprete, appunto. Si vede che i grafi validi avranno l'importante proprietà di originare ciascuno da un nodo ben preciso (una bobina) e di non contenere cicli (il feedback non è permesso, infatti, in Ladder, se non utilizzando la stessa variabile sia per un contatto che per una bobina).

Come ultima, ma non meno importante, premessa alla disamina dell'algoritmo usato, una variante di una visita in profondità di un grafo qui battezzata RtL, val la pena notare come UniSim includa un componente che, ricevuta in input una stringa che rappresenta una espressione booleana, sia in grado di generare il parse tree associato, e di valutare quest'ultimo sulla base dei valori correnti delle variabili di POU e di risorsa: BooleanExpression. È di quest'ultima che si è fatto uso, sfruttando quindi il codice preesistente in maniera elegante ed orientata agli oggetti, invece che riproducendo codice con meccanismi quali copia&incolla o, peggio, riscrivendo ex novo senza necessità un componente. In tal modo si è infatti potuto usufruire già out-of-the-box della fase di testing (molto ampia, in quanto la classe BooleanExpression è in uso dalla prima release del tool UniSim) effettuata sul componente preesistente.

3.3 Sviluppo della soluzione

Nel tentativo di produrre una soluzione informatica al problema della valutazione di un rung Ladder, si è iniziato analizzando il problema. Da tale analisi è scaturito un modello di esso sufficientemente semplificato da potersi facilmente tradurre in un algoritmo.

E' evidente, in base alle considerazioni precedentemente esposte che l'algoritmo

cercato debba, in qualche modo, attraversare il grafo formato dall'editor e ricavarne le espressioni booleane da poi valutare. Sono stati studiati, ed implementati e testati, due approcci: il LtR (left to right) e quello poi usato nella versione definitiva del tool: il RtL (right to left). Questi algoritmi sono stati così battezzati in base all'ordine di scansione rispetto alle alimentazioni: l'LtR parte dalle alimentazioni sinistre, l'RtL da quelle destre. Pur vincendo in semplicità, lo schema LtR è affetto, come si evidenzierà, da una erronea concezione di fondo che lo rende inadatto ad un uso reale.

Sia l'algoritmo LtR che quello RtL sono implementati a partire dall'insieme degli archi, o connessioni, vedendo i vertici o contatti/bobine quali dati addizionali del problema. Sarebbe stato possibile, in maniera del tutto analoga, una concezione in cui la bobina è l'elemento in grado di costruire il rung ad essa associato, e le connessioni dati del problema, ma il doversi concepire il problema come un problema di attraversamento ha portato in modo naturale a collocare la logica come parte dell'insieme degli archi. Inoltre, entrambi gli algoritmi prendono come parametro un alimentazione (destra o sinistra che sia), e producono un oggetto di tipo Rung che è la rappresentazione complessiva di una bobina, l'espressione logica ad essa associata (e l'eventuale rete di tipo FBD come si approfondirà nel seguito della trattazione).

L'algoritmo LtR

L'invocazione dell'algoritmo LtR avviene attraverso la chiamata

```
Public Function BuildExpressionLtR(ByVal root As GraphicalLeftRail) As Rung
```

Tale chiamata esegue il seguente corpo di codice

```
Public Function BuildExpressionLtR(ByVal root As GraphicalLeftRail) As Rung
    Dim ret As New Rung()
    Dim expr As String = MakeExpressionPieceLtR(root, ret)
    expr = expr.TrimEnd("*")
    ret.Expression = expr
    Return ret
End Function
```

Il metodo MakeExpressionPieceLtR è il corpo, ricorsivo, dell'algoritmo. Esso riceve in input un oggetto di tipo BaseGraphicalContact e un puntatore (o meglio, riferimento) ad un Rung, e restituisce l'espressione in formato stringa associato al Rung. Riportare il codice dell'algoritmo sarà di ausilio alla comprensione dei passaggi ed, anche, del successivo algoritmo RtL

```
Private Function MakeExpressionPieceLtR(ByVal root As BaseGraphicalContact, _
```

```
        ByRef rng As Rung) As String
        Dim nextOnes As GraphicalContactList = FindRightOf(root)
        Dim myself As String = ""
        If TypeOf (root) Is GraphicalLeftRail Or TypeOf (root) Is GraphicalRightRail Then
                myself = ""
        Else
                If root.IsContact Then
                        myself = root.GetTerm + "*"
                ElseIf root.IsCoil Then
                        rng.AddCoil(root)
                End If
        End If
        ' Se ci sono nodi dopo questo...e non siamo in una bobina...
        If Not IsNothing(nextOnes) AndAlso Not (root.IsCoil) Then
        ' esegui la scansione del resto del rung
                If nextOnes.Count = 1 Then
                        myself = myself + MakeExpressionPieceLtR(nextOnes(0), rng)
                Else
                        ' Verifica se il resto del rung ha qualcosa da aggiungere all'espressione
                        Dim inneroutput As String = ""
                        For Each n As BaseGraphicalContact In nextOnes
                                inneroutput += MakeExpressionPieceLtR(n, rng)
                                inneroutput = inneroutput.TrimEnd("*") + "+"
                        Next
                        ' Una stringa di nextOnes.Count segni +
                        Dim comparison As New String("+", nextOnes.Count)
                        ' Se non ci sono solo i segni + nella stringa inneroutput
                        ' cioè se almeno una MakeExpressionPiece() ha aggiunto qualcosa
                         If inneroutput <> comparison Then _
                                myself = myself + "(" + inneroutput.TrimEnd("+") + ")"
                 End If
         End If
         Return myself
End Function
```

Il metodo FindRightOf() scansiona l'elenco delle connessioni alla ricerca di tutti i contatti o bobine alla destra del suo parametro (ovvero, di tutti quelli che vedono il contatto specificato come sorgente di una connessione).

Come secondo passo, l'LtR crea la parte di espressione associata al contatto specificato. Si ricorda dalla precedente disamina, come si è visto potersi ridurre la valutazione del rung ad una funzione booleana in cui ad ogni contatto è associato un singolo termine. Questo passaggio teorico viene qui usato per ottenere il "pezzo d'espressione", da cui il nome MakeExpressionPiece() della funzione.

Le alimentazioni non hanno un'espressione logica a loro associata, mentre ce l'hanno i contatti, ed essa viene valutata e restituita dal contatto stesso attraverso il suo metodo GetTerm(). Si noti l'aggiunta di un segno "*" in coda al termine stesso,

esso viene aggiunto in quanto si assume che ogni contatto verrà messo in AND per tutto ciò che lo segue nel suo percorso di valutazione verso le bobine (e tale assunzione è confermata dalla semantica dell'LD). Potrebbero, e spesso ci sono, dei segni "*" spuri che vengono rimossi da una fase di trimming conclusiva.

Le bobine non hanno termini associati, ma vengono aggiunte ad un elenco di bobine collegate al rung, tutte le quali riceveranno come ingresso alla loro funzione *f*, il valore calcolato per il rung. Ciò porta ad un approccio in cui il contatto è visto come elemento passivo, in grado di "trasferire" verso uno strato software richiedente il suo proprio pezzo d'espressione, ma non di computare il proprio valore o il flusso d'energia attraverso sé, mentre la bobina è un elemento attivo che, ricevuto da uno strato software il valore alla sua sinistra, è in grado di computare il flusso d'energia verso di sé e quindi il proprio valore. Questa dualità trova origine e riscontro nel modo in cui si è associata ad ogni bobina una funzione *f* caratteristica, mentre ci si è limitati a descrivere i contatti in termini di espressioni booleane da parserizzare.

Se ci sono vertici successivi nel grafo, e il nodo corrente del grafo non è una bobina (nel qual caso sarebbe seguita solo da un'alimentazione destra in un rung ben formato), si procede a valutare il contributo dei nodi successori all'espressione.

In particolare, si ottimizza il caso di un unico nodo successore, evitando la creazione di parentesi non necessarie e concatenando direttamente il "pezzo d'espressione" del successore a quello del nodo corrente. Se invece un nodo ha più successori, i loro termini associati vengono posti in OR, e il risultato della OR in AND con il nodo corrente.

Se però tutti i nodi successivi sono bobine, non vi sarà alcun contributo aggiuntivo, e quindi l'unico risultato della concatenazione di stringhe sarà un insieme di segni "+", tanti quanti i nodi successivi. In tal caso, si salta la fase di concatenazione.

Applicando ricorsivamente l'algoritmo, si vede che si può ottenere la descrizione di un rung. Ma si vede anche, e lo si farà con un esempio, che questo algoritmo fallisce facilmente e pertanto è semplicemente utile quale esempio non riuscito di traduzione. Il problema della correttezza dell'RtL, l'algoritmo duale, verrà esaminato in appropriata sede.

Si consideri il seguente, semplice, programma

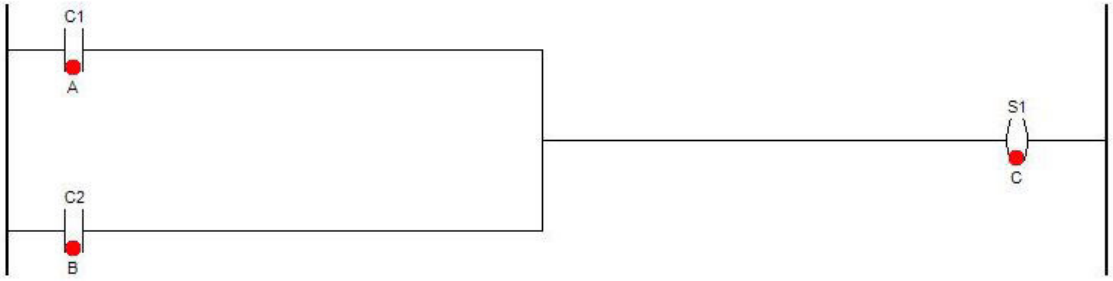

Figura 13. Un programma con cui l'algoritmo LtR fallisce

È evidente, per chiunque conosca anche solo sommariamente il linguaggio a contatti, che tale programma deve leggersi come $C = A$ OR B. Si ripercorrerà ora l'algoritmo LtR assumendo questo programma come input. Primariamente, esso cerca le alimentazioni sinistre e ne trova 3: trascuriamo quella centrale a cui nulla è collegato. La prima alimentazione è collegata ad un contatto, il cui termine associato è "A*", e che ha come unico successore la bobina C. Il primo rung prodotto dall'algoritmo è $C = A$. In maniera del tutto analoga, dalla terza alimentazione, percorrendo la successione dei nodi nel grafo, si ricava $C = B$. Da tali assegnazioni conflittuali, si vede che banalmente l'ultima delle due ad essere eseguita, ovvero la seconda, darà il valore alla variabile C, che pertanto sarà uguale a B. Si vede che l'attribuzione A = true e B = false è sufficiente a dimostrare che i due programmi non sono equivalenti e che, quindi, l'algoritmo LtR è sbagliato.

Nondimeno, esso è un valido primo tentativo, ed è la base dell'algoritmo realmente usato da UniSim. Il suo limite intrinseco è il fatto di partire dalle alimentazioni sinistre, e quindi dai contatti, invece che da quelle destre, cioè sostanzialmente dalle bobine.

L'algoritmo RtL

L'invocazione dell'algoritmo RtL avviene attraverso la chiamata

```
Public Function BuildExpressionRtL(ByVal root As GraphicalRightRail) As Rung
```

Tale chiamata esegue il seguente corpo di codice

```
Public Function BuildExpressionRtL(ByVal root As GraphicalRightRail) As Rung
    Dim ret As New Rung()
    Dim expr As String = MakeExpressionPieceRtL(root, ret)
    expr = expr.TrimStart("*")
    expr = Globals.TrimDuplicates(expr, "+"c)
    ret.Expression = expr
```

```
    Return ret
End Function
```

Il metodo MakeExpressionPieceRtL è il corpo, ricorsivo, dell'algoritmo. Esso riceve in input un oggetto di tipo BaseGraphicalContact e un puntatore (o meglio, riferimento) ad un Rung, e restituisce l'espressione in formato stringa associato al Rung. Il seguente codice costituisce il corpo dell'algoritmo.

```
Private Function MakeExpressionPieceRtL(ByVal root As BaseGraphicalContact, _
    ByRef rng As Rung) As String
    Dim prevOnes As GraphicalContactList = FindLeftOf(root)
    Dim myself As String = ""
    ' Se siamo in una pista di alimentazione, ignora...
    If TypeOf (root) Is GraphicalLeftRail Or TypeOf (root) Is GraphicalRightRail Then
        myself = ""
    Else
        ' Per i contatti, calcola il termine e inseriscilo
        If root.IsContact Then
            myself = "*" + root.GetTerm()
        ' per le bobine, aggiungile all'elenco delle uscite
        ElseIf root.IsCoil Then
            rng.AddCoil(root)
        ' per i blocchi, crea l'albero FBD associato
        ElseIf root.IsBlock Then
            Dim ldTree As LDTree = BuildTree(root)
        ' Se non c'è l'albero FBD non aggiungerlo
        ' Per come è congegnato il metodo BuildTree() verranno ignorati i blocchi che
    fanno
        ' parte di altri alberi FBD (in particolare quelli che non hanno una variabile
    come
        ' uscita) e quindi il loro attraversamento non influirà sul rung
            If ldTree IsNot Nothing Then rng.AddFBDTree(ldTree)
        End If
    End If
    ' Se ci sono nodi prima di questo...
    If Not IsNothing(prevOnes) Then
        ' se ce n'è uno solo bisogna farne la And (il segno "*" è stato inserito
        ' prima se necessario)
        If prevOnes.Count = 1 Then
            myself = MakeExpressionPieceRtL(prevOnes(0), rng) + myself
        Else
        ' Ci sono più nodi precedenti, bisogna farne una Or e poi metterla
        ' in And (come prima, il "*" se necessario è già presente)
            Dim inneroutput As String = ""
            For Each n As BaseGraphicalContact In prevOnes
                inneroutput = MakeExpressionPieceRtL(n, rng) + inneroutput
                inneroutput = "+" + inneroutput.TrimStart("*")
            Next
            ' Una stringa di nextOnes.Count segni +
```

```
                    Dim comparison As New String("+", prevOnes.Count)
                    ' Se non ci sono solo i segni più nella stringa inneroutput
                    ' cioè se almeno una MakeExpressionPiece() ha aggiunto qualcosa
                    If inneroutput <> comparison Then
                        ' aggiungi la Or all'espressione totale
                        myself = "(" + inneroutput.TrimStart("+") + ")" + myself
                    End If
                End If
            End If
            Return myself
End Function
```

Un occhio accorto noterà che, salvo la parte di gestione degli alberi di tipo FBD (usati quale struttura dati per gestire i blocchi aritmetici), l'algoritmo è simmetrico e duale rispetto all'LtR discusso precedentemente. La prima specularità è nel fatto che il segno "*" per i contatti viene premesso piuttosto che aggiunto in coda al termine associato al contatto. Può sembrare un approccio meno naturale, ma è ben giustificato se si immagina il percorso svolto dall'algoritmo attraverso il rung: esso costruisce l'espressione a ritroso e quindi i termini formalmente primi nel rung, vengono invece aggiunti per ultimi, e soprattutto inseriti in testa all'espressione sino a quel momento ottenuta, il che, sebbene non costituisca una dimostrazione formale, conforta tale modus operandi.

Al posto del metodo FindRightOf() si usa qui un metodo FindLeftOf() con semantica speculare: ricerca tutti i nodi che vedono il nodo corrente quale destinazione di una connessione. Si è rimossa inoltre la condizione di terminazione della ricorsione dovuta al trovarsi posizionati su una bobina. È infatti ovvio che l'algoritmo RtL termina il suo percorso su un contatto, prima di incontrare le alimentazioni sinistre. Non potendosi però discriminare in modo banale i contatti terminali (quelli cioè preceduti da sole alimentazioni), dai contatti collegati ad altri contatti, si sceglie di affidare la corretta produzione dell'espressione all'inserimento in testa di stringhe vuote all'incontrare un'alimentazione. Inoltre, essendo le alimentazioni sinistre non precedute da alcun altro nodo del grafo, si vede che i vari percorsi, per quanto numerosi, andranno a terminare tutti nelle alimentazioni sinistre, le quali non modificheranno la stringa di espressione e non invocheranno ulteriormente il metodo MakeExpressionPiece().

Al fine di esemplificare il procedimento seguito si riporta un breve programma, composto da un solo rung, e il parse tree equivalente. Nell'analisi del parse tree bisogna ricordare che, in base all'analisi svolta, i vari contatti e le loro interconnessioni vengono modellati tramite una espressione booleana in formato

stringa, ed è compito dell'interprete di espressioni convertire questa rappresentazione in un AST vero e proprio. Inoltre, l'indicazione relativa ad un fronte di salita, va ricordato, non è contenuta nell'espressione booleana ma nella logica dell'oggetto che rappresenta la bobina.

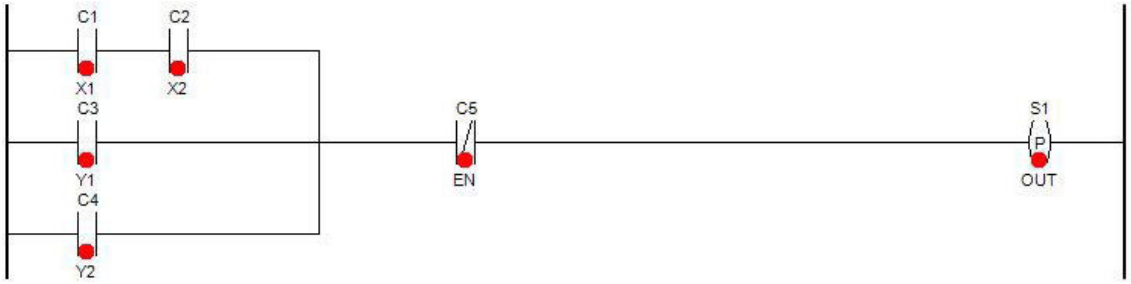

Figura 14. Un semplice programma

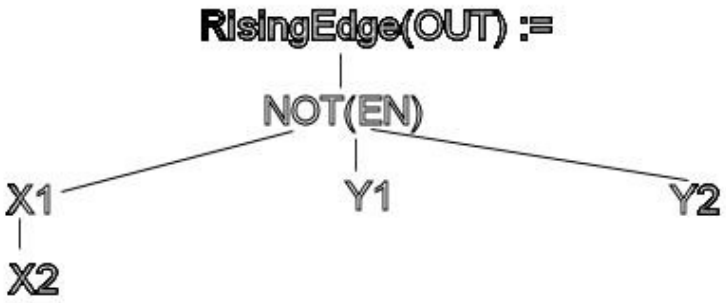

Figura 15. La conversione in un parse tree del programma di figura 4

Dualmente, al caso precedente, gli "+" vanno inseriti in testa, e i segni "*" spuri vanno eliminati anch'essi con un trimming all'inizio della stringa.

Il metodo TrimDuplicates è un banale automa a stati finiti [20] che, partendo da una stringa e da un carattere quali parametri, produce una stringa con gli stessi caratteri della sorgente, salvo che tutte le sequenze di qualsivoglia lunghezza del carattere dato vengono ridotte ad una sua singola istanza. Ad esempio, la stringa "a+++b+++++c+d+e" viene ridotta, indicano come argomento carattere il segno "+", al valore "a+b+c+d+e". Per comprendere il motivo di questa chiamata aggiuntiva, si consideri il seguente programma.

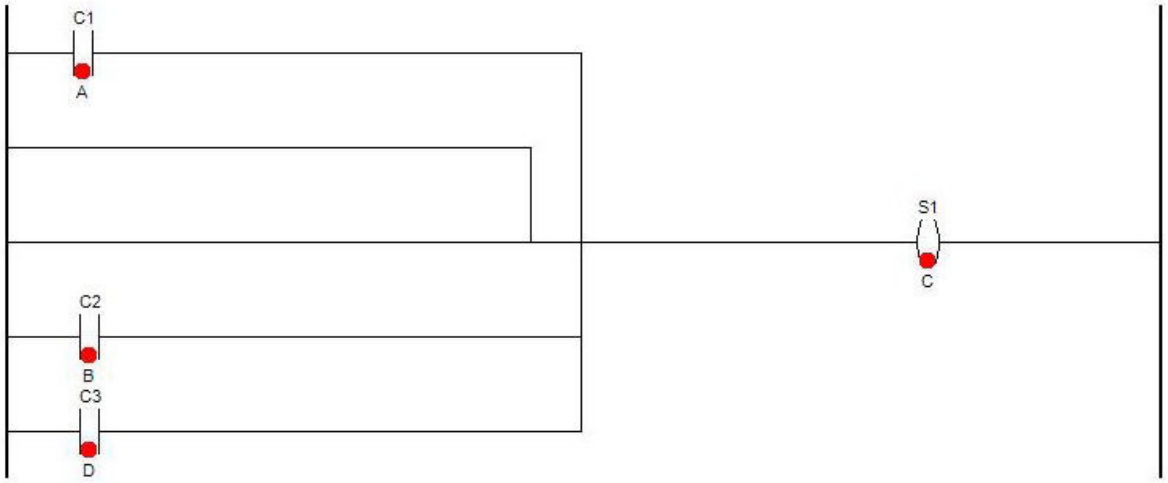

Figura 16. Un esempio del metodo TrimDuplicates()

Trovando a monte di sé un insieme di più connessioni, l'algoritmo RtL procede mettendo in OR tra loro tutti i percorsi. In questo programma, un percorso è semplicemente dato da "*A", due sono dati da stringhe vuote "", e gli ultimi due da "*B" e "*D". Mettendo il tutto insieme con gli opportuni segni di OR, il risultato del rung è $C = D + B +++ A$. Questi tre segni più, due dei quali dovuti ai collegamenti diretti tra la bobina *C* e le alimentazioni (un caso limite, ma non errato in sé), comportano la creazione di una espressione sbagliata. Il metodo TrimDuplicates si occupa di ridurre il tutto a $C = D + B + A$, come atteso.

La fase di esecuzione

Nella sezione precedente si è informalmente discusso l'algoritmo RtL e, pur senza formali dimostrazioni di correttezza, basandosi su una lunga fase di test oltre che sulla analisi teorica a monte, si è assunto che esso possa fornire un Rung valido per ogni bobina inserita nel diagramma (più formalmente, per ogni alimentazione destra preceduta da una bobina).

A valle di questa discussione, si può assumere che esista un semplice algoritmo, chiamiamolo GetRungs(), che dato un diagramma LD (cioè l'insieme dei contatti/bobine e l'insieme delle connessioni), restituisca tutti i Rung esistenti al suo interno sotto la forma già discussa. Essendo l'esecuzione di un programma in linguaggio a contatti definita come l'esecuzione di tutti i suoi rungs, dall'alto verso il basso, è evidente che, se esiste un algoritmo per eseguire un oggetto di tipo Rung, cioè scrivere sulla sua uscita una appropriata funzione dell'espressione logica a

monte e del valore corrente della uscita stessa, allora si è ricavato un interprete per il Ladder Diagram.

Tale algoritmo, fortunatamente, esiste, ed è anche semplice da concepire e codificare. Il suo tempo di esecuzione, se O(E) è il tempo richiesto per valutare l'espressione logica associata al rung è O(E + C) se C è il numero di bobine associate al Rung. Siccome, per regola, si è stabilito essere C = 1, il tempo richiesto è O(E).

L'algoritmo si basa sull'utilizzo dell'espressione logica ricavata dal rung per ottenere un valore unico, true o false, per tutta la rete logica. Una volta ottenuto questo valore, è semplice passarlo in ingresso ad un opportuno metodo della bobina, la quale non solo conosce il suo tipo (aperta, chiusa, latch, …) e quindi sa come utilizzare il valore ottenuto, ma può anche contenere (ed in effetti contiene) il valore precedente dell'intero rung quale dato aggiuntivo (e quindi con accesso a tempo costante), e sempre in tempo costante può ricavare se modificare, e se sì come, il valore della variabile associata. L'esistenza di un tale algoritmo per ogni singolo Rung, e la sua applicabilità a tutti i Rung di un diagramma, porta a concludere di aver così progettato e successivamente tradotto in codice un interprete per il linguaggio a contatti.

L'utilizzo di un *dictionary*, in cui si inseriscono le variabili da modificare come chiavi, e i nuovi valori di esse come valori, che per analogia con il linguaggio COBOL viene detto *working storage set*, garantisce che venga rispettata la logica a copia massiva, in cui solo dopo l'esecuzione di tutti i rung, i nuovi valori vengono effettivamente scritti nelle opportune locazioni.

3.4 Estensione della soluzione

Appena prodotta la soluzione sin qui descritta, ed in parallelo con lo sviluppo del supporto per l'FBD, è nata l'esigenza di aumentare le capacità del supporto LD di UniSim consentendo di far uso di variabili numeriche, intere o reali che siano, e di operatori su di esse, incorporando cioè parti di tipo funzionale nella struttura del linguaggio a contatti. Tale requisito è previsto e descritto, sebbene succintamente, dallo stesso standard IEC. Esso infatti definisce l'uso di funzioni in LD quale "analogo a quanto descritto per i linguaggi grafici in generale, fatte salve le seguenti eccezioni:

- Le connessioni reali ai parametri potranno opzionalmente essere mostrate scrivendo i dati o la variabile appropriati al di fuori del blocco, in posizione adiacente al nome del parametro formale nel blocco
- Per lo meno un input (EN) e un output (ENO) di tipo booleano saranno mostrati su ogni blocco per consentire il flusso di energia attraverso il blocco" [1].

I requisiti obbligatori e vincolanti sono che i blocchi funzione (da non confondersi con i blocchi funzionali così come definiti dallo standard quale tipo di POU), debbano potersi collegare al rung tramite ingressi di abilitazione, e restituiscano alla loro uscita logica, ENO, lo stesso valore ricevuto in ingresso, e che sia possibile collegare ingressi ed uscite addizionali relativi alla semantica specifica del blocco. Facendo perno sulla voce dello standard che richiede ai blocchi un numero "*fixed*", fissato, di punti di collegamento, UniSim (sia nella implementazione Ladder che nella FBD) usa un solo punto di collegamento per i vari parametri e li discrimina in base all'ordine di collegamento al blocco (ovviamente, consultabile e modificabile attraverso apposita funzione dell'interfaccia utente).

Deve essere possibile collegare tra loro i vari blocchi funzionali, sia collegando l'ENO del precedente all'EN del successivo, sia collegando l'uscita semantica dell'uno all'ingresso semantico dell'altro (d'ora in avanti detti ingressi ed uscite FBD, per distinguerli dagli ingressi Ladder, ovvero EN ed ENO).

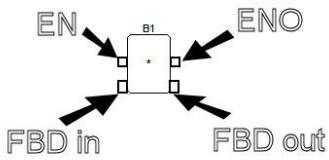

Figura 17. Schema grafico di un blocco in linguaggio a contatti

La semantica dei blocchi di tipo funzionale è di eseguire la loro operazione associata se EN è alto, e di scrivere in FBD out il risultato di essa, computato sui valori collegati ad FBD in, e di mettere ENO al valore alto. Se invece EN è basso, l'operazione non va eseguita, lasciando in FBD out il valore corrente e scrivendo un valore basso anche su ENO.

Sebbene non esplicitamente indicato dallo standard, che si limita a richiedere che le

uscite ENO debbano andare o in un EN di un altro blocco o in una bobina, è una limitazione ragionevole, non ultimo perché elimina diverse ambiguità alla radice, che ad ogni rung possa essere associata una sola bobina e che non sia possibile condividere blocchi tra diversi rung, ovvero che tutti i blocchi di un rung debbano avere lo stesso valore dell'EN in ogni ciclo di scansione.

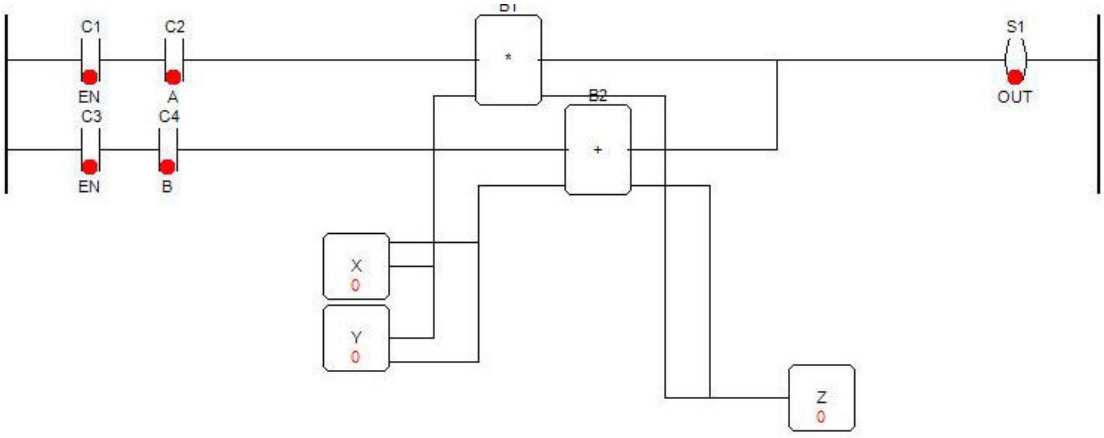

Figura 18. Un programma errato

Questo programma ha associato alla variabile *Z* i due blocchi $Z = X + Y$ e $Z = X * Y$, e $OUT = EN * A + EN * B$. Se si esegue il programma e si abilita *EN* ed una tra *A* e *B*, ponendo i valori di *X* o *Y* diversi da zero si vedrà il valore di *Z* oscillare tra somma e prodotto, segno che UniSim esegue entrambi i blocchi, ritenendoli (correttamente) entrambi associati al rung determinato dalla bobina OUT, invece che dai terminali EN dei singoli blocchi.

UniSim inoltre assume, in questo caso seguendo la lettera oltre che la sostanza dello standard, che un blocco sia seguito, nel flusso logico EN → ENO, da altri blocchi o da bobine. La versione corretta del programma è riportata in figura 19.

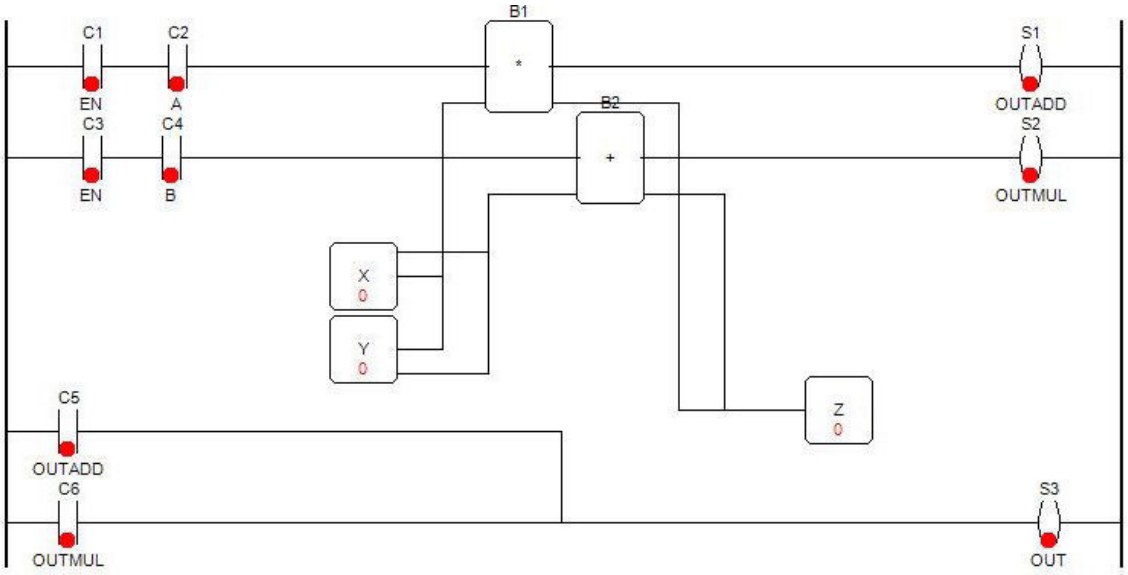

Figura 19. La versione corretta

A seguito delle limitazioni e posizioni sin qui fatte, si vede che il concetto di Rung può estendersi per includere una "rete" di tipo FBD ad esso associata, che consenta, allorquando l'espressione logica associata al rung abbia valore logico alto, di abilitare l'esecuzione della medesima. L'algoritmo che viene usato in UniSim è molto simile a quello che verrà descritto per l'FBD e si basa sulla equivalenza tra una rete del tipo qui mostrato e una espressione da tradurre in un parse tree e valutare.

Il concetto di rete FBD

Una rete FBD può essere vista come un processo di creazione di un valore a partire da altri valori e da funzioni di essi, il tutto interconnesso nei modi appropriati. In modo ancor meno formale ma sicuramente più utile alla trattazione che seguirà, una rete FBD è equivalente ad una espressione composta di variabili e di funzioni. Il processo di esecuzione di una rete FBD consiste nella creazione e successiva valutazione di un parse tree, attraverso un procedimento di riduzione. In tale procedimento, un nodo di tipo "blocco" che ha per figli solo nodi di tipo "variabile" viene ridotto calcolando la funzione associata al blocco con ingressi le variabili associate ai nodi figli, e ricorrendo il processo in profondità nell'albero ove un blocco abbia altri blocchi tra i suoi figli. Una volta ridotto l'albero alla sola radice, il valore di essa sarà scritto sull'uscita.

Il fatto che una rete FBD sia inclusa in una POU Ladder aggiunge una condizione di abilitazione all'esecuzione della rete così come definita precedentemente, ma non cambia la sostanza e la natura del problema.

Riprendendo l'algoritmo RtL, si ricorderà che esso includeva una chiamata ad un metodo di nome BuildTree() con un parametro di tipo BaseGraphicalContact. Tale metodo, qui riportato, è l'entry point di un algoritmo ricorsivo che produce un vero e proprio albero quale istanza della classe LDTree. A differenza di quanto descritto in teoria, il procedimento di riduzione dell'albero non avviene a partire dalle foglie, ma a partire dalla radice, la quale valuta i suoi discendenti, i quali ricorrono il processo finchè non si arriva ad una foglia (una variabile o una funzione in zero parametri) che ha se stessa, sotto opportuna rappresentazione, per valore e che chiude la ricorsione, consentendo di ripercorrere lo stack di chiamate in direzione opposta, fino a tornare alla radice, la quale sarà stata ridotta ad un unico valore finale.

```
' Questo codice è stato adattato dall'FBD
Private Sub BuildTreeNode(ByVal graphNode As GraphicalContact, _
        ByVal treeNode As LDTreeNode)
        Dim previousConnections As GraphicalFBDConnectionsList = _
                graphNode.FindIncomingFBDConnections()
        For Each previousConnection As GraphicalFBDConnection In previousConnections
                Dim srcObject As ILDConnectable = previousConnection.SourceObject
                Dim newNode As LDTreeNode
                If TypeOf (srcObject) Is GraphicalVariable Then
                        newNode = New VariableBoundLDTreeNode(CType(srcObject,
                        GraphicalVariable).BoundVariable)
                Else
                        Dim newObject As GraphicalContact = TryCast(srcObject,
                GraphicalContact)
                        ' siamo alla fine?
                        If newObject Is Nothing OrElse Not (newObject.IsBlock) Then
                Continue For
                        newNode = LDBlocksFactory.CreateBlock(newObject.Qualy,
                MyLadder.PousList)
                End If
                treeNode.AddNode(newNode)
                If TypeOf (newNode) Is BlockLDTreeNode Then _
                        BuildTreeNode(srcObject, newNode)
        Next
End Sub

Private Function BuildTree(ByVal root As GraphicalContact) As LDTree
        Dim GV As GraphicalVariable
        Dim outgoingConnections As GraphicalFBDConnectionsList =
        root.FindOutgoingFBDConnections()
        If outgoingConnections.Count = 0 Then Return Nothing
```

```
        GV = TryCast(outgoingConnections(0).DestinationObject, GraphicalVariable)
        ' non c'è un'uscita???
        If GV Is Nothing Then Return Nothing
        ' all'uscita c'è una variable d'ingresso?!?!?!...
        If Not (GV.VariableType = GraphicalVariableType.Output) Then Return Nothing
        Dim tree As New LDTree(GV.BoundVariable)
        tree.SetRootNode(LDBlocksFactory.CreateBlock(root.Qualy, MyLadder.PousList))
        BuildTreeNode(root, tree.GetRootNode)
        Return tree
End Function
```

Avendo distinto le connessioni di tipo LD da quelle di tipo FBD, assegniamo ad ogni contatto (il quale riceve tramite la POU genitrice un riferimento alle connessioni di tipo FBD) il compito di trovare le connessioni FBD uscenti da esso. Se non ve ne sono, o se ve ne sono ma la prima non è una variabile d'uscita, si assume che questo blocco sia parte di una rete e non il suo punto conclusivo e si salta il processo di creazione dell'albero. Questa logica elimina la necessità per l'algoritmo di creazione della rete di restituire al chiamante il punto in cui ha incontrato l'ultimo blocco[4] per consentire di "saltare" i componenti intermedi della rete. Adottando questo accorgimento, si valutano sì due volte i singoli blocchi, ma nella maggior parte dei casi di interesse pratico ciò non inficia le prestazioni complessive, e si ottiene una notevole semplificazione dell'algoritmo.

Se l'algoritmo valuta che si debba procedere a creare l'albero associato, crea un nuovo LDTree, a cui associa come uscita la variabile d'uscita della rete, e chiede al metodo ricorsivo BuildTreeNode() di creare un albero FBD avendo come blocco radice la radice dell'albero, e come contatto radice il contatto che è stato passato dal MakeExpressionPiece(). Si noti che essendo partiti per definire l'albero dal blocco suo predecessore invece che dalla variabile d'uscita, si può direttamente ed univocamente stabilire quale sia il tipo di blocco da porre alla radice dell'albero, un problema invece non banale nel puro FBD.

Il procedimento svolto dalla BuildTreeNode() è di ricercare tutti i predecessori del nodo che le è stato passato[5]. Per ogni predecessore, se si è in presenza di una variabile d'ingresso, la si aggiunge quale foglia all'albero, se si è in presenza di un blocco, si aggiunge il nodo del tipo da esso specificato all'albero e si invoca ricorsivamente la procedura per aggiungere i nodi figli del blocco.

Una volta "srotolata" la ricorsione, la si può riavvolgere, restituendo l'albero

[4] quello più a sinistra, si ricorda che usando lo schema RtL si procede nell'ordine di lettura arabo invece che in quello europeo e quindi i concetti di primo ed ultimo vengono ribaltati specularmente

[5] predecessori di tipo FBD, ignorando invece i collegamenti EN → ENO, che complicherebbero molto la logica dell'algoritmo senza ottenerne vantaggi dal punto di vista della *compliance* verso lo standard IEC

completo alla MakeExpressionPiece().

Ci si limita a far presente che la classe LDBlocksFactory è una classe di utilità che conosce l'insieme di blocchi implementati ed è in grado di creare nodi dell'albero FBD ad essi associati. Un nodo di albero FBD in Ladder è una sottoclasse concreta di una classe che espone un metodo Calculate() e una proprietà BlockName. Ogni nodo di un albero così fatto è in grado di valutarsi (tramite il metodo Calculate()) eventualmente accedendo ai propri nodi figli usando i metodi della superclasse (astratta) LDTreeNode.

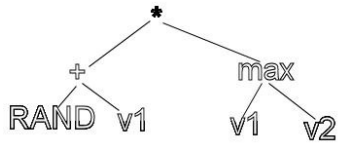

Figura 20. Un albero FBD per l'espressione (RAND+v1)*max(v1,v2) [33]

In figura 20 è riportato un esempio di un albero FBD. La valutazione di esso procede invocando il metodo Calculate() sulla radice, il quale valuta il prodotto dei suoi nodi figli, usando il loro metodo Calculate() per ricevere i valori da moltiplicare. Il nodo somma a sua volta, richiama i metodi Calculate() del nodo RAND e del nodo v1. Il Calculate() di RAND non legge i suoi nodi figli (non ne ha neanche) e ritorna un bit casuale. Il nodo v1 non si aspetta di avere nodi figli (e non potrebbe neanche averne) e ritorna il valore della variabile v1. A questo punto, il Calculate() del nodo prodotto, richiede al suo secondo figlio, max, di eseguire il suo Calculate(). max contiene un algoritmo in cui, dopo aver invocato i Calculate() di tutti i suoi nodi figli, in questo esempio due variabili, è in grado di ritornare il massimo tra tutti questi valori. Fatto questo, e non avendo altri nodi figli, il nodo radice esegue il prodotto e ritorna il suo valore al metodo di esecuzione della rete FBD. Con ciò il processo di riduzione dell'albero è completo, e si può assegnare il corretto valore alla uscita associata.

3.5 Conclusioni

In conclusione di questo capitolo è innanzitutto importante evidenziare come il processo di estensione che ha portato ad aggiungere i blocchi FBD all'interprete LD sia stato quasi naturale ed immediato, e minimo lo sforzo d'integrazione nella infrastruttura esistente. Il metodo scelto per la interpretazione del linguaggio a contatti si presta facilmente ad estensioni ed aggiunte sia di elementi per natura diversi da contatti e bobine, quali appunto i blocchi FBD, sia ed a maggior ragione, di tutti quegli elementi che possano essere ricondotti in fase di valutazione a termini booleani o a funzioni, complesse quanto si vuole, del valore corrente del rung.

In particolare, i più naturali candidati per l'estensione dell'interprete in un futuro lavoro di tesi sono i timer (potendosi ottenere i contatori tramite l'aritmetica e gli operatori di confronto sugli interi). Ancora, sono in corso di studio da parte di alcuni ricercatori [21,22] metodi per la migliore coesistenza tra SFC e LD e per la traduzione dall'LD all'SFC, e il modello ad oggetti di UniSim appare sufficientemente ricco da permettere di pensare all'implementazione di alcune di queste tecniche, così come in questa tesi si è introdotta la conversione da SFC in Ladder. Sono possibili, inoltre, ottimizzazioni sull'algoritmo RtL e sulla costruzione delle reti FBD, omesse in questa prima fase di sviluppo.

L'interprete del linguaggio a contatti di UniSim, comunque, appare allo stato dei fatti sufficientemente potente e completo per poter essere usato in interessanti progetti di simulazione di reali impianti di automazione.

Capitolo 4
Supporto per il linguaggio Functional Block Diagram

Tra i tre linguaggi grafici dello standard, l'SFC si presta alla descrizione di sistemi ad eventi discreti, il Ladder alla descrizione di operazioni logiche sui bit, e l'FBD alla descrizione funzionale di sistemi complessi, in termini di scomposizione in funzioni elementari variamente combinate.

Una parte importante di questo lavoro di tesi è stata l'aggiunta di un editor grafico e di un interprete per il linguaggio Functional Block Diagram. Tra i principali requisiti vi era ovviamente che le funzioni ed il *look-and-feel* del nuovo editor fossero sufficientemente simili a quelli dell'editor SFC e Ladder già implementati.

Questo capitolo descrive la creazione dell'editor FBD a partire dal know-how dei preesistenti editor Ladder e, specialmente, SFC, e dell'interprete per il linguaggio, questo sviluppato totalmente ex novo.

4.1 Presentazione del modello del problema

Il linguaggio FBD si basa sui concetti di variabile e blocco, in cui le variabili possono essere di ingresso o di uscita e ogni blocco simboleggia una ben definita funzione $y = f(x_1,x_2,\ldots,x_n)$, dove il numero e il tipo dei parametri che una funzione può ricevere e il tipo della sua (unica) uscita dipendono dalla semantica del blocco (per esempio un blocco di tipo ">" prenderà ingressi numerici e restituirà una uscita booleana).

Le variabili sono distinte nelle due categorie di ingresso e di uscita: una variabile d'ingresso è analoga ad una funzione di zero parametri e che ha per uscita il valore corrente della variabile ad essa associata. Una variabile d'uscita, viceversa, è una funzione in un parametro e la cui uscita si può assumere non significativa (in effetti è assente). L'utilità della funzione di una variabile d'uscita è di scrivere il suo unico parametro nella variabile stessa, eseguendo quindi una assegnazione.

I collegamenti, possibili tra variabili d'ingresso e blocchi, tra blocco e blocco, e tra blocco e variabile d'uscita, possono essere affermati o negati. Una connessione negata "trasporta" al suo terminale destinazione il negato logico del valore al suo terminale sorgente. Ovviamente, tale operazione non ha senso per variabili non booleane e la si ritiene non definita. L'operatore di negazione è indicato da un piccolo ovale posto sul terminale sorgente della connessione, la mancanza di simbolo indica una connessione affermata. Sono possibili in base allo standard, ma non attualmente implementate, connessioni a riconoscimento di fronte le quali fanno passare un valore alto alla destinazione se e solo se rilevano un fronte del tipo indicato alla loro sorgente [1,2].

Il linguaggio FBD si basa sulla valutazione di un'insieme di funzioni su un'insieme di ingressi atti a definire l'uscita nel modo desiderato, ed è privo di quei tradizionali strumenti di controllo di flusso che si trovano nei linguaggi di programmazione tradizionali, quali WHILE, FOR ed affini.

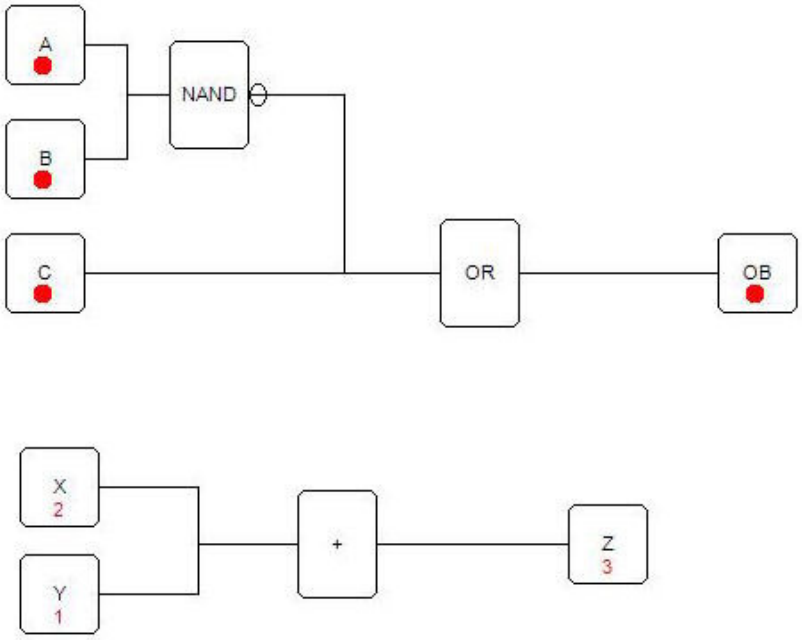

Figura 21. Un semplice programma in FBD prodotto con UniSim

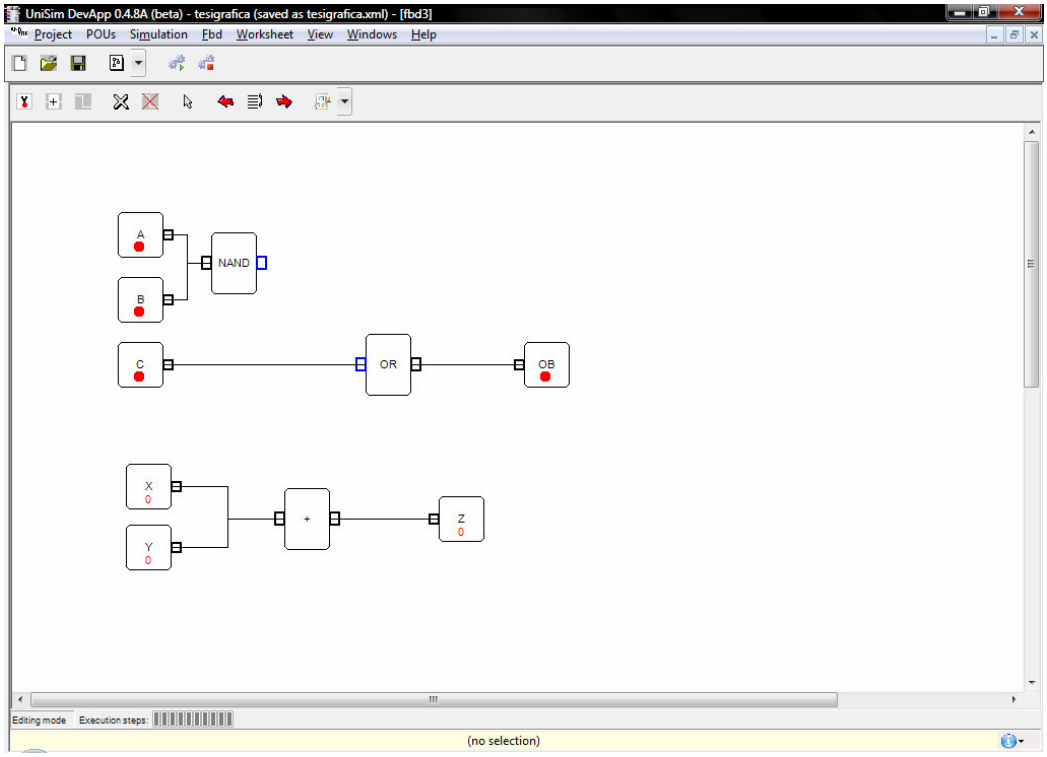

Figura 22. L'ambiente di editing FBD di UniSim

Come evidenziato dalle figure 21 e 22 è possibile che un singolo diagramma FBD abbia più uscite totalmente indipendenti tra loro, o che esse siano almeno parzialmente dipendenti l'una dall'altra. Lo standard IEC definisce alcuni vincoli sull'esecuzione di una rete FBD:

- nessun elemento di una rete verrà valutato prima che siano stati valutati tutti i suoi ingressi
- la valutazione di una rete non sarà ritenuta completa prima che lo stato di tutte le sue uscite sia stato valutato
- la valutazione di una rete non è completa finchè le uscite di tutti i suoi elementi non siano stati valutati
- l'ordine di valutazione delle reti rispecchierà le regole descritte al punto 4.3.3

Esso fornisce tre possibili opzioni e lascia all'implementatore la definizione del metodo che la sua implementazione userà:

- flusso dati: l'ordine di valutazione seguirà il vincolo che la valutazione di una rete sarà completata prima di iniziare a valutare una rete che usa una o più uscite della rete data quale ingresso
- numero di rete: l'ordine di valutazione sarà fornito dall'utente, ad esempio fornendo un ordinamento dei numeri di rete o delle etichette
- ordine grafico: le reti saranno valutate da sinistra a destra e dall'alto in basso

Partendo da questi vincoli e prerequisiti, si è proceduto ad una fase di analisi del problema e di ricerca di una soluzione.

4.2 Sviluppo della soluzione

La prima valutazione svolta è che per poter correttamente gestire l'FBD è necessario preliminarmente definire quale sia lo stato significativo di un FBD da dover creare, mantenere e modificare secondo richiesta dell'utente. Tale considerazione, che nella scrittura dell'interprete Ladder non è intervenuta in modo sostanziale, in quanto il modello a oggetti era già parzialmente predisposto, qui ha invece avuto una importanza centrale avendo dovuto definire un insieme di classi appropriato, che consentisse di programmare l'editor con semplicità e risultasse

sufficientemente potente e completo, nonché efficiente, per consentire la scrittura di un editor dalle prestazioni accettabili.

Creazione del modello a oggetti

In base alle considerazioni precedenti è chiaro che è importante nell'FBD tenere traccia dei blocchi e delle variabili, per ogni blocco della funzione associata e per ogni variabile del tipo (ingresso/uscita) e della variabile sottostante[6].

È importante inoltre definire e tenere traccia delle connessioni, memorizzando gli elementi sorgente e gli elementi destinazione, e poter discriminare rapidamente la loro tipologia e poter sapere se la connessione è negata o affermata, ed è necessario poter gestire in tempi rapidi tutte le informazioni legate alla grafica (se uno o più oggetti sono selezionati dall'utente, se sono selezionati i rettangolini di connessione, e la posizione corrente di ogni oggetto per poterlo disegnare su schermo). Inoltre, un corretto paradigma a oggetti conforta la scelta fatta nella prima versione del tool per l'SFC, in cui il sistema di editor grafico comunica a tutti gli oggetti del diagramma una superficie grafica su cui disegnare (un oggetto Graphics nella terminologia .net) e i vari oggetti sono responsabili, su richiesta del Form editor, di ridisegnare se stessi sulla superficie ricevuta: l'oggetto *GraphToDraw* nella terminologia usata dal codice di UniSim. Pertanto, il modello ad oggetti dell'FBD dovrà anche supportare questo modus operandi oltre a fornire all'editor metodi e proprietà per gestire lo stato di selezionato/non selezionato di ogni oggetto, e per impostarlo su richiesta. La richiesta principale che l'editor inoltrerà verso gli oggetti sarà di selezionare, deselezionare o verificare se è selezionato l'oggetto ad una data locazione spaziale (x,y). UniSim adotta il metodo di inviare la richiesta a tutti gli oggetti e lasciare che siano loro, conoscendo la loro *hit area* (ovvero la zona piana nell'area di disegno che ognuno di essi occupa), a compiere su di sé l'azione richiesta.

[6] Il blocco di tipo variabile infatti è una rappresentazione visiva e un punto di collegamento per fare uso della variabile vera e propria a livello di programma

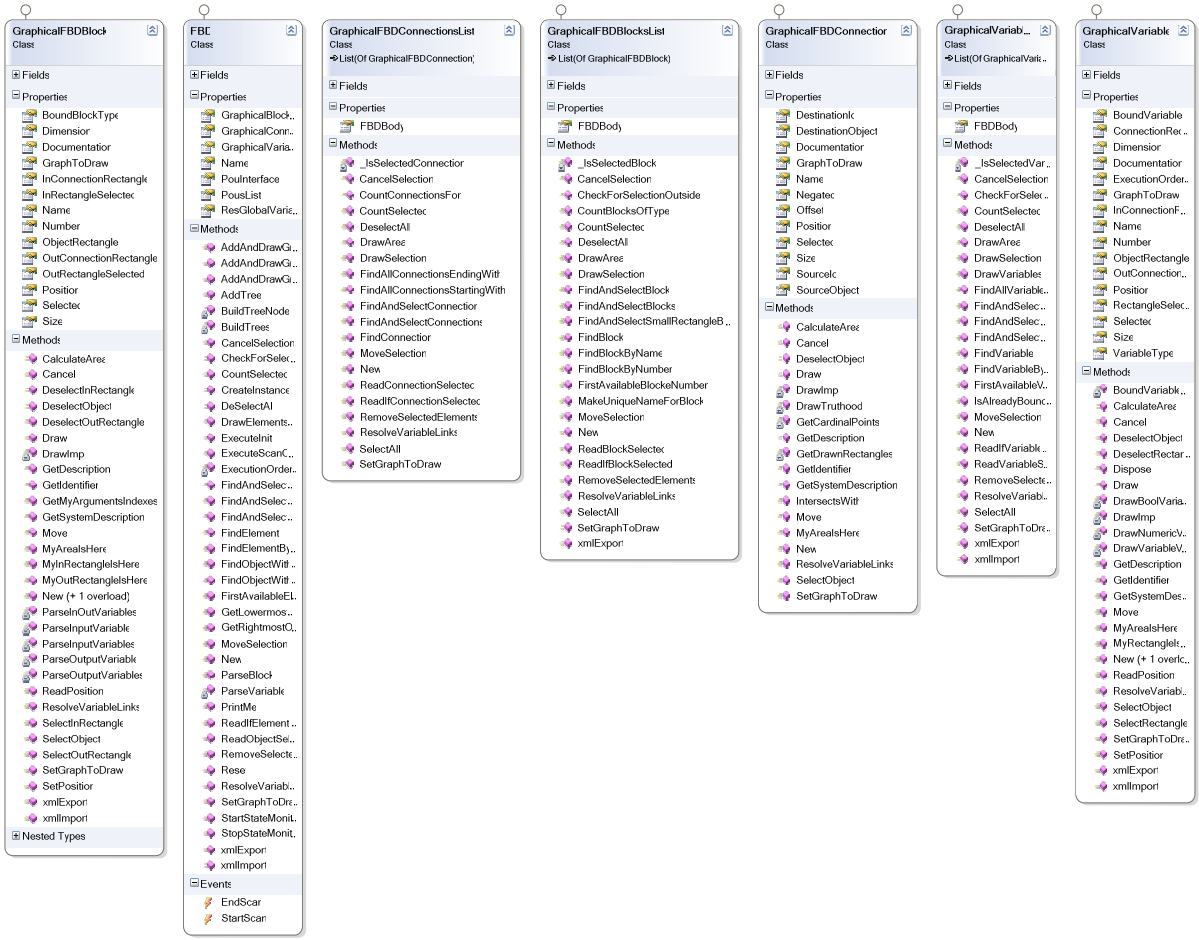

Figura 23. Il modello a oggetti dell'FBD di UniSim

Il modello a oggetti che si ritiene di dover sviluppare, tra le varie motivazioni per analogia con i modelli preesistenti dell'SFC [3] e del LD, è incentrato su una classe FBD, la quale contiene al suo interno un'istanza per ognuna delle classi GraphicalFBDConnectionsList, GraphicalVariablesList e GraphicalBlocksList, sottoclassi di liste parametriche rispettivamente di oggetti GraphicalFBDConnection, GraphicalVariable e GraphicalBlock. Le due ultime classi citate implementano l'interfaccia IFBDConnectable, usata per astrarre il comportamento degli oggetti FBD che supportano di essere collegati ad altri oggetti nel medesimo diagramma.

Gli oggetti modellati in questa soluzione supportano sia l'editor sia quello che poi sarà l'interprete vero e proprio.

Essendo l'editor molto vicino a quanto già descritto per l'SFC nel primo lavoro di tesi su UniSim, l'attenzione nel seguito del discorso sarà incentrata sulla parte algoritmica del supporto all'FBD, ovvero sulla traduzione tra la visione che, di un diagramma FBD da l'editor, e la visione che invece è necessaria per l'interpretazione del diagramma stesso.

L'editor FBD, come già accennato, vede il diagramma come tre insiemi: insieme delle variabili, insieme dei blocchi e insieme delle connessioni tra questi elementi, tutti istanze di una interfaccia IFBDConnectable, la quale assicura che sia possibile conoscere dimensione e posizione di un'oggetto, che esso sia dotato di un identificativo univoco nella propria POU, che abbia un nome, e che sia selezionabile, deselezionabile e dotato di rettangoli di connessione entranti ed uscenti (sarà compito del codice dell'editor discriminare i casi di variabile d'ingresso e d'uscita ed evitare di invocare il metodo non significativo).

A partire da questo schema, ciò che si ritiene di dover costruire, in analogia con quanto fatto per il Ladder, è una opportuna struttura dati che, su semplice chiamata di un metodo di esecuzione, sia in grado di svolgere le computazioni associate e di trasferire le opportune uscite nelle giuste locazioni.

In primo luogo, si vede come il nucleo dell'interprete debba essere l'insieme delle variabili d'uscita, così come per il Ladder si sono usate le bobine. Si vede inoltre che, in questo caso, non si può usare l'approccio di convertire il modello grafico in una espressione elementare, in quanto l'insieme degli operatori è più ampio, differenziato e, come si vedrà, estendibile dall'utente per mezzo di opportune POU di tipo function. È quindi necessario fare ricorso alla struttura generale usata in informatica per la gestione di espressioni arbitrarie: un parse tree con algoritmi di riduzione [19,33].

L'algoritmo usato da UniSim per generare e ridurre l'albero, che sarà descritto in dettaglio nel seguito, si basa sul poter agevolmente (in tempo lineare con il numero di variabili presenti nel diagramma) trovare tutte le uscite nel diagramma, e nel poter percorrere ogni collegamento in un tempo costante (ove per percorrere un collegamento intendiamo poter trovare i due terminali di esso). Se si hanno algoritmi per poter svolgere questa operazione è evidente come si possa navigare l'intero albero formato da ciascuna uscita e da tutte le connessioni che portano da essa agli ingressi (o eventualmente alle funzioni di zero parametri) e costruire così il parse tree.

Creazione dei parse trees

L'interprete dell'FBD si basa sostanzialmente su due passi: creazione del parse tree associato a ciascuna uscita e valutazione degli alberi così creati.

Attraverso l'uso di un approccio di tipo object oriented, favorito dal modello ad oggetti sviluppato, il codice dell'interprete è estremamente sintetico:

```
Public Function ExecuteScanCycle() As Boolean Implements
IIEC61131LanguageImplementation.ExecuteScanCycle
    RaiseEvent StartScan()
    BuildTrees()
    For Each t As FBDTree In m_EvalTrees
      t.Execute()
    Next
    RaiseEvent EndScan()
End Function
```

Il metodo BuildTrees() imposta la variabile locale m_EvalTrees della classe FBD ad un elenco di tutti i parse trees reperibili, ognuno dei quali dotato di un metodo Execute() eseguito in tempo lineare con l'insieme di nodi in esso contenuti (ammettendo che ogni nodo abbia per la sua valutazione un tempo costante, il che è vero salvo che in un caso).

Gli eventi StartScan() ed EndScan() vengono usati dalle POU per sincronizzare le modifiche allo stato con gli editor grafici.

Si riporta il codice completo delle principali routine dell'interprete, tutte nella classe FBD, al fine di poter meglio descrivere e commentare l'algoritmo con cui i parse trees vengono creati:

```
Private Sub BuildTreeNode(ByVal graphNode As IFBDConnectable, _
    ByVal treeNode As FBDTreeNode)
    Dim previousConnections As GraphicalFBDConnectionsList = _
        GraphicalConnectionsList.FindAllConnectionsEndingWith(graphNode)
    For Each previousConnection As GraphicalFBDConnection In previousConnections
        Dim srcObject As IFBDConnectable = previousConnection.SourceObject
        Dim newNode As FBDTreeNode
        If TypeOf (srcObject) Is GraphicalVariable Then
            newNode = New VariableBoundFDBTreeNode(CType(srcObject,
        GraphicalVariable).BoundVariable)
        Else
            newNode = FBDBlocksFactory.CreateBlock(CType(srcObject,
        GraphicalFBDBlock).BoundBlockType, Me.m_pousList)
        End If
        newNode.Negated = previousConnection.Negated
```

```
            treeNode.AddNode(newNode)
            BuildTreeNode(srcObject, newNode)
        Next
End Sub

Private Sub BuildTrees()
        m_EvalTrees.Clear()
        Dim outsList As GraphicalVariablesList = _
            GraphicalVariablesList.FindAllVariablesOfType(GraphicalVariableType.Outp
        ut)
        outsList.Sort(AddressOf ExecutionOrderIDComparison)
        For Each GV As GraphicalVariable In outsList
            Dim tree As New FBDTree(GV.BoundVariable)
            Dim previousConnections As GraphicalFBDConnectionsList = _
                GraphicalConnectionsList.FindAllConnectionsEndingWith(GV)
            If previousConnections.Count = 0 Then Continue For
            Dim previousConnection As GraphicalFBDConnection =
        previousConnections(0)
            Dim previousBlock As IFBDConnectable = previousConnection.SourceObject
            ' Se prima c'è una variabile assumiamo che l'utente voglia semplicemente
        "copiare"
            If TypeOf (previousBlock) Is GraphicalVariable Then
                    tree.SetRootNode(New
                    VariableBoundFDBTreeNode(CType(previousBlock,
                    GraphicalVariable).BoundVariable))
            Else
            ' Dovrebbe esserci un blocco, creiamo l'albero
                Dim treeRoot As GraphicalFBDBlock = CType(previousBlock,
                GraphicalFBDBlock)
                    tree.SetRootNode(FBDBlocksFactory.CreateBlock(treeRoot.Bo
                    undBlockType, Me.m_pousList))
                tree.GetRootNode.Negated = previousConnection.Negated
                BuildTreeNode(treeRoot, tree.GetRootNode)
            End If
            ' Se non c'è l'albero, inutile aggiungere
            If tree.GetRootNode() IsNot Nothing Then m_EvalTrees.Add(tree)
        Next
End Sub
```

Il primo passo dell'algoritmo è svuotare la lista creata al passo precedente. Sarebbe possibile passare per una fase di precompilazione e generare una volta sola gli

alberi (rendendo in tal modo molto più veloce l'algoritmo dal punto di vista dell'analisi ammortizzata) ma tra i requisiti di UniSim vi è appunto la mancanza di una fase di compilazione e la possibilità di editing in tempo reale delle POU durante la simulazione e quindi è impossibile applicare una simile ottimizzazione. Sarebbe possibile utilizzare un meccanismo di flag di marcatura *dirty*/*clean* sullo stato delle varie liste, in modo da ricreare gli alberi solo se vi sono state modifiche allo stato del diagramma tra due cicli di scansione, ma è un approccio riservato ad un futuro implementatore, e si è scelto in questo contesto di sviluppare l'algoritmo nella sua forma più lineare e semplice possibile.

Svolta questa operazione preliminare, l'algoritmo esegue una scansione del diagramma per ritrovare tutte le variabili d'uscita e le ordina in base al valore ExecutionOrderID (più basso il valore, prima verrà eseguito l'albero associato alla specifica variabile). L'algoritmo usato, sebbene soggetto a modifiche tra le varie versioni del framework trattandosi di una routine inclusa in esso, è attualmente il Quicksort di complessità $O(n\lg n)$ nel caso medio e $O(n^2)$ nel caso peggiore [4,18].

Usando gli iteratori inclusi nell'ambiente .net si scorre la lista delle uscite e per ciascuna di esse si verifica se possa esistere un albero associato ad essa. Se non vi sono connessioni che terminano nella variabile stessa si salta l'operazione (l'utente potrebbe star creando un nuovo albero e non aver ancora collegato l'uscita, per cui per rispettare i vincoli sulla libertà di editing per l'utente dobbiamo considerare questo caso al fine di poterlo ignorare).

Assumiamo inoltre, per ovvio motivo, che ogni uscita possa essere collegata ad un solo albero, altrimenti ci sarebbe ambiguità sul valore da assegnare effettivamente alla uscita, e tale limitazione oltre ad essere prevista nel codice dell'interprete è anche prevista dall'editor grafico il quale esegue un algoritmo di ricerca ogni volta che viene creata una variabile d'uscita per impedire che più variabili d'uscita siano legate alla stessa variabile reale (è chiaro che per l'utente è possibile intervenire sul file .XML per prevalere su questa limitazione, ma si sceglie di ignorare a monte tale problema)[7].

L'interprete supporta il collegamento diretto di una variabile d'ingresso a una d'uscita, ma in realtà è previsto che l'utilizzatore faccia uso del blocco funzione MOVE per una copia dei dati, in quanto la connessione diretta non è supportata dal formato di salvataggio dei dati usato da UniSim (ed è perciò proibita dall'editor).

Il meccanismo che si usa per creare l'albero è istanziare una classe FBDTree, che

[7] L'utente potrebbe creare una fork di UniSim senza questo vincolo ed aggirare ogni misura precauzionale

contiene sia un vero e proprio parse tree, sia un riferimento alla variabile d'uscita su cui scrivere il risultato della valutazione. Ogni tipologia di oggetto che si può trovare in un diagramma FBD viene associata ad un preciso tipo di nodo dell'albero. Una classe FBDBlocksFactory si occupa di istanziare i diversi tipi di nodo possibili e di tener traccia, ad uso dell'editor, dei vari blocchi esistenti ed istanziabili da parte dell'utente. Così facendo si evita all'utente il problema della digitazione manuale del nome di un blocco, e si evita di dover gestire gli errori relativi a blocchi inesistenti. Mentre il nodo associato ad una variabile può essere direttamente creato, i nodi relativi a blocchi vanno creati attraverso una factory (in tal modo si centralizza la corrispondenza tra nomi dei blocchi e classi di implementazione e si rende più facile l'aggiunta di nuovi blocchi, in quanto il programmatore deve intervenire in un solo blocco di codice ed automaticamente sia l'editor che l'interprete gestiranno correttamente le sue estensioni).

Ogni nodo dell'albero inoltre supporta la proprietà di essere o meno negato, e sarà il nodo stesso, in base al valore della proprietà, a negare o affermare il proprio valore prima di restituirlo nel processo di riduzione alla radice.

Una volta creato il nodo in quanto tale si entra in un algoritmo ricorsivo, BuildTreeNode(), che semplicemente itera lungo l'elenco degli elementi grafici che hanno una connessione che termina nell'elemento grafico che ha originato il nodo corrente (ovvero, i predecessori del nodo corrente) e per ognuno crea e inserisce tra i figli del nodo, il nodo loro associato. Il procedimento viene iterato per ogni sottonodo, ma si vede banalmente che nel caso di un nodo di tipo variabile non vi saranno sottonodi e quindi il procedimento verrà a terminare naturalmente.

Per i blocchi, viceversa, vi sarà un ulteriore ciclo di ricorsione (salvo per i blocchi senza parametri, quali in questa release il blocco RAND che ritorna un valore casuale di tipo BOOL secondo una distribuzione uniforme). Sarebbe ovviamente possibile ottimizzare alla radice la procedura in base a queste considerazioni escludendo dalla chiamata ricorsiva i blocchi di tipo variabile, ma sempre in base alla scelta progettuale di non perseguire la massima ottimizzazione, riservando tale attività a futuri implementatori, lo si è mantenuto nella sua versione basilare e più immediatamente leggibile e comprensibile. Non si intende in quest'ambito dimostrare la correttezza formale dell'algoritmo, sia perché esso è intuitivamente valido in quanto simile agli algoritmi usati per la creazione di parse trees a partire da stringhe testuali, fatte salve le differenze dovute all'uso di oggetti grafici in luogo di stringhe di caratteri, sia perché la correttezza andrebbe dimostrata alla luce

dell'algoritmo che si usa per poi eseguire il singolo parse tree, nel senso già discusso sia in quest'ambito, sia nel capitolo precedente per quanto concerne le reti FBD nelle POU Ladder.

Esecuzione del singolo parse tree

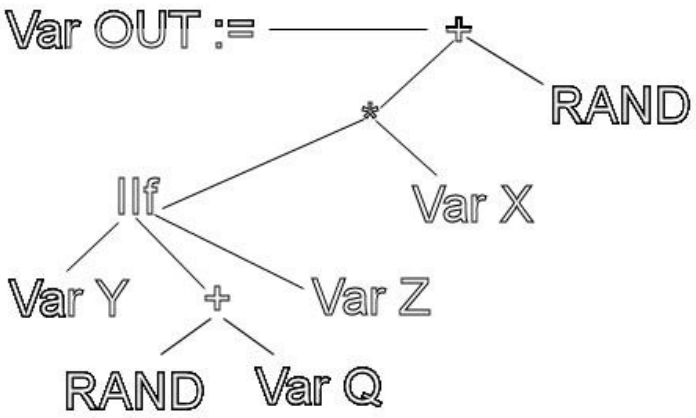

Figura 24. Un esempio di parse tree per l'espressione OUT = IIf(Y,RAND+Q,Z)*X+RAND

In figura 24 è evidenziato un esempio di parse tree così come generato dall'algoritmo descritto nella sezione precedente. Si vuole adesso sviluppare l'analisi di come tale albero possa essere converito in attività concrete, ovvero ridotto fino al punto di poter assegnare un valore univoco alla sua radice, e tale valore univoco assegnarlo a quella che nell'istanza dell'FBDTree è indicata come variabile target. Tutti i nodi creati, tramite binding con una variabile o attraverso la factory, sono oggetti intrinsecamente attivi, cioè in grado di valutare, indipendentemente dalla loro collocazione all'interno dell'albero, ma in sola eventuale dipendenza dai propri sottonodi, un valore da poter poi restituire ai nodi a minor profondità. Tale procedimento si può ritenere analogo ad un collassamento dell'albero su se stesso.

Esso inizia invocando l'opportuno metodo GetNodeValue() sul nodo radice. Il metodo, in dipendenza della classe del nodo radice, potrà restituire direttamente un valore, potrà utilizzare gli opportuni metodi della classe BaseVariable (o piuttosto di una sua sottoclasse concreta) per restituire il valore di una variabile, oppure leggere, tramite i loro metodi GetNodeValue(), i valori associati ai sottonodi e calcolarne una opportuna funzione. I sottonodi su cui verrà invocata la funzione GetNodeValue() itereranno lo stesso procedimento ad un maggior livello di

profondità, e così procederà l'algoritmo fino a giungere alle foglie, oltre le quali non potendosi scendere ulteriormente in profondità, si inizierà a risalire lungo lo stack, restituendo progressivamente i valori, fino a giungere nuovamente alla radice, stavolta con però tutti i sottonodi risolti in valori ben determinati. A tal punto il problema di calcolare la radice non sarà né più né meno complesso del problema di calcolare quella funzione che al nodo radice si è associata, che è in sostanza il meglio che ci si possa aspettare da un interprete: la riduzione della complessità del calcolo di una funzione alla intrinseca complessità della funzione stessa, senza alcun sovraccarico e sovrastruttura associate all'interprete medesimo.

Questa struttura consente di implementare ciascuna funzione, ovvero ciascun tipo di blocco, in una classe a sé stante spesso di piccole dimensioni. Si riporta, come esempio, la classe che realizza l'operatore ternario IIf (equivalente all'operatore ?: nei linguaggi C e derivati [11,12]).

```
Public Class IIfFBDBlock
    Inherits BlockFBDTreeNode

    Public Overrides ReadOnly Property BlockName() As String
        Get
            Return "IIf"
        End Get
    End Property

    Public Overrides Function Calculate() As Object
        Return IIf(CBool(Me(0).GetNodeValue()), Me(1).GetNodeValue(),
        Me(2).GetNodeValue())
    End Function
End Class
```

Ogni classe che eredita da BlockFBDTreeNode deve implementare la proprietà di sola lettura BlockName, che restituirà al chiamante il nome del blocco associato all'istanza, e la funzione Calculate() che ritorna il valore vero e proprio del nodo dati i suoi ingressi. Ogni nodo contiene una funzione di nome GetNodeValue(), che è la funzione da chiamarsi effettivamente per conoscere il valore associato ad un nodo. Scindendo in due metodi diversi il calcolo del valore, si consente al programmatore del singolo blocco di concentarsi sulla funzione da implementare, e si delega al nodo la gestione della negazione oppure di eventuali riconoscimenti di fronte che potrebbero essere implementati in future release del tool.

Gestione delle POU di tipo function

Lo standard IEC 61131-3 definisce tre tipi di unità organizzative di programma: program, function e function block. Sin dalla sua prima release UniSim ha supportato le POU di tipo program. Una parte di questo lavoro di tesi è consistita nell'introdurre la gestione delle POU di tipo function, sia dal punto di vista della loro definizione, sia dal punto di vista del loro utilizzo all'interno di altre POUs.

Una function è definita dall'IEC come "una POU che, quando eseguita, restituisce esattamente un elemento dato (che può anche essere composto di valori multipli, esempio un array o struttura) e la cui invocazione può essere usata nei linguaggi testuali come operando di una espressione" [1].

Al fine di rispettare il vincolo sulla singola uscita, quando si richiede di creare una POU di tipo function, UniSim presenta immediatamente la finestra di creazione variabile con il nome fissato ad "OUT" (viene stabilito per convenzione nel programma che il nome dell'uscita sia sempre OUT). Sfruttando appieno le potenzialità dello standard, UniSim pone sempre l'uscita nella lista delle variabili output della POU, e gli ingressi nella lista delle variabili input. In tal modo, si rispetta l'intenzione del comitato di standardizzazione che è stato particolarmente attento a classificare le variabili in base sia alla loro condivisibilità tra le diverse unità organizzative di programma, sia in rapporto all'utilizzo ammesso di esse (lettura,scrittura,lettura/scrittura). Ad oggi, UniSim non supporta il concetto di variabile d'input/output (InOut nella terminologia dello standard) e pur avendo definito all'interno del suo modello ad oggetti la relativa lista di variabili, non la utilizza ancora. Il supporto viene riservato per utilizzi futuri, e viene previsto sin d'ora per aumentare la compatibilità verso prodotti di terze parti che esportino nel medesimo formato di UniSim i propri progetti e facciano uso di tale lista variabili.

All'interno di POUs di tipo function, l'ambiente di programmazione facilita l'uso della lista variabili di input inserendola nel rapido menu associato all'editor di POU, oltre che nelle altre finestre in cui ha senso aggiungere tale indicazione.

Il tool consente la creazione di sole POU function nei linguaggi FBD e LD, in quanto l'SFC è un linguaggio intrinsecamente dotato di un concetto di stato (l'insieme delle fasi attive) e che quindi non si presta al concetto di una POU che, ad ogni esecuzione, a parità di ingressi restituisce medesima uscita. L'SFC si presta invece bene al concetto di blocchi funzionali, ovvero entità dotate di stato proprio e che possano ritornare uscite dipendenti dalla storia degli ingressi oltre che dai soli ingressi correnti (una function può essere vista quindi come una rete logica

combinatoria, un function block quale una rete logica sequenziale) [20].

Inoltre, l'SFC è anche limitato dalla impossibilità di invocare in una POU scritta in quel linguaggio le function definite all'interno del progetto. Lo standard prevede che all'interno di POU SFC possano comparire porzioni di programma scritte in FBD e/o LD, in particolare quali condizioni di transizione ed azioni. Avendo UniSim sin qui definito ed implementato il Sequential Functional Chart quale linguaggio autosufficiente, il che era una scelta progettuale obbligata allorquando non erano definiti nel tool né il Functional Block Diagram né tantomeno il Ladder, si è scelto di non estendere lo standard consentendo di invocare functions con una sintassi ed un formato di memorizzazione proprietari, lasciando eventualmente a future estensioni l'inserimento di blocchi di tipo FBD all'interno del modello ad oggetti dell'SFC, così come in questo lavoro di tesi è stato fatto per il Ladder Diagram.

L'uso di POUs di tipo function è reso possibile tramite un blocco generale, che riceve come parametro un riferimento ad una POU ed esegue un semplice algoritmo, delegando sostanzialmente alla POU l'esecuzione del proprio corpo, e fungendo da *driver*. L'algoritmo usato da tale blocco resetta tutte le variabili locali alla POU al proprio valore iniziale (in modo da garantire che la variazione dell'uscita possa dipendere dai soli ingressi).

Come secondo passaggio, l'algoritmo riceve dalla POU un riferimento all'elenco di variabili di ingresso, e copia i valori dei propri sottonodi (ottenuti tramite il passo ricorsivo già descritto in precedenza) nelle variabili d'ingresso, usando l'ordine dei propri sottonodi e quello di definizione degli ingressi per far corrispondere parametri formali a reali. L'algoritmo prevede anche il caso, ammissibile, che alcuni ingressi vengano omessi nella invocazione e pertanto debbano essere mantenuti ai valori di default previsti per il proprio tipo di dati (ciò è garantito da un precedente passo di reset dell'elenco variabili d'ingresso). Terminato il processo di interscambio parametri, alla POU viene richiesta l'esecuzione di un ciclo di scansione e il valore della variabile "OUT" della POU al termine di tale passaggio viene restituito quale valore complessivo del nodo.

Tale metodo non consente la ricorsione, né diretta né indiretta, in quanto i valori degli ingressi e delle variabili locali non vengono preservati. Questo, però, non è un problema né un limite in quanto lo standard non consente la creazione di funzioni ricorsive.

4.3 Conclusioni

Si è evidenziato nel corso del capitolo come la struttura dell'interprete FBD sia facilmente estendibile da parte di futuri sviluppatori, e si sono anche brevemente descritti i "punti di estensione" oltre che alcune delle principali funzioni e ottimizzazioni possibili che non fanno però parte della release qui discussa del tool UniSim.

Nonostante queste limitazioni, la versione dell'editor e dell'interprete FBD inclusa in UniSim, è sufficientemente potente per un'ampia gamma di utilizzi: la logica booleana e l'aritmetica fanno parte della attuale release ed utilizzando la possibilità di creare functions, inserita nella versione corrente per la prima volta, è possibile per l'utente aggiungere i propri "mattoncini" e combinare in modo naturale le proprie estensioni alle funzionalità built-in. Potrebbe risultare interessante un'estensione del formato di memorizzazione nonché del tool, per poter gestire oltre che progetti d'automazione anche librerie di sole POUs function e, quando saranno disponibili, function blocks che gli utenti possano importare nei propri progetti e riutilizzare in maniera semplice ed elegante. Fino a quel momento, oltre alla ovvia possibilità di utilizzare il copia&incolla per trasportare POUs tra diversi progetti, vi sarà comunque a disposizione di tutti gli utenti del tool, la possibilità di definire schemi funzionali e di decomporli nei modi più opportuni.

Capitolo 5
Traduzione del Sequential Functional Chart in Ladder

Il Sequential Functional Chart, quale strumento di modellazione e progettazione di alto livello, oltre che linguaggio vero e proprio, basato sul formalismo delle Reti di Petri, per la descrizione di sistemi ad eventi discreti, viene definito in termini dei concetti di fase e transizione, oltre che di regole che ne dettano l'evoluzione [1,8,21,22]. In particolare si definisce come stato di un SFC oltre che il valore delle sue variabili (il che è ovviamente vero per ognuno dei linguaggi dell'automazione) anche l'insieme delle sue fasi attive. Si definiscono le transizioni quali mezzi per l'evoluzione di tale stato, come meccanismi condizionali per la modifica dell'insieme di fasi attive. La transizione è definita da un insieme di fasi a monte, da una condizione di superabilità e dall'insieme delle fasi a valle. Se le fasi a monte di una transizione risultano tutte attive, la transizione è detta superabile. Se la condizione ad essa associata è vera, la transizione verrà superata in un ciclo di scansione, con la disattivazione contemporanea di tutte le fasi a monte, e l'attivazione contemporanea di tutte le fasi a valle. Ad ogni fase risultano associate zero o più azioni, usualmente consistenti nell'impostare le variabili ad opportuni valori (esistono azioni di Set, Reset, continue indicate con N, ed a limitazione di tempo). Le azioni si possono distinguere in tre grandi categorie: impulsive, continue e macroazioni. Le azioni impulsive vengono eseguite nel momento in cui si attiva la fase, le continue ad ogni ciclo di scansione in cui la fase risulta attiva, e le macroazioni consistono nel modificare l'insieme delle fasi attive di un altro SFC, e sono una estensione proprietaria (ma presente in quasi tutti gli ambienti) dello standard IEC. UniSim, infine, per sopperire alla mancata integrazione di blocchi

aritmetici all'SFC, fornisce azioni aritmetiche, consistenti nell'eseguire una espressione su variabili numeriche ed assegnare il valore ad un opportuno target.

Per come esso è definito, il linguaggio SFC può essere descritto in termini di equazioni booleane sui marker di fase, che saranno successivamente presentate. Una volta definito un problema in termini di variabili booleane, ne risulta semplice la risoluzione in termini di linguaggio a contatti.

Il lavoro discusso in questo capitolo è la creazione di un algoritmo che converta il Sequential Functional Chart in Ladder, e generi quindi una POU Ladder semanticamente equivalente ad una POU SFC assegnata (fatte, ovviamente, le opportune corrispondenze concettuali tra i concetti propri dell'SFC e quelli nativi del LD).

5.1 Analisi del problema

Si desidera una funzionalità semplice nell'utilizzo che, a partire da una POU nel linguaggio SFC, ne crei una semanticamente equivalente nel linguaggio LD.

Si ipotizza, nel modo più naturale possibile, di far corrispondere ad ogni fase nel linguaggio SFC, una variabile che funga da marker per la fase stessa nel linguaggio LD.

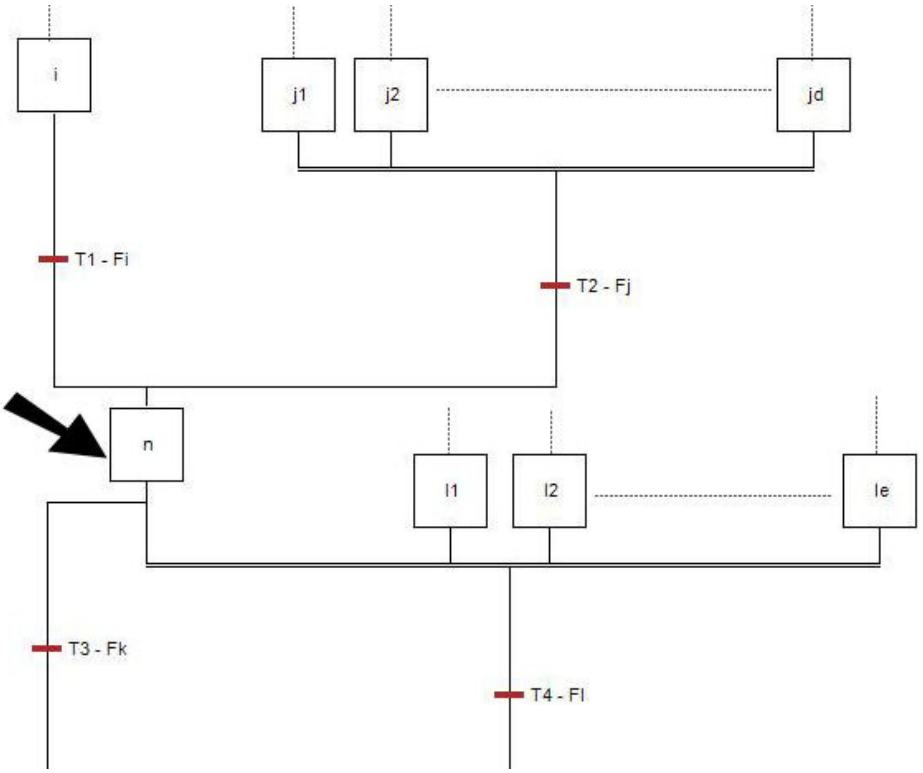

Figura 25. Una POU SFC d'esempio

Al fine di descrivere le equazioni booleane d'interesse si concentri l'attenzione sulla fase n. Si vede che essa sarà attiva se era attiva nel precedente scan cycle e nessuna transizione a valle è superabile oppure se è superabile una delle transizioni a monte. L'equazione che descrive l'essere superabile una delle transizioni a monte, nella forma generale per la fase n è $n.X := \sum_i (i.X \wedge Fi) \vee \sum_j \left[Fj \wedge \prod_h j.X_h \right]$. Una fase non sarà attiva se non era attiva nel precedente scan cycle e nessuna delle sue transizioni rispetta l'equazione di cui sopra, oppure se attiva nel precedente scan cycle deve essere disattivata in base all'equazione di attivazione di una fase a valle di essa, ovvero in base alla equazione $n.X = n.X \wedge \overline{\left\{ \sum_k Fk \vee \sum_l \left(Fl \wedge \prod_h l.X_h \right) \right\}}$ [23]. In questo lavoro si stabilisce la convenzione di seguire l'evoluzione del diagramma nel senso dell'attivazione, ovvero si controlla l'attivazione di tutte le fasi a monte e la condizione di transizione, per cui sarà usata soltanto la prima delle equazioni.

Le azioni hanno ognuna una propria equazione booleana associata, quelle impulsive associate ai fronti di salita dei marker di fase delle fasi collegate, quelle continue associate ai marker di fase in quanto tali. In questo schema generale ovviamente non rientrano le macroazioni, le quali modificano sia le equazioni di attivazione delle fasi e sia escono dallo schema generale descritto per le azioni in questa sede. Tali macroazioni inoltre richiedono che vi siano due POU in linguaggio SFC e che tra di esse sia possibile stabilire una gerarchia. Essendo questo lavoro dedicato alla traduzione della singola POU, avulsa da un contesto gerarchico, il problema delle macroazioni viene trascurato in quest'ambito. Inoltre, in questa sede si imposta il progetto di traduzione, limitandosi ai soli elementi fondamentali dell'SFC: fasi (ma non macrofasi), transizioni con sole condizioni booleane e marker di fase (sono esclusi i confronti aritmetici e temporali, i primi sarebbero possibili da implementare usando i blocchi FBD, i secondi necessitano di un supporto per i timers ancora non previsto nella release attuale), e le sole azioni dei tipi Set (S), Reset (R) e Continuous (N).

5.2 Sviluppo di una soluzione

L'algoritmo usato per il processo di traduzione si basa su un modulo, con un unico *entry point* pubblico e altre chiamate, private al modulo, che vengono invocate in successione l'una dopo l'altra per tradurre le diverse componenti della POU sorgente in quella destinazione:

```
Public Sub Translate(ByVal source As Sfc, ByVal target As Ladder)
    Dim convParams As New SFC2LDConversionData(source, target)
    TranslateVariables(convParams)
    convParams.AddRails() ' separa le diverse porzioni logiche del diagramma
    PrepareFirstScan(convParams)
    convParams.AddRails() ' separa le diverse porzioni logiche del diagramma
    TranslatePulseActions(convParams)
    convParams.AddRails() ' separa le diverse porzioni logiche del diagramma
    TranslateContinuousActions(convParams)
    convParams.AddRails() ' separa le diverse porzioni logiche del diagramma
    TranslateTransitions(convParams)
End Sub
```

L'algoritmo parte creando un oggetto di tipo SFC2LDConversionData che, oltre a racchiudere riferimenti alle due POU sorgente e destinazione, contiene diversi metodi di utilità. Tali metodi, scritti in analogia a quelli simili previsti dall'editor Ladder, permettono di "pilotare" la classe Ladder per creare, modificare, posizionare e collegare elementi in un diagramma in modo perfettamente analogo a quanto fa, appunto, l'editor. Tali operazioni servono non solo a poter materialmente scrivere il programma in Ladder, ma anche a disporlo in una maniera grafica che risulti quanto più possibile elegante e simile a ciò che farebbe un programmatore umano. La struttura esistente di UniSim si basa sul fatto che le medesime classi gestiscano sia la fase di esecuzione di un programma, sia la istanziazione e gestione dei vari elementi che lo compongono (creazione di fasi in SFC, di contatti in Ladder, di blocchi in FBD, …). L'editor è il principale componente client di queste operazioni, le quali risultano spesso più verbose in parametri dello stretto necessario. Si è dovuto costruire all'interno della classe SFC2LDConversionData un insieme di wrappers per tali funzioni della classe Ladder, che forniscano valori di default ragionevoli per i parametri non necessari e consentano il grado di libertà necessario ad un "editor" di tipo programmatico, in cui cioè le operazioni vengono svolte in maniera pilotata a partire da dati di configurazione preconfezionati, nello specifico da una POU SFC.

Un ottimo esempio è il metodo AddRails() della classe, che aggiunge ad un diagramma Ladder una coppia di alimentazioni destra e sinistra in modo analogo a quanto si fa usando l'apposito comando dell'editor

```
Public Function AddRails() As RailsObject
        Dim P1 As New Drawing.Point(10, 48 + m_RailsCounter)
        Dim leftid As Integer = target.FirstAvailableElementNumber
        target.AddAndDrawLeftRail(leftid, "", "", P1, 20, "", _
                "", Nothing, Nothing, Nothing)
        Dim P2 As New Drawing.Point(780, 48 + m_RailsCounter)
        Dim rightid As Integer = target.FirstAvailableElementNumber
        target.AddAndDrawRightRail(rightid, "", "", P2, 20, "", "", _
                Nothing, Nothing, Nothing)
        m_RailsCounter += 60
        Return New RailsObject(target.FindElementByLocalId(leftid),
target.FindElementByLocalId(rightid))
End Function
```

RailsObject è una classe di utilità che racchiude con due variabili membro le alimentazioni aggiunte. Il metodo deve mantenere un contatore intero della posizione delle nuove alimentazioni in modo analogo a quanto fatto dal Form di editing, operazione che non viene svolta in automatico dalla classe Ladder, la quale inoltre non permette di ottenere in modo banale l'informazione associata al parametro m_RailsCounter (che è l'ordinata massima delle alimentazioni). I parametri 10, 48 e 60 usati sono costanti usate anche dall'editor e che garantiscono un corretto posizionamento e un risultato grafico indistingubile da quello ottenibile programmando manualmente la POU destinazione.

Il metodo crea una istanza di alimentazione sinistra ed una di alimentazione destra nella opportuna posizione (P1 e P2), assegnando i minimi localId accettabili, questi sì ottenibili dalla classe Ladder. Infine, un riferimento alla coppia di alimentazioni viene restituito al chiamante. Usando questo strato wrapper risulta molto semplificato il compito di interagire con il supporto Ladder di UniSim, "fingendo" di essere la classe editor.

Fase 0: preparazione delle variabili

La prima e necessaria fase consiste nella traduzione delle variabili, la quale banalmente avviene ricreando l'elenco di variabili locali della POU SFC nell'elenco di variabili locali della POU Ladder. Non essendo possibile creare funzioni in SFC, e non includendo UniSim il supporto per i blocchi funzionali, non bisogna considerare il problema che la POU sorgente potrebbe avere variabili significative in liste diverse da quella di variabili locali. Ovviamente, le variabili globali a livello di risorsa non vengono duplicate ma rimangono quelle originali. Ciò potrebbe dar luogo a problemi nel caso l'evoluzione delle due POU non seguisse strade parallele,

ma non è questo che interessa nell'ambito di questo lavoro e quindi si demanda all'utilizzatore finale del prodotto l'accertamento del corretto funzionamento del suo programma tradotto.

Oltre alle variabili vere e proprie dell'SFC da tradursi 1 ad 1, il Ladder deve prevedere anche la gestione dei marker di fase. In SFC, le fasi, e quindi i relativi marker, sono concetti nativi del linguaggio che non necessitano di supporto esterno né di variabili d'appoggio esplicite. È l'interprete a farsi carico di gestire i simboli associati ai marker di fase e di leggerli/scriverli secondo le necessità e secondo la semantica dell'SFC e del programma utente.

Mancando il Ladder Diagram di tale supporto nativo, è necessario utilizzare locazioni addizionali di memoria, e quindi variabili addizionali, una associata a ciascuna fase dell'SFC, che si possa ritenere funzionalmente equivalente al marker di fase vero e proprio. Tale equivalenza consta, banalmente, nel ritenere equivalente l'attivazione di una fase al porre alto la variabile marker, e la disattivazione nel porre al valore basso il marker stesso. Ovviamente, è necessario che il programma SFC sia ben tradotto per poter ritenere una variabile BOOL equivalente ad un marker di fase, in quanto nulla osta, nella semantica del LD, ad alzare una variabile marker nel momento in cui nessuna transizione che abbia la fase stessa come destinazione risulti superabile, mentre in un interprete SFC privo di bugs tale operazione è impossibile.

Fase 1: preparazione del "first scan"

La fase di inizializzazione di un SFC viene svolta dall'interprete all'avvio della simulazione e consiste nella attivazione di tutte e sole le fasi che il programmatore dell'SFC abbia dichiarato iniziali. In tal senso, l'inizializzazione di un SFC è dichiarativa, in quanto il programmatore dichiara la sua intenzione mentre scrive il programma e delega all'interprete il compito di rendere operativa, in modi non di suo interesse, la propria volontà.

Il Ladder non prevede alcuna fase né costrutto di tipo dichiarativo, e la fase di inizializzazione propria delle POU di tale linguaggio è sostanzialmente vuota. Per ovviare a questo problema, UniSim analogamente ad altri ambienti di programmazione include in ogni POU Ladder una variabile interna, detta first_scan. Tale variabile, gestita da UniSim stesso in fase di simulazione, viene posta al valore

alto ogni volta che si avvia la simulazione e resettata al valore basso alla fine del primo scan cycle di ogni simulazione. Con questa semantica, è possibile usare il bit first_scan per inizializzare la POU, in analogia a quanto fatto dall'interprete SFC. In particolare, viene predisposto un insieme di rung consistenti in un contatto associato al first_scan ed una bobina di Set associata al marker di fase, e questo per ogni fase iniziale. In tal modo alla fine del primo ciclo di scansione tutte e sole le fasi iniziali risulteranno alte. Ciò è in parziale contrasto con l'interprete SFC di UniSim che, essendo basato su un algoritmo con ricerca di stabilità, pone a preattive le fasi iniziali alla fine del primo ciclo di scansione, e ad attive alla fine del secondo ciclo di scansione se non vi sono transizioni uscenti da esse superabili. La traduzione Ladder viene invece effettuata senza la ricerca di stabilità (per poterla ottenere sarebbero necessarie variabili logiche a tre stati associati rispettivamente alla condizione: active, preactive, inactive oppure due variabili logiche in cui una delle quattro possibili combinazioni venga trascurata e le altre tre mappate sugli stati opportuni) e quindi pone semplicemente attive le fasi iniziali nel primo scan cycle e supera completamente le transizioni (con disattivazione delle fasi sorgente e attivazione delle fasi destinazione) in un unico ciclo di scansione invece dei due necessari all'interprete SFC. Tra le possibili ottimizzazioni di questa funzionalità vi è di sicuro l'uso di un unico contatto first_scan e il collegamento ad esso di tutte le bobine relative alle fasi iniziali. Essendo i rung di UniSim determinati dalle bobine e non dai contatti, l'ottimizzazione ridurrebbe il numero di contatti e connessioni complessive nel diagramma, ma non andrebbe a ridurre il numero di rung da valutare ad ogni scan cycle da parte dell'interprete, per cui avrebbe in realtà un maggior impatto grafico che funzionale e prestazionale, ed è per questo motivo che è stata omessa dalla prima versione del traduttore qui presentata.

Fase 2: traduzione delle azioni impulsive

Le azioni impulsive sono per definizione quelle azioni che vanno eseguite per il solo ciclo di scansione in cui il marker della fase ad esse associato passa da basso ad alto. Sfruttando i contatti a fronte di salita è facile creare una serie di rung, a sinistra dei quali vi sia un contatto a fronte di salita associato al marker della fase, e come bobina vi sia l'opportuna bobina di Set o Reset associata alla variabile target della azione. È evidente come, in caso, di ripetuta attivazione della stessa fase la variabile non sarà modificata (in senso di alzata o abbassata a seconda di quale sia l'azione)

più volte, così come tra l'altro è nell'interprete SFC di UniSim (il comportamento in caso di una attivazione multipla è delegato all'implementazione in quanto non supportato dallo standard)

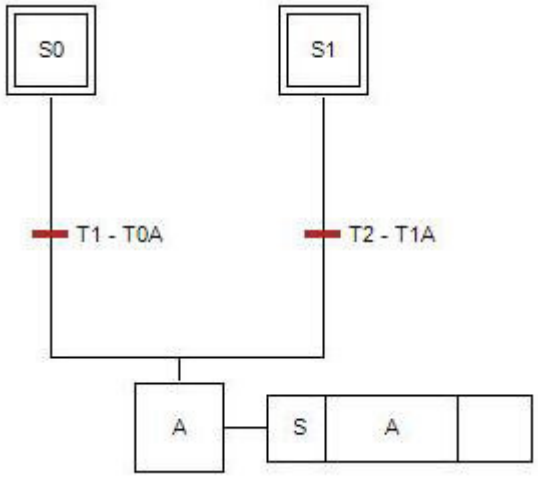

Figura 26. Una POU SFC che dimostra il comportamento di UniSim in caso di multiattivazione

Fase 3: traduzione delle azioni continue

Le azioni continue sono quelle azioni, per definizione, che vengono eseguite ad ogni ciclo di scansione. L'unica azione continua implementata da UniSim è l'azione continua, demarcata dalla lettera N nello standard, che consiste nel tenere alta una certa variabile finchè è attiva la fase. Tale azione si traduce in LD con il trasferire il valore del marker di fase nella variabile target, e quindi con un rung alla cui sinistra si trova un contatto normalmente aperto associato al marker di fase ed alla cui destra si trova una bobina normalmente aperta associata alla variabile target, come nell'esempio in figura

Figura 27. Una azione continua tradotta in Ladder

Fase 4: traduzione delle transizioni

La traduzione delle transizioni è la parte algoritmicamente più interessante del progetto di traduzione, ed anche limitata ai soli marker di fase[8] e alle AND, OR e NOT di variabili booleane ha richiesto uno studio sistematico.

Si riporta l'algoritmo adottato per la traduzione, e a partire da esso si svilupperà l'analisi teorica che ha portato a concepirlo in tal modo.

```
Private Function TranslateTransitionPiece(ByVal params As SFC2LDConversionData, _
        ByVal T As GraphicalTransition, ByVal condPiece As BoolExpressionNode, _
        ByVal hookingPoint As GraphicalContactList) As GraphicalContactList

        If condPiece Is Nothing Then Return Nothing

        Dim condPieceObj As Object = CObj(condPiece)

        Select Case condPiece.GetType().Name
                Case "PlusNode"
                        If condPiece.Neg Then
                                ' !(A+B)=!A*!B
                                Dim newNode As New MultNode()
                                newNode.Neg = False
                                newNode.NextNodes = condPieceObj.NextNodes
                                For Each node As BoolExpressionNode In newNode.NextNodes
                                        node.Neg = Not (node.Neg)
                                Next
                                TranslateTransitionPiece = TranslateTransitionPiece(params, T,
                        newNode, hookingPoint)
                                ' rimetti i nodi a posto
                                For Each node As BoolExpressionNode In newNode.NextNodes
                                        node.Neg = Not (node.Neg)
                                Next
                                Exit Function
                        End If
                        Dim newHookingPoint As New GraphicalContactList()
                        For Each node As BoolExpressionNode In condPieceObj.NextNodes
                                Dim rails As RailsObject = params.AddRails()
                                Dim formalHP As New GraphicalContactList(rails.leftRail)
                                newHookingPoint.AddRange( _
                                        TranslateTransitionPiece(params, T, node, _
                                                IIf(Of
                                                GraphicalContactList)(condPieceObj.NextNodes.
                                                Count = 1, _
                                                hookingPoint, formalHP)))
                        Next
```

[8] in realtà i marker di fase, essendo essi stessi variabili booleane, non presentano difficoltà particolari di traduzione

```
            Return newHookingPoint
        Case "MultNode"
            If condPiece.Neg Then
                ' !(A*B)=!A+!B
                Dim newNode As New PlusNode()
                newNode.Neg = False
                newNode.NextNodes = condPieceObj.NextNodes
                For Each node As BoolExpressionNode In newNode.NextNodes
                    node.Neg = Not (node.Neg)
                Next
                TranslateTransitionPiece = TranslateTransitionPiece(params, T,
            newNode, hookingPoint)
                ' rimetti i nodi a posto
                For Each node As BoolExpressionNode In newNode.NextNodes
                    node.Neg = Not (node.Neg)
                Next
                Exit Function
            End If
            Dim newHP As GraphicalContactList = hookingPoint
            For Each node As BoolExpressionNode In condPieceObj.NextNodes
                newHP = _
                    SFC2Ladder.TranslateTransitionPiece(params, T, _
                    node, newHP)
            Next
            Return newHP
            Case "CompareNode"
                Exit Select
            Case "VariableNode"
                Dim varCopy As BaseVariable =
            params.target.PouInterface.localVars.FindVariableByName(condPiece
            Obj.Var.Name)
                If varCopy Is Nothing Then varCopy =
            params.target.ResGlobalVariables.FindVariableByName(condPieceObj
            .Var.Name)
                Dim varNode As GraphicalContact = params.AddContact( _
                    IIf(Of String)(condPiece.Neg, "Normal Closed Contact",
                "Normal Open Contact"), _
                    varCopy)
                Dim hpPosition As Drawing.Point =
            hookingPoint(0).ReadPosition
                hpPosition.X += 2 * varNode.Size.Width
                varNode.SetPosition(hpPosition)
                Dim theList As New GraphicalContactList(varNode)
                params.ConnectEntriesLists(hookingPoint, theList)
                Return theList
            Case "StepMakerConditionNode"
                If condPieceObj.Type Then Exit Select
                Dim theStep As BaseGraphicalStep = condPieceObj.StepMaker
                Dim boundVar As BaseVariable =
```

```
                    params.GetStepMarkerVariable(theStep)
                        Dim varNode As GraphicalContact = params.AddContact( _
                            IIf(Of String)(condPiece.Neg, "Normal Closed Contact",
                        "Normal Open Contact"), _
                            boundVar)
                        Dim theList As New GraphicalContactList(varNode)
                        params.ConnectEntriesLists(hookingPoint, theList)
                        Return theList
            End Select
            Return Nothing
End Function


Private Sub TranslateTransition(ByVal params As SFC2LDConversionData, ByVal T As
GraphicalTransition)
        Dim ld As Ladder = params.target
        Dim prevSteps As GraphicalStepsList = T.ReadPreviousGraphicalStepsList()
            Dim precond As BaseVariable =
            params.target.PouInterface.localVars.CreateAndAddVariable( _
            params.target.PouInterface.localVars.MakeUniqueName("TPRECOND"), "",
        "", _
            "false", "BOOL")
        Dim rails As RailsObject = params.AddRails()
        Dim pos As Drawing.Point = rails.leftRail.ReadPosition()
        Dim prevOne As BaseGraphicalContact = rails.leftRail
        Dim contact As GraphicalContact
        pos.X = 200
        For Each S0 As BaseGraphicalStep In prevSteps
            If Not (TypeOf (S0) Is GraphicalStep) Then Continue For
            contact = params.AddContact("Normal Open Contact", _
                params.GetStepMarkerVariable(S0))
            contact.SetPosition(pos)
            params.ConnectEntries(prevOne, contact)
            prevOne = contact
            pos.X += 60
        Next
        Dim coil As GraphicalContact = params.AddCoil("Normal Coil", precond)
        coil.SetPosition(New Drawing.Point(Math.Max(600, pos.X), pos.Y))
        params.ConnectEntries(prevOne, coil)
        params.ConnectEntries(coil, rails.rightRail)
        rails = params.AddRails()
        pos = rails.leftRail.ReadPosition()
        pos.X = 200
            Dim condVar As BaseVariable =
            params.target.PouInterface.localVars.CreateAndAddVariable( _
            params.target.PouInterface.localVars.MakeUniqueName("TCOND"), "", "", _
            "false", "BOOL")
        coil = params.AddCoil("Normal Coil", condVar)
            Dim transen As BaseVariable =
```

```
            params.target.PouInterface.localVars.CreateAndAddVariable( _
            params.target.PouInterface.localVars.MakeUniqueName("TEN"), "", "", _
            "false", "BOOL")
Dim contact2 As GraphicalContactList = TranslateTransitionPiece(params, T, _
            T.ReadCondition().RootNode, New GraphicalContactList(rails.leftRail))
pos.X = 400
pos.X = 600
coil.SetPosition(pos)
If contact2 IsNot Nothing Then
            params.ConnectEntriesLists(New GraphicalContactList(rails.leftRail),
contact2)
            params.ConnectEntriesLists(contact2, New GraphicalContactList(coil))
Else
            params.ConnectEntries(rails.leftRail, coil)
End If
params.ConnectEntries(coil, rails.rightRail)
rails = params.AddRails()
pos = rails.leftRail.ReadPosition
contact = params.AddContact("Normal Open Contact", precond)
pos.X = 200
contact.SetPosition(pos)
params.ConnectEntries(rails.leftRail, contact)
Dim contactC As GraphicalContact = params.AddContact("Normal Open Contact",
condVar)
pos.X = 400
contactC.SetPosition(pos)
params.ConnectEntries(contact, contactC)
coil = params.AddCoil("Normal Coil", transen)
pos.X = 600
coil.SetPosition(pos)
params.ConnectEntries(contactC, coil)
params.ConnectEntries(coil, rails.rightRail)
Dim nextSteps As GraphicalStepsList = T.ReadNextsGraphicalStepsList
contact = params.AddContact("Normal Open Contact", transen)
rails = params.AddRails()
pos = rails.leftRail.ReadPosition
pos.X = 200
contact.SetPosition(pos)
params.ConnectEntries(rails.leftRail, contact)
For Each S0 As BaseGraphicalStep In prevSteps
            If Not (TypeOf (S0) Is GraphicalStep) Then Continue For
            coil = params.AddCoil("Reset Coil", params.GetStepMarkerVariable(S0))
            rails = params.AddRails()
            pos = rails.leftRail.ReadPosition
            pos.X = 600
            coil.SetPosition(pos)
            params.ConnectEntries(contact, coil)
            params.ConnectEntries(coil, rails.rightRail)
Next
```

```
        For Each S0 As BaseGraphicalStep In nextSteps
            If Not (TypeOf (S0) Is GraphicalStep) Then Continue For
            coil = params.AddCoil("Set Coil", params.GetStepMarkerVariable(S0))
            rails = params.AddRails()
            pos = rails.leftRail.ReadPosition
            pos.X = 600
            coil.SetPosition(pos)
            params.ConnectEntries(contact, coil)
            params.ConnectEntries(coil, rails.rightRail)
        Next
End Sub

Private Sub TranslateTransitions(ByVal params As SFC2LDConversionData)
        Dim ld As Ladder = params.target
        For Each T As GraphicalTransition In params.source.GraphicalTransitionsList
            TranslateTransition(params, T)
        Next
End Sub
```

L'algoritmo si basa sull'idea di scomporre la transizione in diversi elementi, e questa idea è rinforzata dall'uso della classe BooleanExpression, che è rappresentata internamente in forma di albero radicato, nell'interno del codice dell'interprete SFC per definire le transizioni. Il metodo TranslateTransitionPiece si basa sull'idea di attraversare l'albero in modo ricorsivo, un nodo alla volta, valutando eventualmente la negazione che può essere associata ad ogni nodo. Inoltre, per come è definito il linguaggio a contatti, ogni parte di ogni rung deve essere collegata al resto del rung mediante una connessione, e ovviamente il punto esatto in cui effettuare la connessione dipenderà dal tipo di nodo oltre che dai suoi sottonodi. Tale punto viene detto, in questo algoritmo, *hooking point* (punto d'aggancio) ed è l'insieme di tutti i contatti da collegare al resto del rung per formare un rung valido (con metodo ricorsivo ogni punto della ricorsione assume di star completando un rung altrimenti già completo).

In effetti, l'algoritmo qui esaminato, divide una transizione in tre parti: *precondizione*, *condizione* e *superamento*. La precondizione consiste nell'equazione descritta all'inizio del capitolo, ed è data dalla AND dei marker di tutte le fasi precedenti. La condizione è la vera e propria condizione associata alla transizione e viene tradotta secondo lo schema di hooking point descritto in precedenza, con accorgimenti speciali per gestire il caso di OR o AND negate (che vengono ricondotte ad AND e OR semplici in base alle leggi di De Morgan). Mettendo in AND le due variabili di precondizione e condizione, una per ogni transizione e definite dal modulo di conversione quali variabili locali alla POU, ma in pratica

riservate nell'uso all'implementazione, si attivano o meno i rung dell'effettivo superamento della transizione, consistenti in disattivazione delle fasi a monte e attivazione di quelle a valle, il tutto in un unico scan cycle, in modo che l'operazione risulti a durata nulla così come detta lo standard[9].

Figura 28. Fase di precondizione

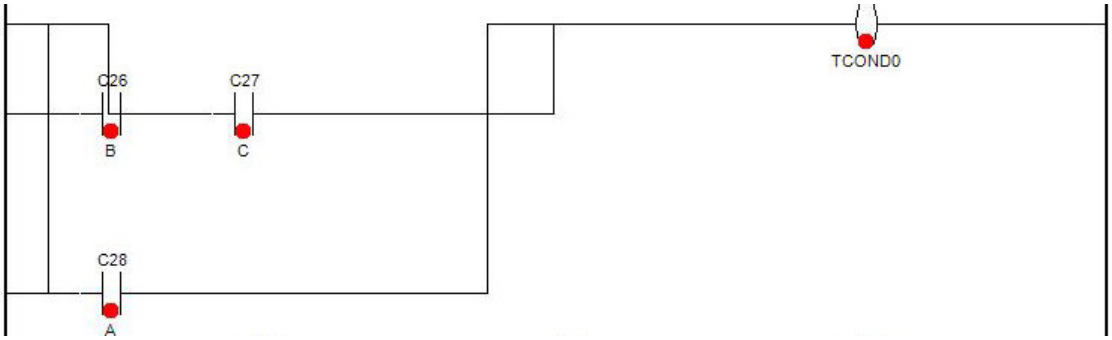

Figura 29. Fase di condizione

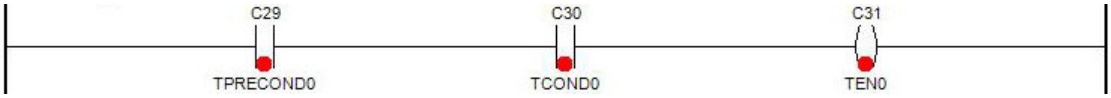

Figura 30. Verifica di superabilità

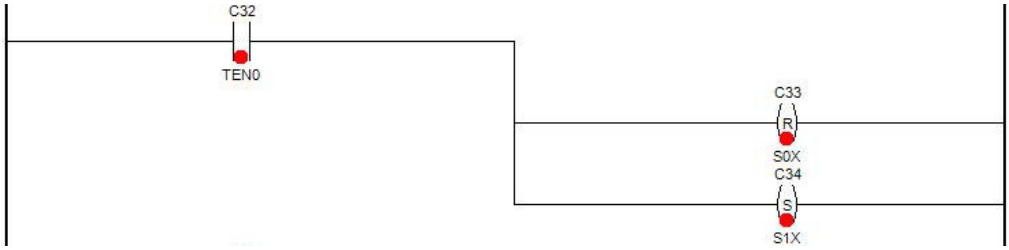

Figura 31. I rung di superamento della transizione

[9] Ovviamente la durata reale della modifica di una variabile, per quanto infinitesima, non può mai essere davvero nulla. Il requisito della durata nulla è un'astrazione che sta a indicare che tale durata è, appunto, piccola quanto può essere utile nella pratica di programmazione

5.3 Conclusioni

Il meccanismo di conversione sviluppato in questo ambito, per quanto ancora incompleto, è sicuramente un valido strumento didattico, in quanto evidenzia come sia realmente possibile implementare il linguaggio Sequential Functional Chart. Inoltre, essendo quest'ultimo inteso da alcuni autori quale strumento di modellazione, e non essendo disponibile su alcune marche di PLC [24], la traduzione in Ladder è un ausilio al porting di programmi al di fuori dell'ambiente UniSim verso sistemi reali, ove quest'ultimi ovviamente supportino lo standard IEC e il relativo formato di esportazione XML. Infine, l'estensibilità dell'algoritmo è molto ampia, basta aggiungere al modulo le nuove funzioni che traducono gli elementi di programma SFC voluti ed inserire le opportune chiamate senza dover intervenire sull'architettura del modulo. Le principali possibilità di estensione del modulo sono state indicate nella introduzione al capitolo, in cui si è fatto riferimento esplicito a tutte le limitazioni in esso presenti attualmente. Alcune di esse saranno eliminabili direttamente agendo sul codice del modulo di traduzione, altre invece dipenderanno anche dall'esistenza di un supporto per determinate funzioni nel supporto LD (un esempio su tutti, il supporto per le azioni a temporizzazione).

Conclusioni e sviluppi futuri

Come già più volte accennato nelle conclusioni dei vari capitoli, l'incremento apportato ad UniSim con questo lavoro di tesi ha esteso il valore didattico dello strumento e le sue funzionalità con l'aggiunta di nuovi linguaggi, tipi di dati, e un nuovo tipo di POU, oltre ad averne profondamente influenzato ed esteso l'architettura. Nondimeno, nessuno strumento software, tranne forse i più banali, può mai ritenersi davvero "completo", in quanto le specifiche e le linee guida al suo sviluppo sono rapidamente mutevoli e, spesso, ogni nuova funzione ne rende possibili e desiderabili altre, e quindi lo sviluppo procede ancora.

Così è sicuramente anche nel caso di questo lavoro di tesi. Si è introdotto un nuovo tipo di dati, e lo si è reso "operativo" nei tre linguaggi grafici dello standard. Un ragionevole esempio di sviluppo successivo è l'aggiunta degli altri tipi di dati previsti nella normativa IEC.

Si è previsto il supporto per le POUs function nei linguaggi FBD ed LD. È utile, e ragionevole, che un successivo lavoro di tesi estenda l'ambiente prevedendo le POUs di tipo function block, e consentendo l'utilizzo di POUs function anche nell'SFC (beninteso non la loro dichiarazione).

Inoltre, con l'aggiunta di blocchi FBD e rungs LD all'SFC si completerebbe quella che è la visione che dell'SFC danno sia lo standard IEC, sia quello PLCOpen: uno strumento di modellazione di sistemi che interopera con gli altri due linguaggi grafici per la definizione di azioni e transizioni. Ancora, risulterebbe di grande importanza il completamento del traduttore SFC → Ladder, con quindi l'aggiunta di blocchi temporizzatori all'ambiente LD.

In conclusione, però, dopo oltre due anni di vita e diversi *maintainers* si può ritenere che l'ambiente UniSim abbia raggiunto una discreta maturità, implementando i tre linguaggi grafici, due dei tre tipi di POUs, e un completo supporto per la logica booleana e l'aritmetica sia intera che in virgola mobile. Esso è l'unico ambiente didattico open-source sotto licenza GPL a fornire un supporto così completo per i linguaggi dell'automazione ed a fare uso del formato standard PLCOpen XML Formats for IEC 61131-3, ponendosi non solo come simulatore didattico ma anche come ambiente di prototipazione rapida per un successivo porting verso un reale PLC dotato di supporto per il medesimo formato.

Bibliografia

1. International Electrotechnical Committee – standard 1131, sezione 3 "Programming Languages"
2. PLCOpen – standard "XML Formats for IEC 61131-3", PLCOpen TC6
3. Pierangelo Di Sanzo – "Un tool di sviluppo validazione e controllo per progetti di automazione"
4. Microsoft Corporation, "MSDN Library" ed. "Visual Studio 2005"
5. Julien Templeman, David Vitter – "Visual Studio .net – The .net framework black book", The Coriolis Group LLC 2002
6. http://primates.ximian.com/~miguel/momareports-previous/ http://primates.ximian.com/~miguel/momareports-previous/d4a34671a9d14c58a99349996c60f4f8.txt
7. Microsoft Corporation, "MSDN Library" ed. "Visual Studio 2008 Beta 2"
8. Bonfatti, Monari, Sampieri – "IEC 1131-3 Programming Methodology", Altersys
9. Steven Holzner – "XML Tutto e oltre", Apogeo
10. Elizabeth Castro – "HTML 4 per il World Wide Web", Addison Wesley Longman Italia
11. Bjarne Stroustrup – "The C++ Programming language" (3rd edition), Addison Wesley
12. Brian W. Kernighan, Dennis M. Ritchie – "Linguaggio C" (2a edizione italiana), Jackson Libri
13. Alessandra D'Alessio – "Lezioni di calcolo numerico e MATLAB", Liguori
14. Free Software Foundation, Inc. – "GNU General Public License version 2 June 1991"
15. Eric Steven Raymond "The Cathedral and the Bazaar" (3rd edition), http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/
16. I. Miyazawa, T. Nagao, M. Fukagawa, Y. Itoh, T. Mizuya, T. Sekiguchi – "Implementation of Ladder Dagram for programmable controllers using FPGA", IEEE Online
17. V. Carl Hamacher, Zvonko G. Vranesic, Safwat G. Zaky – "Introduzione all'architettura dei calcolatori", McGraw-Hill

18. Thomas E. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein – "Introduzione agli algoritmi e strutture dati", McGraw-Hill
19. Giorgio Bruno – "Linguaggi formali e compilatori", UTET Libreria
20. Bruno Fadini, Aldo Esposito – "Teoria e progetto delle reti logiche", Liguori
21. Robert Wareham – "LD and SFC in PC", IEEE Online
22. Shogo Nakamura, Yasumaw Fuji, Takishi Sekiguchi – "Study of a transformation method of Ladder Diagram into Sequential Function Chart on the basis of Linear Programming technique", IEEE Online
23. Pasquale Chiacchio, Francesco Basile – "Tecnologie informatiche per l'automazione" (2ª edizione), McGraw-Hill
24. Guida in linea del tool Internet TRiLOGI vers. 5.3, Triangle Research International, Inc
25. Apple Computer Inc – "The Objective C Programming Language", parte della collana "Inside Mac OS X"
26. Cay S. Horstmann, Gary Cornell – "Java 2 i fondamenti", The Sun Microsystems Press e McGraw-Hill Italia
27. Cay S. Horstmann, Gary Cornell – "Java 2 tecniche avanzate", The Sun Microsystems Press e McGraw-Hill Italia
28. International Organization for Standardization e International Electrotecnhical Committee – standard 23270:2003 "The C# Programming Language", http://standards.iso.org/ittf/PubliclyAvailableStandards/c036768_ISO_IEC_23270_2003(E).zip; European Computer Manufacturers Association – standard 334 "The C# Programming Language", http://msdn2.microsoft.com/it-it/netframework/Aa569283.aspx
29. European Computer Manufacturers Association – standard 335, "The Common Language Infrastructure", http://msdn2.microsoft.com/it-it/netframework/Aa569283.aspx
30. http://www.mono-project.com
31. Frederick P. Brooks – "The mythical man month, essays on software engineering", Addison Wesley
32. Bruno Fadini, Ugo De Carlini – "Macchine per l'elaborazione dell'informazione", Liguori
33. Donald E. Knuth – "The art of computer programming, Vol. 1 Fundamental Algorithms" (3rd edition), Addison Wesley