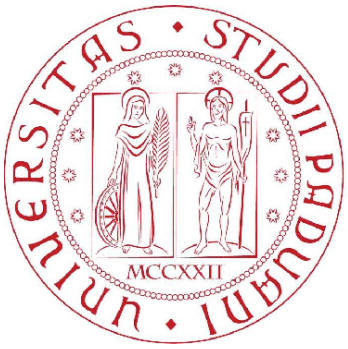
UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA



Sviluppo di una nuova interfaccia LabVIEW per il sistema di acquisizione dati MDSplus

Laureando: Enrico De Marchi
Relatore: Ch.mo Prof. Michele Moro
Correlatore: Ch.mo Prof. Gabriele Manduchi

Anno accademico 2011/2012

Sommario

L'obiettivo di questa tesi è la realizzazione di una nuova interfaccia software per rendere direttamente accessibili nell'ambiente di sviluppo LabVIEW[1] le più recenti funzionalità del sistema di acquisizione dati MDSplus, largamente utilizzato nell'ambito della ricerca scientifica e tecnologica sulla fusione termonucleare controllata.

Il progetto nasce dalla volontà di National Instruments Italy di ampliare la propria offerta commerciale a favore dei grandi centri di ricerca che operano in questo settore, accostando all'hardware di acquisizione dati una nuova interfaccia software per MDSplus pienamente supportata da LabVIEW, per uno sviluppo integrato delle applicazioni di acquisizione e gestione dei dati sperimentali. Uno dei centri di ricerca che ha maggiormente contribuito allo sviluppo di MDSplus è l'Istituto Gas Ionizzati del CNR[2], che guida il progetto con l'obiettivo di estendere la fruibilità del proprio sistema per facilitarne l'ulteriore diffusione.

L'idea fondamentale che ha guidato lo sviluppo dell'interfaccia è il riutilizzo di codice già testato ed una successiva integrazione dello stesso in LabVIEW. Per concretizzare questa idea è stata dapprima creata una libreria che esportasse tutte le funzionalità di MDSplus a partire dal codice sorgente C++, è stato quindi implementato il collegamento tra le funzioni della libreria e i corrispettivi *VI*[3] e, per finire, è stata ricreata in LabVIEW, utilizzando *LVOOP*[4], la struttura delle classi e dei metodi presenti nell'implementazione originale.

L'effettivo funzionamento della nuova interfaccia è stato verificato tramite un'applicazione che simula l'acquisizione di segnali reali e utilizza alcune delle principali funzionalità di MDSplus, come l'acquisizione segmentata, per la scrittura dei dati su disco e per una successiva visualizzazione. I risultati sono stati positivi e la nuova interfaccia LabVIEW di MDSplus verrà utilizzata per un'applicazione di acquisizione dati reale nell'ambito dell'esperimento RFX[5].

[1] LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench) è l'ambiente di sviluppo integrato per il linguaggio di programmazione grafica di National Instruments [1].

[2] Consiglio Nazionale delle Ricerche, area della ricerca di Padova.

[3] VI (Virtual Intrument) è sinonimo di funzione nell'ambiente di sviluppo LabVIEW.

[4] *LVOOP* (*LabVIEW Object Oriented Programming*) è l'implementazione nativa della programmazione orientata agli oggetti supportata a partire da LabVIEW 8.20 [2].

[5] RFX (Reversed Field eXperiment) è la macchina toroidale per la produzione di plasmi in uso presso l'Istituto Gas Ionizzati [3].

Indice

1 Introduzione

1.1 Scopo della tesi

MDSplus è il sistema più utilizzato per l'acquisizione e la gestione dei dati sperimentali nell'ambito della ricerca sulla fusione termonucleare controllata. Sviluppato congiuntamente da Massachussetts Institute of Technology, Istituto Gas Ionizzati e Los Alamos National Laboratory, è attualmente installato in oltre 30 siti dislocati in quattro continenti. MDSplus rende possibile il salvataggio di tutti i dati sperimentali in un'unica e intuitiva struttura dati organizzata gerarchicamente e permette all'utente di costruire basi di dati sperimentali complete e coerenti [4, MDS]. Recentemente è stata sviluppata una versione del sistema orientata agli oggetti e sono state aggiunte numerose funzionalità, tra cui alcuni metodi che rendono possibile l'acquisizione segmentata dei segnali, per garantire la riuscita del processo di salvataggio dei dati anche in esperimenti di durata considerevole [MLP, MOO]. Un'interfaccia LabVIEW per la versione precedente del sistema era già disponibile, ma la nuova versione di MDSplus l'ha resa obsoleta. Da più parti è quindi giunta la richiesta di una nuova interfaccia, che il gruppo di sviluppo di MDSplus, in particolare l'Istituto Gas Ionizzati, e National Instruments Italy, che fornisce l'hardware di acquisizione all'IGI[1] e ad altri grandi centri di ricerca in Europa, hanno deciso congiuntamente di soddisfare promuovendone lo sviluppo attraverso questa tesi.

1.2 Ambiente di lavoro

Una prima fase dell'attività di tirocinio, di carattere esclusivamente formativo, è stata svolta presso la sede di National Instruments Italy, che ha finanziato due corsi di formazione su LabVIEW (LabVIEW Core 1 e LabVIEW Core 2) e ha promosso un seminario sull'acquisizione dati e sul controllo avanzato in applicazioni industriali. Il coordinatore del progetto per National Instruments Italy è stato l'Ing. Augusto Mandelli, responsabile commerciale per il segmento *Big Physics* in Europa. L'attività di progettazione e sviluppo è stata invece svolta, per la maggior parte del tempo, presso l'Istituto Gas Ionizzati del CNR, situato nell'Area della Ricerca di Padova. Il Coordinatore del progetto per il CNR è stato il Prof. Gabriele Manduchi, che ha contribuito in modo sostanziale alla realizzazione di MDSplus e ha sviluppato la più recente versione del sistema, basata su un'architettura orientata

[1] Istituto Gas Ionizzati.

agli oggetti, per la quale è stata creata l'interfaccia LabVIEW obiettivo del progetto di tesi.

1.2.1 Istituto Gas Ionizzati e Consorzio RFX

L'Istituto Gas Ionizzati costituisce l'asse portante del Consorzio RFX (CNR, ENEA[2], Università degli Studi di Padova, Acciaierie Venete S.p.A. e INFN[3]), fondato a Padova nel 1996 per rendere più efficace e stabile la collaborazione tra i vari enti e imprese nell'ambito dell'associazione con Euratom[4]. Scopo dell'iniziativa è incrementare studi e ricerche di ingegneria e fisica sulla fusione termonucleare controllata sviluppando conoscenze sui plasmi con regimi di densità, corrente e fluttuazioni attualmente inesplorati e contribuire al progetto ITER[5], realizzando dispositivi complessi per il riscaldamento addizionale del plasma. L'Istituto è accorpato nel Consorzio, che ne garantisce la dimensione critica, mentre l'associazione con Euratom assicura la base del collegamento internazionale, nonché la collaborazione e l'integrazione programmatica con gli altri istituti di ricerca italiani operanti nello stesso settore. In totale RFX conta su un team di oltre 140 persone. Il Consorzio opera negli edifici sede dell'IGI, presso l'Area della Ricerca del CNR di Padova [5].

1.2.2 National Instruments

National Instruments Corporation è una società americana con oltre 5000 dipendenti e filiali in più di 40 paesi. Con sede a Austin, Texas, produce soluzioni innovative per la progettazione, la prototipazione e il rilascio di sistemi di test e controllo. Tra i principali prodotti software vi sono LabVIEW e LabWindows/CVI[6], tra i principali prodotti hardware si annoverano i moduli VXI[7], i moduli PXI[8], le interfacce per GPIB[9] e per I^2C[10] e altri standard per l'automazione industriale. Degni di nota i controller embedded real-time, tra cui CompactRIO [7]. Le principali applicazioni includono l'acquisizione dati, il controllo degli strumenti e la visione artificiale. National Instruments Italy S.r.l. ha sede a Milano e collabora con diverse aziende partner dislocate su tutto il territorio nazionale [6].

[2] Agenzia nazionale per le nuove tecnologie, l'energia e lo sviluppo economico sostenibile.
[3] Istituto Nazionale di Fisica Nucleare.
[4] European Atomic Energy Community.
[5] International Thermonuclear Experimental Reactor.
[6] LabWindows/CVI è un ambiente di sviluppo integrato ANSI C che include una serie completa di strumenti per la programmazione e applicazioni di controllo e test [8].
[7] VME eXtensions for Instrumentation.
[8] PCI eXtensions for Instrumentation.
[9] General Purpose Interface Bus.
[10] Inter-Integrated Circuit.

1.3 Organizzazione del lavoro

Dopo una definizione iniziale degli obiettivi da raggiungere, degli strumenti da utilizzare e dei vincoli progettuali da rispettare, il lavoro è stato svolto in sostanziale autonomia ed è stato intervallato da incontri settimanali con il Prof. Manduchi, finalizzati a fare il punto sulla situazione in merito ai risultati intermedi ottenuti e alle eventuali problematiche incontrate, concordando i conseguenti cambi di direzione da effettuare.

L'attività svolta si può suddividere in quattro fasi, anche se tale classificazione non rispetta un'esatto ordine cronologico.

1. *Formazione*: la prima parte del lavoro è stata dedicata allo studio dell'architettura di MDSplus, alla familiarizzazione con il linguaggio di programmazione utilizzato in LabVIEW, all'approfondimento delle possibilità offerte da *LVOOP*, l'estensione alla programmazione orientata agli oggetti di LabVIEW, al processo di creazione di librerie esterne in Windows e al meccanismo di integrazione tra queste ultime e il codice prodotto in LabVIEW.

2. *Progettazione*: la seconda parte del lavoro, spesso sovrapposta con la fase di formazione, è stata dedicata allo studio di una possibile architettura che rispettasse i vincoli progettuali e fornisse una soluzione efficiente al problema della creazione dell'interfaccia LabVIEW di MDSplus.

3. *Sviluppo e documentazione*: la terza fase del lavoro è stata dedicata allo sviluppo di quanto definito nella fase di progettazione, con intersezioni con quest'ultima e con la fase successiva di test. Una parte del lavoro è stato dedicato alla documentazione delle nuove funzionalità dell'interfaccia che andavano definendosi con il procedere dello sviluppo.

4. *Test*: la quarta ed ultima fase è stata dedicata alla verifica e al *debugging* di quanto sviluppato nella fase precedente, anche attraverso la creazione di un'applicazione di acquisizione dati comprensiva di tutte le principali funzionalità della nuova interfaccia.

Verso la fine del periodo di tirocinio è iniziata una collaborazione con Diego Ravarotto, tecnologo presso l'IGI, per la realizzazione di un'applicazione di acquisizione ed elaborazione di dati sperimentali, ottenuti direttamente dai sensori di RFX, che utilizzasse la prima versione funzionante, sebbene non ancora completa, dell'interfaccia LabVIEW del sistema.

1.4 Strumenti di lavoro

Nella fase iniziale e centrale dello sviluppo della nuova interfaccia è stata utilizzata una macchina virtuale con sistema operativo Windows 7, mentre nella fase finale del lavoro è stata utilizzata una macchina fisica dell'IGI con sistema operativo Windows Xp, direttamente collegata alle schede di acquisizione dati di RFX. In entrambe

le macchine è stata installata una delle versioni più recenti di MDSplus, costruita a partire dal codice sorgente C++. La scelta di adottare Windows come piattaforma sulla quale basare lo sviluppo è stata determinata dalla maggiore diffusione di LabVIEW su questo sistema operativo. Gli ambienti di sviluppo utilizzati sono LabVIEW 2010 e Visual Studio 2008.

1.5 Struttura della tesi

In questo capitolo, il primo, viene data breve descrizione della struttura della tesi e del contesto in cui questa si è svolta. Nel capitolo 2 verranno illustrate le funzionalità e l'architettura di MDSplus. Nel capitolo 3 verrà introdotto l'ambiente di sviluppo LabVIEW e verranno descritti in maggior dettaglio i principali costrutti utilizzati nella fase di implementazione. Nel capitolo 4 verranno illustrati i vincoli progettuali, l'architettura proposta e i dettagli implementativi, nel capitolo 5 verrà illustrata la realizzazione di un'applicazione di acquisizione dati segmentata che utilizza le nuove funzionalità di MDSplus. Nell'ultimo capitolo, il 6, si riassumeranno i risultati dello studio e si cercherà di trarre le conclusioni sul lavoro svolto, proponendo possibili sviluppi futuri.

1.6 Prerequisiti

Per una lettura consapevole di questa tesi è necessario conoscere i fondamenti della programmazione grafica in LabVIEW, i fondamenti della programmazione orientata agli oggetti, soprattuto nella sua declinazione C++, i fondamenti dell'architettura degli elaboratori, dei sistemi operativi e della teoria dei segnali.

2 MDSplus

Questo capitolo introduce le principali caratteristiche e funzionalità di MDSplus e descrive le novità introdotte dalla nuova interfaccia *object-oriented* del sistema. Una trattazione dettagliata è reperibile sul sito ufficiale di MDSplus www.mdsplus.org.

2.1 Introduzione

MDSplus è stato progettato per offrire ai ricercatori uno strumento in grado di produrre basi di dati sperimentali complete, coerenti e intuitive, che siano accessibili in un ambiente distribuito. Flessibile e facilmente estensibile, MDSplus permette la memorizzazione nella stessa struttura dati di tutte le informazioni associate ad un esperimento o ad una simulazione e le rende accessibili attraverso l'utilizzo dello stesso insieme di funzioni. Unificando la descrizione della configurazione degli apparati sperimentali con la descrizione e la programmazione delle attività e con tutti i dati acquisiti e successivamente rielaborati, MDSplus rende più agevole la condivisione dei dati tra le applicazioni, facilitando la collaborazione nell'attività di ricerca [9].

2.1.1 Sviluppo di applicazioni *data-driven*

In MDSplus la definizione dei dati che verranno utilizzati non viene inclusa nel codice, ma è interamente contenuta nelle strutture dati, per rendere possibile la creazione di applicazioni flessibili che possono essere facilmente adattate e modificate senza dover mettere mano ai sorgenti: la definizione di procedure per l'elaborazione e la visualizzazione di segnali, associate ai relativi parametri di controllo, sono infatti interamente contenute nelle strutture dati e possono essere utilizzate per creare un nuovo applicativo senza bisogno di scrivere ulteriore codice. Una delle più importanti applicazioni *data-driven*[1] offerte da MDSplus è l'acquisizione dati, che viene supportata da strumenti di *scheduling* e *dispatching* ed è affiancata da procedure di analisi dei dati [9].

2.1.2 Un nuovo paradigma per l'analisi dei dati

L'approccio tradizionale all'elaborazione dei dati prevedeva che in ogni fase fossero definiti distintamente l'insieme di interfacce di accesso, il tipo di strutture dati e le varie funzioni di visualizzazione. In Figura 2.1 si può osservare il flusso di esecuzione

[1] La programmazione *data-driven* opera una netta distinzione tra il codice sorgente e le strutture dati sulle quali lo stesso andrà ad agire, in modo tale che gli eventuali cambiamenti alla logica del programma vengano effettuati modificando le strutture dati anziché il codice sorgente [10].

di un particolare tipo di analisi (TRANSP) su dati sperimentali, portata a termine tramite metodologie di tipo tradizionale: la presenza di numerose tipologie di file e interfacce rende difficile la condivisione degli strumenti o il confronto diretto dei risultati ottenuti da diverse piattaforme di analisi. MDSplus definisce un nuovo paradigma per l'elaborazione dei dati ottenuti da attività sperimentali o di simulazione, che rende possibile un facile confronto tra dati provenienti da sorgenti diverse ed una immediata condivisione degli strumenti di analisi. In Figura 2.2 si può osservare che, tramite MDSplus, in ogni fase dell'elaborazione i dati vengono salvati nello stesso formato utilizzando un'unica API[2] e possono essere caricati e visualizzati dallo stesso insieme di funzioni [9].

2.2 Caratteristiche e funzionalità

MDSplus fornisce il supporto per l'acquisizione dati e la descrizione di attività sperimentali, per operazioni di *scheduling* e *dispatching*, per la generazione e distribuzione di eventi asincroni e per il salvataggio e la gestione dei dati [9].

2.2.1 Acquisizione e analisi dei dati

MDSplus fornisce un insieme di strumenti per l'acquisizione e l'analisi dei dati derivanti da esperimenti a impulso. L'intero processo di acquisizione e analisi è guidato da un modello sperimentale[3] (*model*). Il *model* contiene tutte le informazioni di configurazione di un esperimento e viene utilizzato come matrice per la definizione delle strutture dati per gli impulsi (*shot*), contenenti a loro volta tutte le informazioni di configurazione e tutti i dati acquisiti e rielaborati in seguito all'impulso. I task responsabili del ciclo di acquisizione dati sono gestiti da un *dispatcher*, che inoltra le azioni ricevute (*action*) ad un *server* per una successiva esecuzione. Le *action* definiscono le operazioni da eseguire, il tempo di avvio e il *server* deputato all'esecuzione. Le *action* possono essere inoltrate al *server* in modo sequenziale, asincrono o condizionato e solitamente generano un evento specifico (*event*) quando vengono completate [12]. In Figura 2.3 è illustrato un tipico ciclo di acquisizione dati che opera secondo quanto appena descritto.

2.2.2 Gestione dei dati

MDSplus è stato progettato per offrire un ricco insieme di caratteristiche e funzioni per la gestione dei dati. Tra le più significative si possono citare la presenza di un ampio insieme di tipi di dati (semplici e compositi), di una struttura gerarchica per il salvataggio dei dati, di un livello per la descrizione dei dati, di un meccanismo di valutazione di espressioni sui dati, di strumenti per la compressione e la decompressione dei dati, di un meccanismo di accesso remoto ai dati tramite il modello *client/server*

[2] API (*Application Programming Interface*) indica una specifica destinata a definire un'interfaccia tra diverse componenti software per permettere la corretta comunicazione tra di esse [11].
[3] MDS è l'acronimo per *Model Driven System* [12].

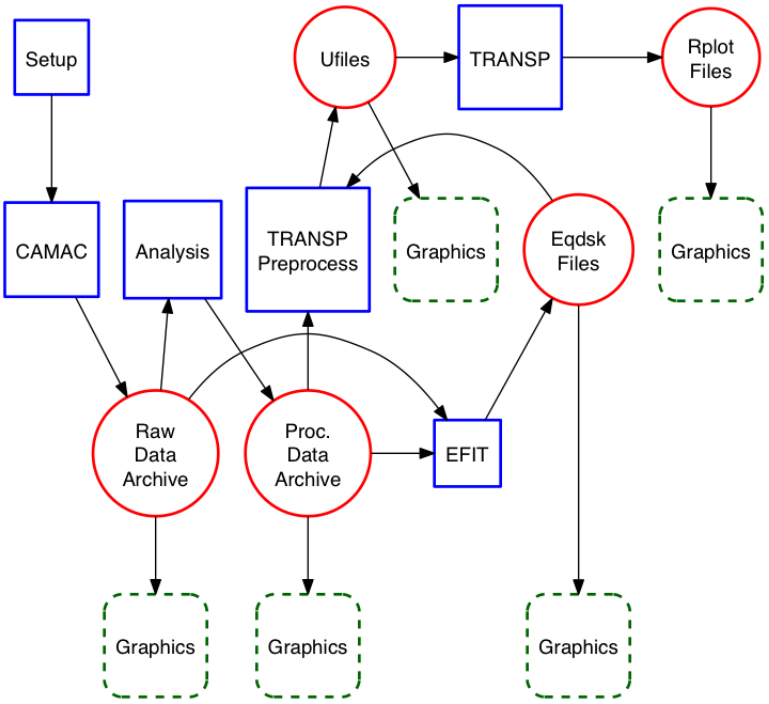


Figura 2.1: Flusso di esecuzione di una *transport analysis* (TRANSP) su dati sperimentali con un sistema tradizionale: ogni fase dell'elaborazione (riquadri continui) richiede una distinta interfaccia per ogni diverso tipo di file (cerchi) e per ogni tipologia di dato viene utilizzata un'apposita funzione di visualizzazione (riquadri tratteggiati) [9].

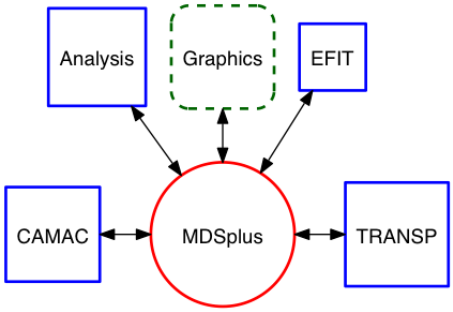


Figura 2.2: Flusso di esecuzione del processo rappresentato in Figura 2.1 che utilizza MDSplus per salvare tutti i dati relativi all'esperimento: il diagramma è sensibilmente più semplice e tutti i processi (riquadri continui) condividono le stesse interfacce (cerchi) e le stesse funzioni di visualizzazione (riquadri tratteggiati) [9].

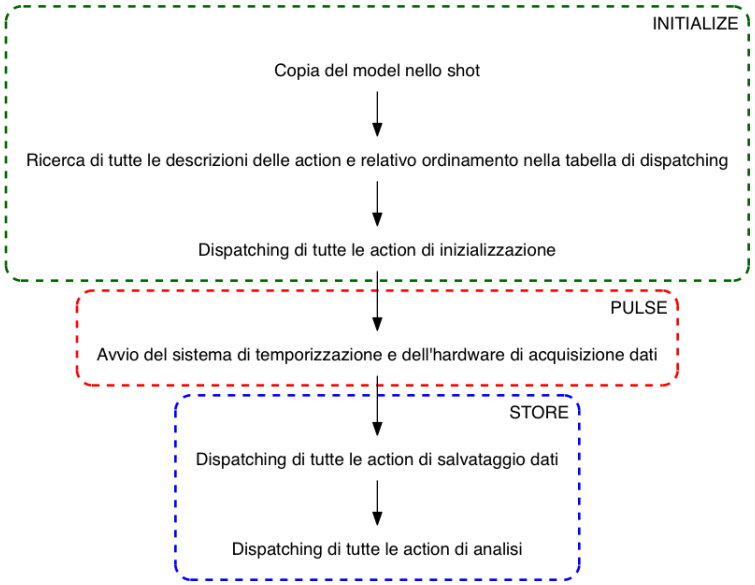


Figura 2.3: Tipico ciclo di acquisizione dati con MDSplus. Si possono distinguere tre fasi successive: una prima fase di inizializzazione (*INITIALIZE*), una seconda fase di avvio dell'esperimento (*PULSE*) ed una terza ed ultima fase di salvataggio dei dati (*STORE*), con le relative sottofasi [12].

e di un meccanismo di accesso logico ai dati. Il sistema è inoltre facilmente estensibile e scalabile su grandi moli di dati [9].

Struttura gerarchica per il salvataggio dei dati

In MDSplus i dati vengono salvati in strutture gerarchiche definite dall'utente. Questo approccio permette di rendere visibili le eventuali associazioni logiche presenti tra i dati, i quali possono essere correttamente individuati e interpretati sulla base del contesto in cui sono inseriti, per una maggiore facilità di utilizzo: grazie a questa struttura gerarchica le applicazioni possono infatti sfruttare le informazioni sul contesto dei dati che andranno ad elaborare. La presenza di questo tipo di architettura e il supporto per un'ampia varietà di tipi di dati permettono all'utente di salvare tutte le informazioni relative ad un esperimento nella stessa struttura dati [12].

I dati presenti su un particolare *server* MDSplus sono suddivisi in specifiche strutture ad albero (*tree*), ciascuna delle quali contiene un certo numero di nodi (*node*) organizzati gerarchicamente. Ogni *tree* è memorizzato in tre file distinti sul sistema operativo del *server*: il primo file contiene la struttura del *tree*, il secondo i dati, il terzo la descrizione (*characteristic*) di ogni dato presente nel *tree*. Diversi *tree* possono avere reciproche relazioni di dipendenza gerarchica oppure essere indipendenti. Per ogni *tree* sono solitamente definiti più *shot*, ciascuno dei quali corrisponde ad una particolare esecuzione di un esperimento o di una simulazione [12].

Interfaccia comune per l'accesso ai dati

In MDSplus tutti i dati sono accessibili attraverso la stessa interfaccia. L'API del sistema consiste di un numero limitato di semplici istruzioni. L'utente può usare dei comandi per accedere ad un particolare *server* o *tree* e quindi valutare il valore di un'espressione (*expression*), nella maggior parte dei casi il nome di un nodo nella struttura gerarchica. I nodi sono rappresentati dal loro percorso completo nella gerarchia, da uno dei percorsi relativi ad una posizione *default* oppure da un *tag*, alias per un particolare dato e indipendente dalla struttura gerarchica. La *characteristic* di un nodo è ottenuta valutando l'espressione che specifica il nome del nodo e il nome della caratteristica desiderata [12].

Valutazione di espressioni sui dati

L'API di MDSplus supporta la valutazione di espressioni (*expression*) scritte in un linguaggio denominato TDI (*Tree Data Interface*), che rende disponibili all'utente un gran numero di funzioni e comandi. Le espressioni più semplici sono rappresentate dal nome di un nodo e le relative valutazioni ritornano i dati contenuti nel nodo stesso. Sono supportate semplici operazioni logiche e matematiche, funzioni per la manipolazione di stringhe, semplici istruzioni e comandi per analizzare o creare specifici costrutti in MDSplus. Si possono includere anche invocazioni a procedure esterne scritte con altri linguaggi di programmazione, per una flessibilità del sistema quasi illimitata. Le espressioni TDI possono essere salvate come dati in un nodo di un *tree*. Una volta referenziato, tale nodo ritorna il risultato dei comandi TDI dell'espressione in esso contenuta [12].

Descrizione dei dati

La struttura dati di MDSplus è facilmente comprensibile e utilizzabile: in aggiunta ai dati, per ogni nodo di ciascun *tree* sono presenti informazioni che possono includere il tipo di dato contenuto, le dimensioni dell'array, la lunghezza in byte del dato, l'unità di misura associata al dato, la posizione nella gerarchia, eventuali *tag* associati, informazioni temporali sulla creazione e così via. La struttura dati può essere attraversata senza bisogno di leggere i dati contenuti nei nodi incontrati lungo il cammino. La gerarchia fornisce una sorta di documentazione attraverso le relazioni strutturali esistenti tra i nodi e le scelte dei nomi ad essi assegnati. Ogni nodo in un particolare sottoalbero del *tree* può a sua volta avere dei nodi figli corredati da commenti ed etichette [12].

Accesso remoto ai dati

MDSplus offre la possibilità di accedere ai dati da remoto attraverso il modello di comunicazione *client/server*. L'accesso è *service-oriented*: i dati sul server remoto vengono letti direttamente dalle applicazioni connesse, senza bisogno di operare trasferimento di file. Il protocollo di comunicazione si basa su TCP/IP. Per l'utente non

c'è differenza tra i comandi utilizzati per accedere ai dati residenti su un server remoto e tra quelli utilizzati per l'accesso ai dati di un server locale [12].

Tipi di dati

MDSplus supporta un ampio insieme di tipi di dati, semplici e compositi. Tra i tipi semplici ci sono gli interi con segno e senza segno fino a 64 bit, i numeri reali a precisione singola e doppia, stringhe e array. Due tra i più importanti tipi compositi includono i segnali (*signal*), che contengono dati associati ai relativi assi indipendenti, e i dispositivi (*device*), che vengono utilizzati per combinare impostazioni di configurazione, *action* e dati per l'acquisizione o l'analisi. I *device* forniscono un meccanismo ben strutturato per implementare applicazioni *data-driven* [12].

2.2.3 Strumenti di gestione e visualizzazione

MDSplus fornisce una serie di strumenti software per la gestione dei *tree* e per la visualizzazione dei segnali acquisiti.

jTraverser

jTraverser è uno strumento software sviluppato in Java per la visualizzazione e la gestione della struttura di un *tree*. *jTraverser* implementa e rende disponibili all'utente tutte le funzionalità di gestione dei dati precedentemente descritte. L'aspetto dell'interfaccia grafica della finestra principale di *jTraverser* è mostrato in Figura 2.4.

jScope

jScope è uno strumento software sviluppato in Java per la visualizzazione dei segnali acquisiti con MDSplus. Con *jScope* è possibile definire un insieme di pannelli di visualizzazione e associare un segnale ad ogni pannello. Il pannello permette di eseguire operazioni di ingrandimento su aree specifiche del segnale, di ottenere le esatte coordinate di un particolare punto del segnale e di scorrere il segnale [23]. L'aspetto dell'interfaccia grafica della finestra principale di *jScope* è mostrato in Figura 2.5.

2.3 Una nuova interfaccia *object-oriented*

Una delle principali ragioni del successo di MDSplus è la semplicità di utilizzo della sua API, che garantisce un facile accesso ai dati nascondendo la complessità dell'organizzazione interna degli stessi. Nonostante questa semplicità di utilizzo, l'insieme dei tipi di dati supportati è molto ampio e include, oltre a quei tipi di dati utilizzati per descrivere le grandezze fisiche in gioco, anche tutte quelle strutture dati necessarie alla descrizione della configurazione dell'esperimento e alla definizione delle operazioni da portare a termine durante l'esecuzione dello stesso. Se da un lato la ricchezza dei tipi di dati disponibili permette di soddisfare il bisogno di descrivere una grande varietà di informazioni distinte, dall'altro la complessità che ne deriva va a limitare la crescita

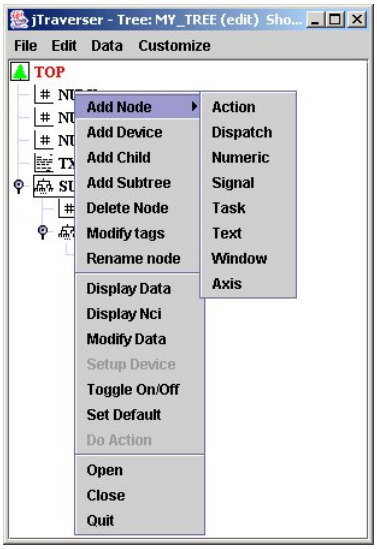


Figura 2.4: Screenshot della finestra principale di *jTraverser*. Fonte dell'illustrazione: http://www.mdsplus.org/images/2/28/JTraverser4.jpg.

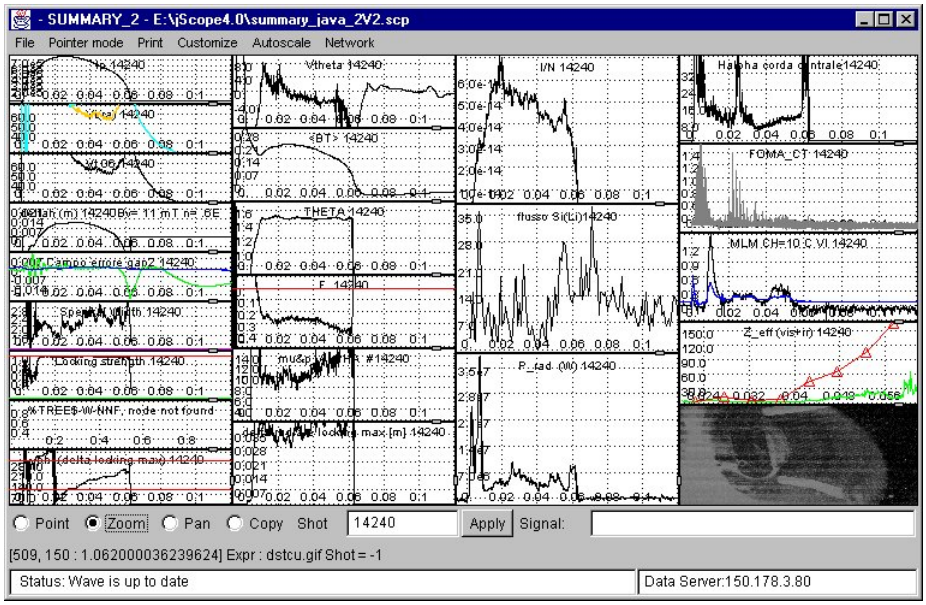


Figura 2.5: Screenshot della finestra principale di *jScope*. Fonte dell'illustrazione: http://www.mdsplus.org/documentation/tutorial/jScope.jpg.

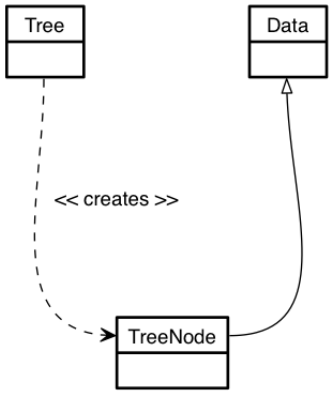


Figura 2.6: Diagramma UML delle classi utilizzate per l'accesso ai dati in MDSplus.

della comunità di sviluppatori di MDSplus. Le motivazioni di tale complessità sono da ricercare nei limiti imposti dalla potenza espressiva del linguaggio C, utilizzato per implementare il kernel del sistema. Per ridurre questa complessità è stata sviluppata una nuova interfaccia di MDSplus orientata agli oggetti, per una migliore e più naturale descrizione dei tipi di dati e della loro combinazione in espressioni. Attualmente esiste un'implementazione completa del sistema in C++, Java e Python [MOO].

2.3.1 Classi per l'accesso ai dati

Le funzionalità di accesso ai dati in MDSplus sono implementate dalla classe *Tree* e dalla classe *TreeNode*. In Figura 2.6 sono rappresentate le relazioni di interdipendenza tra le due classi.

La classe *Tree*

Un *tree* è rappresentato da un'istanza della classe *Tree*. Tale istanza è costruita a partire dal nome del particolare *tree*, dal numero di *shot* considerato e dalla modalità operativa, che può avere i seguenti valori [MOO, 21]:

- *NORMAL*: il *tree* specificato viene aperto per operazioni di lettura e scrittura e non sono permesse operazioni di modifica alla struttura dello stesso.
- *READONLY*: il *tree* specificato viene aperto per sole operazioni di lettura.
- *NEW*: viene creato un nuovo tree *ex novo*.
- *EDIT*: il *tree* specificato viene aperto per operazioni di lettura, scrittura e modifica alla struttura dello stesso.

Per modificare la struttura di un *tree*, ovvero per eseguire operazioni di aggiunta di nuovi *node* o *tag* o di rimozione e modifica di *node* esistenti, è necessario aprire il *tree* in modalità *EDIT*, mentre per costruire un nuovo *tree* da zero è necessario aprirlo in

modalità *NEW*. I seguenti metodi della classe *Tree* sono i più utilizzati per la modifica del *tree* ad essa associato: con il metodo *addNode()* viene aggiunto un nuovo nodo al *tree*, con il metodo *deleteNode()* vengono eliminati il *node* specificato e tutti i suoi discendenti, con il metodo *renameNode()* viene rinominato il *node* specificato, mentre con il metodo *write()* vengono salvate tutte le modifiche apportate al *tree* corrente su disco [MOO, 21].

La classe *TreeNode*

La classe *TreeNode*, derivata dalla classe astratta *Data*, rappresenta un *node* in un *tree*. Le istanze della classe *TreeNode* sono ottenute da un'istanza di *Tree* attraverso il metodo *getNode()*, che prende come parametro in ingresso il percorso del *node* nel *tree*. Un'istanza di *TreeNode* rappresenta il punto di accesso per il nodo ad essa associato, che può essere letto o scritto tramite i metodi *getData()* e *putData()*, rispettivamente. Un *node* di un *tree* può fornire informazioni di vario tipo attraverso i metodi della classe *TreeNode*: il percorso è ritornato dai metodi *getMinPath()*, *getPath()* e *getFullPath()*, la dimensione dei dati contenuti è ritornata dal metodo *getLength()*, la modalità di utilizzo è ritornata dal metodo *getUsage()*, il nodo padre è ritornato dal metodo *getParent()*, la lista dei *tag* associati al *node* è ritornata dal metodo *getTags()*. Il metodo *getNodeWild()* della classe *Tree* ritorna invece la lista di tutti i discendenti di un dato *node* in un *tree*, organizzati in un'array di *TreeNode*, la cui implementazione è fornita dalla classe *TreeNodeArray* [MOO, 21].

Dettagli implementativi

Nell'interfaccia ad oggetti di MDSplus non sono presenti operazioni esplicite di apertura e chiusura dei *tree* in quanto implicitamente gestite dal costruttore e dal distruttore della classe *Tree*, rispettivamente. La gestione contemporanea di *tree* distinti è resa possibile in quanto ogni nodo fa riferimento al proprio *tree*: operazioni concorrenti di lettura e scrittura su oggetti *TreeNode* derivanti da oggetti *Tree* distinti sono quindi gestite correttamente (*thread-safe*) [MOO].

2.3.2 Classi per la rappresentazione dei dati

In MDSplus un *node* può essere rappresentato da una grande varietà di tipi di dati distinti, semplici oppure aggregati in espressioni: questa varietà potrebbe rappresentare un fattore di complicazione per l'utilizzo delle funzioni di accesso, che aumenterebbero in numero per supportare ogni diverso tipo di dato, qualora non fosse fornita un'interfaccia che gestisca in modo adeguato questo elemento di complessità. La grande varietà di tipi di dati disponibili e la facilità di utilizzo delle funzioni di accesso sono due requisiti di segno opposto che possono tuttavia coesistere grazie al meccanismo dell'ereditarietà: in MDSplus, infatti, ogni tipo di dato è rappresentato da una classe derivata dalla classe astratta *Data*. In questo modo i metodi di accesso *getData()* e *putData()* della classe *TreeNode* possono ritornare in uscita e accettare in ingresso

generiche istanze di *Data*, che in realtà si riferiscono a specifiche classi derivate che rappresentano un tipo di dato ben definito [MOO].

La classe *Data*

La classe *Data* definisce alcuni metodi che sono comuni a tutti i tipi di dati e che vengono ereditati e reimplementati a livello di classe derivata. Tra questi sono comunemente utilizzati quei metodi che si occupano della conversione del dato rappresentato in un tipo di dato specifico del linguaggio di programmazione in uso, ovvero *getByte()*, *getShort()*, *getInt()*, *getLong()*, *getFloat()*, *getDouble()*, *getString()*, *getByteArray()*, *getShortArray()*, *getIntArray()*, *getLongArray()*, *getFloatArray()*, *getDoubleArray()*. Per esempio, il metodo *getInt()* ritornerà il contenuto del *node* convertito in una variabile di tipo intero, se compatibile, mentre il metodo *getDoubleArray()* ritornerà il contenuto del *node* sotto forma di array di valori in virgola mobile a precisione doppia. I programmi che utilizzano i dati salvati nei *tree* di MDSplus non hanno quindi bisogno di conoscere la rappresentazione interna dei dati, ma richiedono che il contenuto venga convertito in una variabile riconoscibile e utilizzabile dalle istruzioni del linguaggio di programmazione in uso [MOO, 21].

Sebbene in fase di lettura sia possibile sfruttare l'astrazione derivante dall'utilizzo della classe *Data*, in fase di scrittura è necessario instanziare gli oggetti delle specifiche classi derivate. L'insieme delle classi utilizzate per la rappresentazione dei dati è mostrato in Figura 2.7 [MOO].

La classe *Scalar*

Scalar è una classe astratta che descrive un generico valore di tipo scalare. Le sue classi derivate *Int8*, *Int16*, *Int32*, *Int64*, *Uint8*, *Uint16*, *Uint32*, *Uint64*, *Float32*, *Float64* e *String* descrivono tutti i tipi scalari supportati in MDSplus, ovvero, nell'ordine, gli interi con segno a 8, 16, 32 e 64 bit, gli interi senza segno a 8, 16, 32 e 64 bit, i numeri in virgola mobile a precisione singola, i numeri in virgola mobile a precisione doppia e il tipo stringa [MOO, 21].

La classe *Array*

Array è una classe astratta che descrive un generico array. Le sue classi derivate *Int8Array*, *Int16Array*, *Int32Array*, *Int64Array*, *Uint8Array*, *Uint16Array*, *Uint32Array*, *Uint64Array*, *Float32Array*, *Float64Array* e *StringArray* descrivono i tipi di array supportati in MDSplus, ovvero, nell'ordine, gli array di interi con segno a 8, 16, 32 e 64 bit, gli array di interi senza segno a 8, 16, 32 e 64 bit, gli array di numeri in virgola mobile a precisione singola, gli array di numeri in virgola mobile a precisione doppia e gli array di stringhe [MOO, 21].

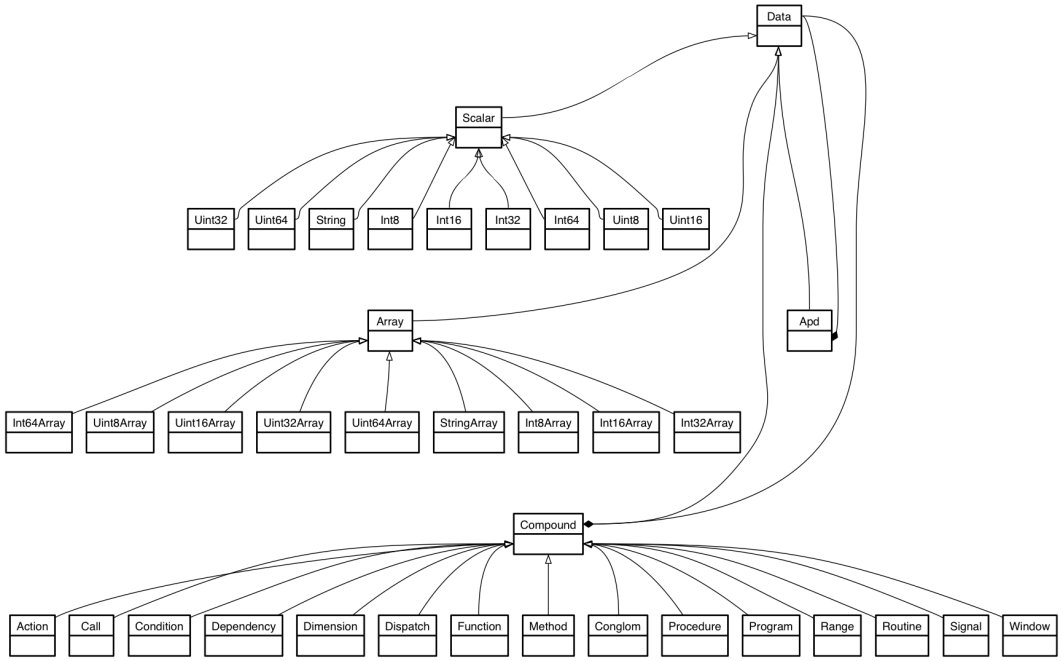


Figura 2.7: Diagramma UML delle classi utilizzate per la rappresentazione dei dati in MDSplus.

La classe *Compound*

Compound è la classe astratta progenitrice di tutte le classi di MDSplus che rappresentano espressioni e dati compositi. Le sue classi derivate sono *Action*, *Call*, *Condition*, *Dependency*, *Dimension*, *Dispatch*, *Function*, *Method*, *Conglom*, *Procedure*, *Program*, *Range*, *Routine*, *Signal* e *Window*. Un'istanza della classe *Compound* contiene uno o più riferimenti ad altre classi di tipo *Data*. Un esempio di classe derivata da *Compound* è la classe *Function*, che definisce un'operazione applicata ad un insieme di argomenti, rappresentati anch'essi da istanze di *Data* collegate all'istanza di *Function* considerata. Un'altra classe molto usata e derivata da *Compound* è la classe *Signal*, utilizzata per la rappresentazione di segnali [MOO].

La classe *Apd*

Apd (*Array of Pointers to Descriptors*) è la classe usata per rappresentare liste di *node* eterogenei. Utilizzando la classe *Apd*, che fa da contenitore per un numero variabile di istanze di classi derivate dalla classe *Data*, è possibile leggere e scrivere nei *tree* di MDSplus strutture dati generiche. La classe *Apd* è usata estensivamente quando si lavora con Python, essendo l'utilizzo di strutture dati eterogenee molto comuni in questo linguaggio [MOO].

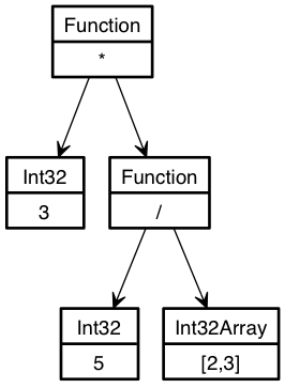


Figura 2.8: Gerarchia delle istanze di *Data* corrispondente alla rappresentazione dell'*expression* `3*(5/[2,3])` [MOO].

2.3.3 Espressioni

Per la creazione di *expression* è necessario definire in memoria un albero di istanze di classi derivate da *Data* collegate tra loro. Il modo più semplice per creare questo tipo di rappresentazione è utilizzare il metodo statico *compile()* definito nella classe astratta *Data*. Il metodo *compile()* prende in ingresso una stringa che rappresenta una *expression* di MDSplus e produce una struttura gerarchica in memoria con tutte le istanze delle classi derivate da *Data* che rappresentano l'*expression* considerata. Per esempio, `Data.compile(‘‘3’’)` ritornerà una singola istanza della classe *Int32*, mentre `Data.compile(‘‘3*(5/[2,3])’’)` ritornerà la gerarchia di istanze di *Data* rappresentata in Figura 2.8 [MOO].

2.3.4 Acquisizione continua dei dati

Solitamente un'operazione di scrittura in un *node* prevede la sostituzione del contenuto eventualmente preesistente con i nuovi dati disponibili. Esiste tuttavia anche una diversa strategia di salvataggio, che prevede l'aggiunta di un nuovo blocco di dati a quanto attualmente memorizzato nel *node*. Questa strategia viene utilizzata per gestire processi di acquisizione di lunga durata in cui non è possibile tenere in memoria tutti i dati acquisiti per salvarli sul *node* con un'unica operazione di scrittura, ma è necessario salvare di volta in volta blocchi successivi di dati appena vengono resi disponibili dall'hardware di acquisizione. I dati e i blocchi di dati già scritti nel *tree* sono disponibili per operazioni di lettura in qualsiasi momento [21].

In MDSplus l'acquisizione continua dei dati è ottenuta attraverso il concetto di segmento (*segment*). Nuovi segmenti di dati possono essere aggiunti in coda ad una sequenza di segmenti già acquisiti. Ogni segmento viene identificato dai seguenti elementi [21]:

- *Start*: istante temporale di inizio del segmento (in *sec*).

Algoritmo 2.1 Esempio di acquisizione continua di dati

```
void simple_segment_acquisition()
{
  float data[1000];
  float currData[100];
  Tree *tree = new Tree("my_tree", -1);
  TreeNode *node = tree->getNode("SIG_1");
  Data *start = new Float64(10.);
  Data *end = new Float64(11.);
  Data *samplingPeriod = new Float64(1E-3);
  Data *dimension = new Range(start, end, samplingPeriod);
  Array *dataArray = new Float32Array(data, 1000);
  node->beginSegment(start, end, dimension, data);
  for(int i = 0; i < 10; i++)
  {
    getSamplesFromHardwareBuffer(&currData);
    Array *subArr = new Float32Array(currData, 100);
    n->putSegment(subArr, -1);
    deleteData(subArr);
  }
  deleteData(dimension);
  deleteData(dataArray);
  delete(node);
  delete(tree);
}
```

- *End*: istante temporale di fine del segmento (in *sec*).
- *Dimension*: base dei tempi utilizzata per l'acquisizione del segmento (in *sec*).
- *Data*: segmento contenente i dati acquisiti.

L'elemento *Data* di un segmento deve essere rappresentato da una classe di tipo array, mentre *Start*, *End* e *Dimension* possono essere rappresentati da espressioni generiche. Le informazioni sull'inizio e sulla fine del segmento vengono utilizzate per accedere su disco in modo efficiente a singole porzioni di segnale, limitando l'operazione di lettura all'intervallo temporale selezionato [21].

L'Algoritmo 2.1 riporta un frammento di codice C++ che esemplica un tipico processo di acquisizione continua dei dati: si tratta dell'aggiunta al contenuto del *node* `SIG_1` nel *tree* `my_tree` di un segmento di dati ottenuti da un processo di acquisizione della durata di `1 sec` su un segnale campionato a `1 kHz`, tenendo conto che l'hardware di acquisizione rende disponibili blocchi di `100` campioni per volta tramite la procedura *getSamplesFromHardwareBuffer()*. Osservando il codice si può dedurre che il segmento, inizializzato dal metodo *beginSegment()*, viene progressivamente riempito richiamando il metodo *putSegment()*, che prende in ingresso l'array parziale dei dati in arrivo e un indice di riga che specifica la posizione di salvataggio. In questo caso l'indice di riga è pari a `-1`, per indicare che i dati devono essere aggiunti alla fine della porzione di segmento già memorizzata [21].

2.4 Piattaforme e linguaggi utilizzati

MDSplus è stato installato e testato sulle seguenti piattaforme: AIX[4], Compaq Tru64 Unix[5], HP-UX[6], IRIX[7], Linux, OpenVMS[8], SunOS[9], Windows, Mac OS X. Tra i linguggi utilizzati per sviluppare applicazioni basate su MDSplus si possono trovare Fortran, C, C++, Java, IDL[10], Matlab, Visual Basic, LabVIEW, PHP, Python [13]. Per una lista dettagliata degli enti di ricerca che utilizzano MDSplus si faccia riferimento alla tabella presente in http://www.mdsplus.org/index.php/Introduction:Sites.

[4] AIX (Advanced Interactive eXecutive) è una serie proprietaria di sistemi operativi Unix sviluppati e commercializzati da IBM [14].
[5] Tru64 Unix è un sistema operativo Unix a 64 bit attualmente posseduto da Hewlett-Packard [15].
[6] HP-UX (Hewlett-Packard UniX) è un'implementazione del sistema operativo Unix sviluppata da Hewlett-Packard [16].
[7] IRIX è un'implementazione del sistema operativo Unix sviluppata da Silicon Graphics [17].
[8] OpenVMS (Open Virtual Memory System) è un sistema operativo server sviluppato da Digital Equipment Corporation [18].
[9] SunOS è un'implementazione del sistema operativo Unix sviluppata da Sun Microsystems [19].
[10] IDL (Interactive Data Language) è un linguaggio di programmazione usato per l'analisi dei dati [20].

3 LabVIEW

Questo capitolo introduce i concetti fondamentali di LabVIEW e i principali costrutti utilizzati per lo sviluppo della soluzione proposta. Per una comprensione approfondita di quanto trattato è necessario avere familiarità con la programmazione in LabVIEW, nonché una conoscenza base dei principi della programmazione ad oggetti e del linguaggio C++.

3.1 Introduzione

LabVIEW è un ambiente di sviluppo per il linguaggio di programmazione grafica G di National Instruments. Originariamente prodotto per Macintosh nel 1986, LabVIEW è comunemente utilizzato per lo sviluppo di applicazioni di acquisizione dati, controllo di strumenti e automazione industriale su un ampio numero di piattaforme, tra cui Windows, varie versioni di Unix, Linux e Mac OS X. LabVIEW offre la possibilità di creare applicazioni *stand-alone* distribuite che comunicano tra loro sfruttando il modello *client/server*. L'ultima versione di LabVIEW è LabVIEW 2011, uscita nell'Agosto 2011 [1].

3.1.1 Il modello di programmazione *dataflow*

Il linguaggio G utilizzato in LabVIEW è caratterizzato quasi esclusivamente da un modello di programmazione *dataflow*, ovvero a flusso di dati. L'esecuzione è determinata dalla struttura di un *block diagram*[1], sul quale il programmatore dispone dei nodi, rappresentanti delle specifiche funzioni utilizzate, e li connette disegnando dei fili. I fili propagano i contenuti delle variabili del programma da un nodo (o da un controllo) ai nodi o agli indicatori ad esso connessi. Ogni nodo può essere processato non appena tutti i dati di input sono disponibili ai terminali di ingresso dello stesso. Il linguaggio G supporta la descrizione di operazioni concorrenti: nodi diversi possono infatti essere pronti all'esecuzione nello stesso istante [1].

3.1.2 Concetti di programmazione grafica

In LabVIEW la creazione di interfacce utente, denominate *front panel*, è integrata nel processo di sviluppo. I programmi e le funzioni sono chiamati *VI* (*Virtual Instrument*). Ogni *VI* è formato da tre componenti: un *block diagram*, un *front panel* e un *connector pane*. Il *connector pane* è utilizzato per definire i terminali dei parametri

[1] In LabVIEW un *block diagram* è il contenitore del codice sorgente G per una data funzione o un dato programma.

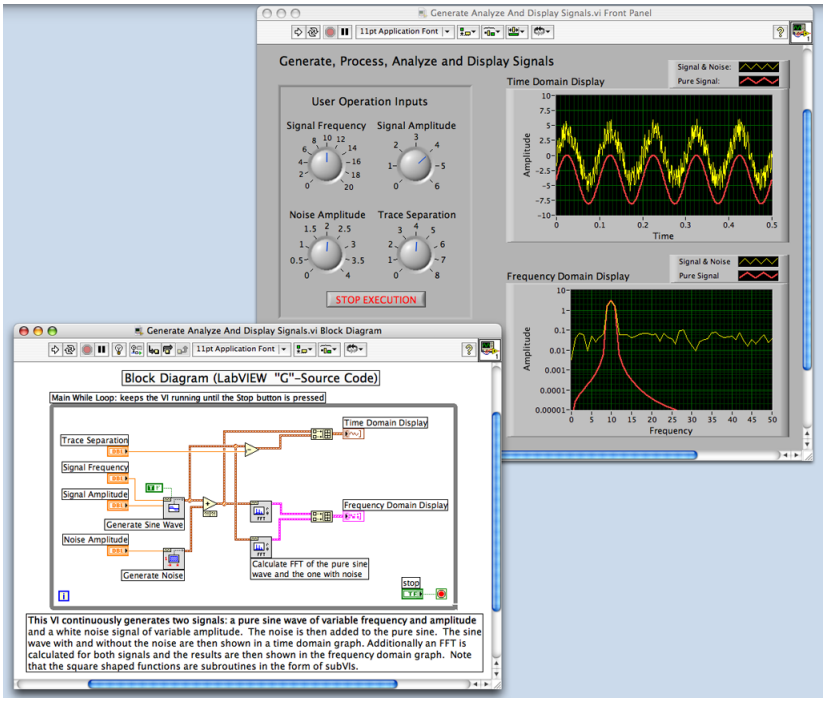


Figura 3.1: Screenshot del *front panel* (in alto a destra) e del *block diagram* (in basso a sinistra) di un semplice programma LabVIEW che genera, elabora e visualizza segnali sinusoidali [1]. Fonte dell'illustrazione: http://en.wikipedia.org/wiki/File:WikipediaFPandBD.png.

di ingresso o di uscita di un *VI*, in modo che possa essere eventualmente utilizzato come *subVI*[2]. *Controlli* e *indicatori* posizionati sul *front panel* di un *VI* permettono all'utente di impostare i dati di ingresso e leggere i dati di uscita. Il *front panel* può anche essere utilizzato come una semplice interfaccia per il codice contenuto nel corrispettivo *block diagram*: un *VI* può essere infatti eseguito come programma, con il *front panel* che fa da interfaccia utente, oppure come *subVI* inserito nel *block diagram* di un altro *VI*, che lo utilizzerà come proprio nodo di elaborazione. Questo approccio grafico di codifica permette anche ad utenti con scarse competenze di programmazione di costruire applicazioni di misura e controllo trascinando, posizionando e collegando tramite fili le rappresentazioni virtuali della strumentazione di laboratorio con cui hanno familiarità su *front panel* e *block diagram* della loro applicazione. La Figura 3.1 è un'illustrazione di un semplice programma LabVIEW che mostra il codice sorgente *dataflow* in forma di *block diagram* nella finestra in basso a sinistra e i controlli e gli indicatori delle variabili di ingresso e uscita in forma di *front panel* nella finestra in alto a destra [1].

[2]Con il termine *subVI* si indica un *VI* inserito nel *block diagram* di un altro *VI*.

3.2 *LVOOP*

La programmazione orientata agli oggetti ha dimostrato nel corso del tempo la sua superiorità sulla programmazione procedurale: l'utilizzo del paradigma *OOP*[3] incoraggia infatti la creazione di interfacce più razionali tra sezioni distinte di codice, è più semplice da testare ed è maggiormante scalabile. La necessità di fornire ai programmatori uno strumento di sviluppo che unisca la semplicità di utilizzo di LabVIEW ai punti di forza della programazione orientata agli oggetti ha spinto National Instruments a rendere disponibile, a partire dalla versione 8.2 di LabVIEW, un nuovo strumento di sviluppo che risponda a questo tipo di esigenza, ovvero *LVOOP* (*LabVIEW Object-Oriented Programming*) [24].

3.2.1 Classi

LVOOP si basa sul concetto di *classe*. Una *classe* definisce un insieme di dati associati ad un oggetto e i metodi che possono operare su tali dati. I dati di una classe sono *privati*, ovvero accessibili solamente dai *VI* che sono membri della classe. La classe viene salvata in un *class library file* (`.lvclass`) che definisce un nuovo tipo di dato. Il *class library file* memorizza il controllo contenente i dati privati e le informazioni su tutti i *VI* membri della classe, come la lista dei nomi e le varie proprietà degli stessi (Figura 3.2). Il controllo contenente i dati privati della classe è rappresentato da un *cluster*[4] e sarà l'oggetto trasmesso nei fili collegati alla classe stessa (Figura 3.3). Attraverso la definizione del concetto di classe vengono realizzate le due principali caratteristiche su cui si fonda la programmazione ad oggetti in LabVIEW, ovvero l'*incapsulamento* e l'*ereditarietà*, che rendono possibile la creazione di applicazioni modulari facilmente estensibili e modificabili [24].

3.2.2 Incapsulamento

Ogni classe LabVIEW è rappresentata da un *cluster* di dati e dai metodi di accesso ai dati contenuti nel *cluster* stesso. I dati della classe sono privati e resi perciò inacessibili ai *VI* non appartenenti alla classe. Per utilizzare i dati privati della classe è quindi necessario definire dei *VI* di classe. L'*incapsulamento* è la creazione delle proprietà e dei metodi in una classe in cui i dati sono accessibili solamente attraverso *VI* membri della classe stessa. A differenza dei dati contenuti in una classe, sempre privati, i *VI* membri possono essere utilizzati in modalità diverse, a seconda dello *scope* di accesso che li caratterizza. Un *VI* membro può avere uno dei seguenti *scope* di accesso [26]:

- *Public scope*: il *VI* può essere utilizzato come *subVI* da qualsiasi altro *VI*.
- *Community scope*: il *VI* può essere utilizzato come *subVI* solamente da *VI* della stessa classe, da *VI* che sono definiti *friend*[5] della classe oppure da *VI* appartenenti ad una libreria definita *friend* della classe.

[3] *Object-Oriented Programming*.
[4] Un *cluster* è un tipo di dato che raggruppa elementi eterogenei.
[5] Il concetto di *friendship* è trattato in 3.2.2.

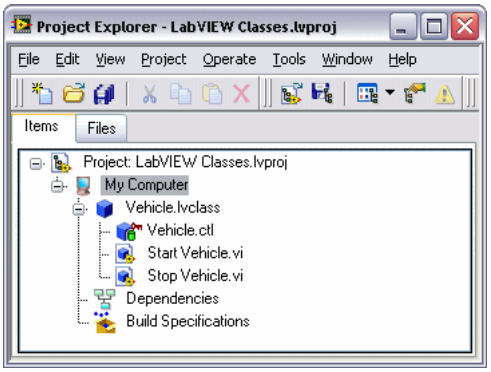


Figura 3.2: *Class library file* della classe *Vehicle* [26]. Fonte dell'illustrazione: http://zone.ni.com/images/reference/en-XX/help/371361F-01/loc_env_automobile_class.gif.

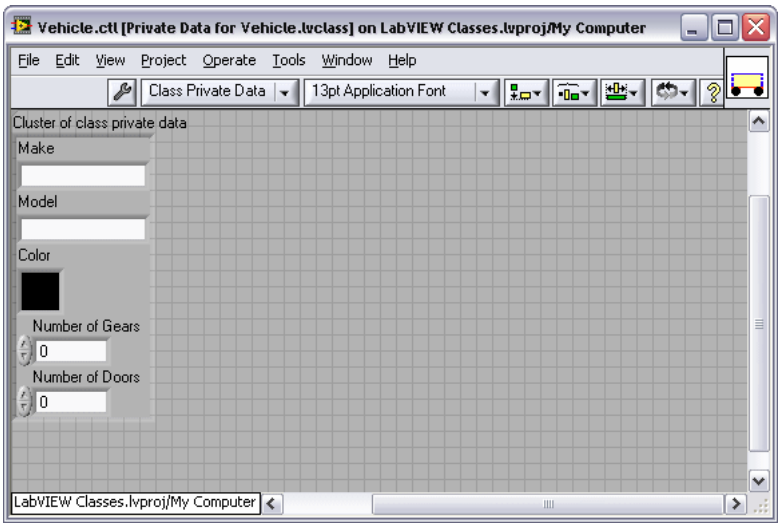


Figura 3.3: *Cluster* contenente i controlli per i dati privati per la classe *Vehicle* [26]. Fonte dell'illustrazione: http://zone.ni.com/images/reference/en-XX/help/371361F-01/loc_fp_automobile_ctl.gif.

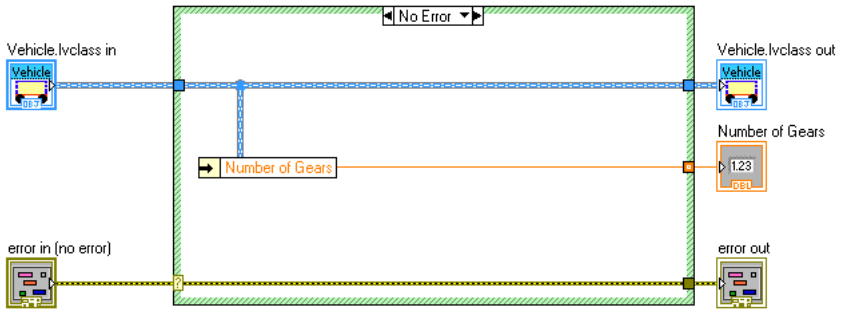


Figura 3.4: *Block diagram* del *VI* che ritorna il valore di *Number of Gears* della classe *Vehicle* [26]. Fonte dell'illustrazione: http://zone.ni.com/images/reference/en-XX/help/371361F-01/loc_bd_read_number_gears.gif.

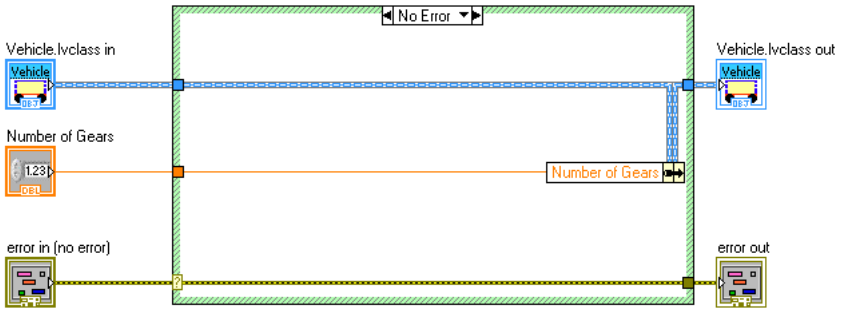


Figura 3.5: *Block diagram* del *VI* che imposta il valore di *Number of Gears* della classe *Vehicle* [26]. Fonte dell'illustrazione: http://zone.ni.com/images/reference/en-XX/help/371361F-01/loc_bd_write_number_gears.gif.

- *Protected scope*: il *VI* può essere utilizzato solamente da *VI* appartenenti alla stessa classe o a classi derivate.
- *Private scope*: il *VI* può essere utilizzato solamente da *VI* della stessa classe.

In Figura 3.4 e Figura 3.5 sono rappresentati due *VI* della classe *Vehicle*, che accedono rispettivamente in lettura e scrittura al valore di *Number of Gears*, precedentemente definito in Figura 3.3.

Friendship

Un *VI* definito *friend* di una data libreria può accedere ad ogni *VI* in *community scope* della stessa libreria. Si può anche definire un'intera libreria come *friend* di una data libreria. Il concetto di *friendship* comporta la limitazione dei punti di accesso

ad una classe ed è utilizzato per ottenere una maggiore facilità di debug del codice in virtù della diminuita possibilità di introdurre errori nei dati [26].

3.2.3 Ereditarietà

L'*ereditarietà* permette la creazione di una nuova classe a partire dell'estensione di una classe esistente. La nuova classe erediterà i dati e i *VI* della classe di partenza e potrà utilizzarli nativamente, potrà inoltre aggiungere dati o *VI* per estendere le funzionalità della classe progenitrice oppure fare l'*override* di alcuni *VI* per reimplementarne il funzionamento a livello di classe derivata. I dati della classe progenitrice sono privati e possono essere modificati solamente tramite le funzioni di accesso messe a disposizione dai *VI* dalla stessa. Per accedere a tali dati i *VI* della classe derivata possono quindi utilizzare qualsiasi *VI* in *public scope* e tutti i *VI* in *protected scope* della classe progenitrice [26].

Dynamic dispatching e *static dispatching*

In *LVOOP* i metodi sono rappresentati dai *VI* membri di una classe. Tali *VI* operano sui dati della classe cui appartengono. Un metodo può essere implementato da un solo *VI* oppure da più *VI* omonimi, appartenenti alla stessa gerarchia di classe. Nel primo caso il metodo viene definito *static dispatch*: LabVIEW utilizza infatti lo stesso *VI* ad ogni invocazione. Nel secondo caso il metodo viene definito *dynamic dispatch*: non è infatti noto quale delle diverse implementazioni dello stesso *VI* verrà eseguita da LabVIEW *runtime*. Un metodo *dynamic dispatch* è simile ad un *polymorphic VI*. Tuttavia, mentre un *polymorphic VI* determina *compile-time* quale *VI* eseguire in base ai tipi di dati collegati ai suoi terminali di ingresso, un metodo *dynamic dispatch* decide *runtime* quale *VI* della gerarchia di classe utilizzare sulla base del tipo di dato che arriva ai terminali di ingresso. Un metodo *static dispatch* si comporta invece come qualsiasi altro *subVI*. Il *connector pane* permette di definire se un *VI* è un metodo *static dispatch* oppure se fa parte di un metodo *dynamic dispatch*. Una classe derivata eredita tutti i metodi in *public scope* e *protected scope* della classe progenitrice. Per ridefinire l'implementazione di un metodo della classe progenitrice a livello di classe derivata è sufficiente creare un nuovo *VI* e rinominarlo con lo stesso nome del *VI* della classe di partenza. Per un metodo è possibile definire più *VI dynamic dispatch*, ciascuno per ogni livello della gerarchia di classe. Se un *VI dynamic dispatch* definito nella classe progenitrice viene ridefinito in una delle classi derivate, l'implementazione della classe derivata attua un'*override* oppure un'*estensione* della classe progenitrice. Nell'esempio in Figura 3.6, la classe *Vehicle* e la classe *Truck* definiscono entrambe un'implementazione di *Set Make*, un metodo *dynamic dispatch*. Utilizzando un *dynamic dispatch VI* sul *block diagram* di un altro *VI*, il dato effettivamente contenuto nel terminale di ingresso del *dynamic dispatch VI* andrà a determinare quale implementazione nella gerarchia di classe verrà invocata da LabVIEW. Dal momento che un filo può trasportare un oggetto della classe specificata oppure di una qualsiasi classe

derivata, il nodo esegue la particolare implementazione del metodo corrispondente al tipo di oggetto contenuto nel filo [26].

3.2.4 Sintassi *by-value* e *by-reference*

LabVIEW utilizza una sintassi basata quasi esclusivamente sul modello di programmazione *dataflow*. Ogni nodo compie delle operazioni sui propri valori di ingresso senza interferire con i valori contenuti in altri fili ad esso non collegati. Quando un filo viene sdoppiato, il valore in esso contenuto viene duplicato. Questo tipo di sintassi è definita *by-value* ed è consistente con il *dataflow*. I dati di tipo *refnum* rappresentano tuttavia un'eccezione: un *refnum* è infatti un riferimento ad una locazione di memoria condivisa, che può essere utilizzata da più nodi attraverso dei meccanismi di acquisizione e rilascio. Quando il filo di un *refnum* si sdoppia non c'è duplicazione del contenuto della locazione di memoria da esso indicata, ma viene semplicemente sdoppiato il riferimento. Questo tipo di sintassi è definita *by-reference*. In un linguaggio *dataflow* l'adozione di una sintassi *by-value* permette l'esecuzione di più *thread* senza la necessità di gestire eventuali interferenze tra operazioni che interessano una stessa area di memoria: i *thread* sono cioè indipendenti. L'adozione di una sintassi *by-reference* non implica questo tipo di indipendenza [24].

LabVIEW, diversamente da linguaggi come Java, C# e C++, utilizza *in toto* una sintassi *by-value* per i suoi oggetti: quando un filo viene sdoppiato, l'oggetto in esso contenuto viene cioè duplicato, salvo eventuali ottimizzazioni operate dal compilatore. L'adozione di una sintassi *by-value* è dovuta al fatto che l'*incapsulamento* viene realizzato attraverso il concetto di *classe*, basato a sua volta sul tipo di dato *cluster*, che viene manipolato sfruttando una sintassi *by-value* [24].

3.2.5 Costruttori e distruttori

In *LVOOP* i *costruttori* e i *distruttori* sono impliciti. Non è necessario invocare un costruttore per inizializzare un classe, dal momento che LabVIEW invoca un costruttore di default non appena una classe deve essere inizializzata. L'inizializzazione delle proprietà di una classe avviene per impostazione diretta dei valori tramite i controlli posizionati sul *front panel* oppure tramite costanti inserite nel *block diagram*. Quando una classe non è più utilizzata viene automaticamente deallocata dalla memoria da LabVIEW, con le stesse modalità utilizzate per *cluster* e array [27].

3.2.6 *Data Value Reference*

Diversamente da altri linguaggi di programmazione testuali, LabVIEW non ha il concetto di *spatial lifetime*[6] dei dati, ma utilizza il concetto di *temporal lifetime*, che prevede che un blocco di dati venga mantenuto in memoria finché utilizzato ed eliminato quando non più necessario. Tale blocco di dati, se copiato in un controllo

[6]Per *spatial lifetime* di un dato si intende uno *scope* di esistenza basato sulla posizione in cui lo stesso è stato definito all'interno del codice (tipicamente testuale).

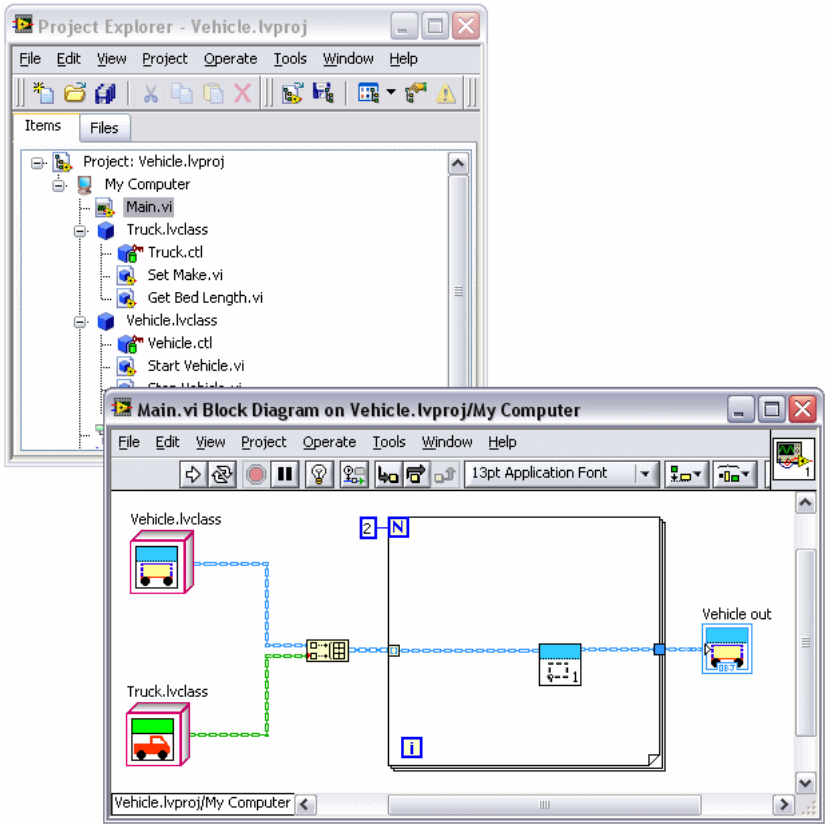


Figura 3.6: Esempio di utilizzo di un metodo *dynamic dispatch*. Sul *block diagram* di *Main* è visibile solamente l'implementazione appartenente alla classe *Vehicle* di *Set Make*. Nella prima iterazione del ciclo, LabVIEW esegue l'implementazione appartenente alla classe *Vehicle* di *Set Make*, essendo il contenuto del terminale di ingresso un oggetto di tipo *Vehicle*. Nella seconda iterazione del ciclo invece, LabVIEW esegue l'implementazione appartenente alla classe *Truck* (derivata da *Vehicle*) di *Set Make*, essendo il contenuto del terminale di ingresso un oggetto di tipo *Truck*. [26]. Fonte dell'illustrazione: http://zone.ni.com/images/reference/en-XX/help/371361F-01/loc_env_set_make.gif.

del *front panel*, non viene eliminato, nemmeno ad esecuzione terminata. Copie di dati presenti sui fili esistono fino alla prossima esecuzione del *VI* ad essi connesso. In un linguaggio *dataflow* i dati hanno un'esistenza molto lunga, che talvolta supera anche la fine dell'esecuzione del programma che li ha generati [24].

Alla luce della natura *dataflow* e *by-value* di LabVIEW, il problema dell'allocazione e del rilascio da parte di un oggetto delle risorse di sistema (memoria, canali di comunicazione, hardware) sembra non trovare una facile soluzione: non è infatti semplice decidere quando eseguire i costruttori impliciti descritti in 3.2.5 e, soprattutto, quando attivare i distruttori per il rilascio delle risorse precedentemente acquisite. Normalmente LabVIEW, essendo un linguaggio *dataflow*, non utilizza variabili. I fili e i controlli sul *front panel* non sono variabili. Nemmeno quei costrutti di LabVIEW che permettono di definire variabili locali o globali sono gestiti in memoria secondo il concetto di *spatial lifetime* in uso negli altri linguaggi. Le variabili sono quella parte del modello di programmazione di LabVIEW che si scontra con il *dataflow*. Senza disporre di variabili e di una modalità per definire il tempo di vita delle stesse, concetti come la costruzione e la distruzione *by-reference* di oggetti, in uso in altri linguaggi *object-oriented*, perdono ogni significato. Per questo motivo i costruttori e i distruttori descritti in 3.2.5 non sono stati pensati per allocare o rilasciare risorse di sistema [24].

La soluzione a questo problema è l'utilizzo di *DVR* (*Data Value Reference*). Essendo dati di tipo *refnum*, i *DVR* hanno uno *spatial lifetime* ben definito e possono essere utilizzati per creare riferimenti a qualsiasi tipo di dato. I *DVR* svolgono una funzione particolarmente utile se utilizzati per creare riferimenti ad oggetti di una classe. Esiste infatti la possibilità di limitare la capacità di creazione e distruzione di *DVR* di oggetti di una particolare classe solamente a *VI* membri della stessa classe. Qualsiasi altro *VI* non appartenente alla classe in questione non ne sarebbe autorizzato. Grazie a questa restrizione, una classe può inserire del codice di inizializzazione nei suoi *VI* per garantire che non vengano creati riferimenti ad oggetti della stessa classe senza prima aver compiuto le dovute inizializzazioni e tutte le allocazioni di risorse del caso. In modo speculare sarebbe garantita la distruzione di ogni riferimento da parte di uno specifico *VI* di classe, che implementerebbe tutte quelle operazioni di deallocazione e rilascio di risorse precedentemente acquisite. Il codice di deallocazione non verrebbe eseguito solamente in caso di arresto improvviso del programma, ma in tutti gli altri casi i *DVR* sarebbero in grado di realizzare lo *spatial lifetime* per gli oggetti LabVIEW, che sarebbero quindi utilizzati secondo una sintassi *by-reference*. [24].

3.2.7 *LVOOP* e C++

C++ è un linguaggio testuale imperativo, G è un linguaggio grafico *dataflow*. Ciascuno di questi due linguaggi implementa in modo diverso i principi e le funzionalità della *OOP*, sulla base dei rispettivi paradigmi di programmazione. Di seguito si elencano le principali differenze, riassunte in Tabella 3.1 [25]:

	C++	*LVOOP*
Classe progenitrice comune	✗	✓
Costruttori *by-reference*	✓	—
Distruttori *by-reference*	✓	—
Sintassi *by-value*	✓	✓
Sintassi *by-reference*	✓	—
Automatic data mutation	✗	✓
Template	✓	✗
Funzioni *pure virtual*	✓	✗
Ereditarietà multipla	✓	✗

Tabella 3.1: Differenze fra C++ e *LVOOP* [25].

- *LVOOP*, a differenza di C++, ha una classe progenitrice[7] per tutti gli oggetti.
- C++ utilizza costruttori *by-reference*, *LVOOP* non ne ha bisogno.
- C++ utilizza distruttori *by-reference*, *LVOOP* non ne ha bisogno.
- C++ supporta sia una sintassi *by-value* che una sintassi *by-reference* per il passaggio di oggetti come parametri. *LVOOP* supporta nativamente una sintassi *by-value* e realizza una sintassi *by-reference* attraverso l'uso di *DVR*.
- *LVOOP*, a differenza di C++, implementa un meccanismo di *automatic data mutation*, che permette all'utente di utilizzare classi definite in passato sebbene non più allineate con la definizione attuale delle stesse.
- C++, a differenza di *LVOOP*, supporta l'utilizzo di *template*[8].
- C++, a differenza di *LVOOP*, supporta la definizione di funzioni *pure virtual*[9].
- C++, a differenza di *LVOOP*, supporta l'ereditarietà multipla[10].

Questa panoramica sulle differenze più significative tra i due linguaggi completa la descrizione delle principali caratteristiche di *LVOOP*.

[7] In *LVOOP* tutte le classi derivano da una classe progenitrice denominata *LabVIEW Object* [26].
[8] I *template* sono uno dei costrutti del linguaggio C++ che permettono a classi e funzioni di operare con tipi di dati generici senza dover essere riscritte o modificate [28].
[9] Una funzione *pure virtual* è un metodo virtuale di una classe astratta che deve essere implementato da almeno una classe derivata non astratta [29].
[10] L'ereditarietà multipla è una caratteristica di alcuni linguaggi di programmazione orientata agli oggetti che prevede la possibilità per una classe di ereditare proprietà e metodi da più classi progenitrici [30].

3.3 Utilizzo di codice esterno

LabVIEW mette a disposizione un particolare *subVI* denominato *Call Library Function Node* (*CLFN*) per poter richiamare direttamente funzioni di libreria esterna (*Windows DLL*[11], *Mac OS X Framework*, *Linux Shared Library*) all'interno del codice G [31]. *CLFN* è dotato di coppie di terminali di ingresso e uscita. Ciascuna coppia di terminali corrisponde ad un parametro della lista degli argomenti della funzione da invocare, specificati attraverso un'apposita finestra di dialogo. Il passaggio di un parametro si realizza collegando il filo corrispondente al terminale sinistro di una coppia di terminali. Per leggere un valore di uscita generato dall'esecuzione della funzione è necessario collegare un filo al terminale destro della corrispondente coppia di terminali. *CLFN* supporta un gran numero di tipi di dati e convenzioni di chiamata [32]. *CLFN* è infatti in grado di utilizzare sia una convenzione di chiamata *C* che una convenzione di chiamata *standard* (*PASCAL*, *WINAPI*). Le *DLL* utilizzano solitamente una convenzione di chiamata *standard*, che rappresenta la convenzione di chiamata di default per *CLFN* [33]. Il nome della funzione con il numero e il tipo di argomenti richiesti determinano il prototipo della funzione [33]. Per esempio, la funzione con prototipo

```
void MyFunction(int a, double* b, char* string, unsigned long arraysize, short* dataarray);
```

della libreria `mydll.dll` con convenzione di chiamata *standard* verrà richiamata attraverso il *CLFN* di Figura 3.7, configurato tramite la finestra di dialogo mostrata in Figura 3.8 [33]. Per una trattazione più approfondita si faccia riferimento a [34].

[11] *Dynamic Link Library.*

Figura 3.7: *CLFN* della funzione *MyFunction()*. Fonte dell'illustrazione: http://www.ni.com/cms/images/devzone/tut/a/b80e74b4409.gif.

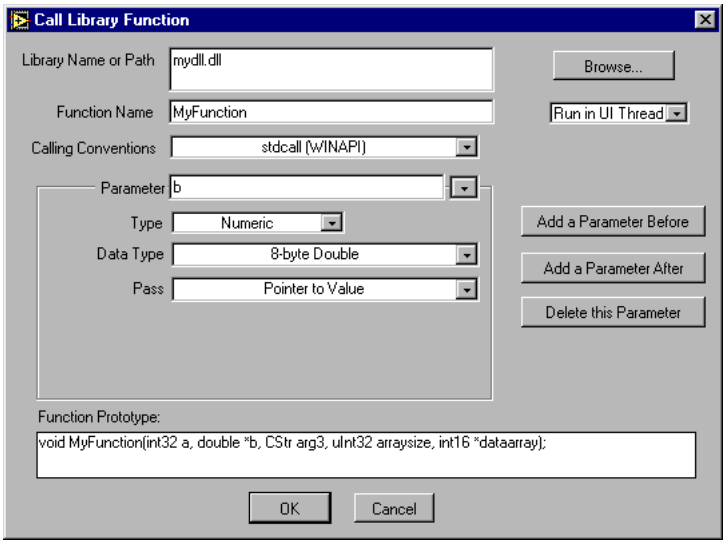


Figura 3.8: Finestra di dialogo relativa alla configurazione del *CLFN* della funzione *MyFunction()*. Fonte dell'illustrazione http://www.ni.com/cms/images/devzone/tut/a/b80e74b4406.gif.

4 Soluzione proposta

Questo capitolo descrive l'attività di progettazione e implementazione dell'interfaccia LabVIEW di MDSplus e rappresenta il nucleo del lavoro di tesi.

4.1 Introduzione

L'approfondimento della conoscenza di MDSplus, in particolare della sua nuova interfaccia *object-oriented*, e delle possibilità offerte da LabVIEW, sondate anche attraverso la realizzazione di semplici applicazioni didattiche, lascia il posto alla fase di progettazione, sviluppo e test dell'interfaccia LabVIEW del sistema.

L'obiettivo fondamentale del lavoro è l'individuazione di un'architettura che renda possibile la costruzione di applicazioni di acquisizione dati in grado di sfruttare le ultime e più utilizzate funzionalità della nuova interfaccia *object-oriented* di MDSplus. L'implementazione di tutte le rimanenti funzionalità seguirà una volta appurato il funzionamento e la solidità della soluzione proposta. Si individua quindi nella realizzazione di una semplice applicazione LabVIEW di acquisizione continua di dati, descritta in dettaglio nel capitolo 5, il banco di prova per la verifica del raggiungimento di questo obiettivo.

4.2 Vincoli progettuali

Le linee guida dell'attività di progettazione prevedono che si prenda come riferimento l'ultima versione C++ di MDSplus per Windows e si cerchi di operare una sorta di *mapping* delle funzionalità di quest'ultima nella nuova interfaccia LabVIEW del sistema, cercando di riutilizzare, dove possibile, il codice C++ già testato e funzionante. Queste linee guida si inseriscono nell'insieme dei diversi vincoli progettuali considerati nel corso dell'attività di progettazione, con l'obiettivo di ottenere un'interfaccia robusta, multipiattaforma ed eseguibile in *real-time*. Si elencano e si approfondiscono di seguito le caratteristiche fondamentali da implementare nella nuova interfaccia:

- *Supporto delle funzionalità di* MDSplus.
- *Supporto multipiattaforma.*
- *Supporto dell'esecuzione real-time.*
- *Gestione della concorrenza.*
- *Serializzabilità di operazioni composte.*

4.2.1 Supporto delle funzionalità di MDSplus

Uno dei requisiti fondamentali della nuova interfaccia è la capacità di supportare tutte le operazioni e i tipi di dati in uso nell'interfaccia C++ di MDSplus, descritta in 2.3, in particolare le nuove funzionalità di acquisizione continua dei dati, descritte in 2.3.4. I tipi di dati e i *VI* della nuova interfaccia devono essere compatibili con i tipi di dati e i metodi definiti dalla corrispettiva interfaccia C++, per permettere allo sviluppatore LabVIEW di utilizzare i dati acquisiti con l'hardware National Instruments nello specifico formato di LabVIEW, senza doversi preoccupare di operare le dovute conversioni. In più, le modalità di utilizzo dei dati e dei *VI* che li elaborano, nonché delle operazioni di lettura, scrittura e modifica su *node* e *tree* devono avere la stessa logica, in modo tale che uno sviluppatore abituato a utilizzare la versione C++ di MDSplus possa ritrovare lo stesso principio di funzionamento nella versione LabVIEW del sistema, ad eccezione di comprensibili differenze dovute all'utilizzo di un linguaggio di programmazione grafica e *dataflow*.

4.2.2 Supporto multipiattaforma

La versione C++ di MDSplus può essere facilmente compilata per l'esecuzione su piattaforme e architetture distinte. La diversa rappresentazione dei dati, dovuta alla particolare architettura o piattaforma considerata, viene gestita attraverso delle direttive al compilatore presenti all'interno del codice. Le principali piattaforme attualmente supportate sono Linux, Unix, Windows e Mac OS X. Per una lista completa di tutte le piattaforme su cui è stato installato MDSplus nel corso del tempo si faccia riferimento a 2.4. La nuova interfaccia LabVIEW del sistema deve quindi essere in grado di gestire questa diversità nella rappresentazione dei dati, in modo da poter essere utilizzata nell'ambito delle principali piattaforme e architetture supportate da LabVIEW, ovvero Windows, Linux e Mac OS X, a 32 bit e 64 bit.

4.2.3 Supporto dell'esecuzione *real-time*

Un requisito importante, anche se non indispensabile, per la nuova interfaccia LabVIEW di MDSplus è la possibilità di essere utilizzata su hardware National Instruments gestito da sistemi operativi *real-time*, per esempio CompactRIO gestito da VxWorks[1]. L'estensione del supporto nativo di MDSplus all'hardware *embedded* è infatti uno degli obiettivi aggiuntivi di National Instruments per il progetto.

4.2.4 Gestione della concorrenza

I metodi dell'interfaccia C++ di MDSplus, pur essendo in grado di gestire correttamente accessi concorrenti in lettura e scrittura su specifici *node* e di impedire operazioni concorrenti di modifica della struttura di un dato *tree*, sono stati pensati per essere utilizzati da applicazioni *batch* costituite da una sequenza ordinata di

[1] VxWorks è un sistema operativo *real-time* proprietario sviluppato da Wind River Systems of Alameda, California, USA [36].

istruzioni e tramutate *runtime* in processi *single-threaded*. Nel dominio della programmazione sequenziale risulta molto più semplice gestire operazioni di allocazione, deallocazione e accesso alla memoria, nonché individuare eventuali errori logici nelle istruzioni, come l'utilizzo di un oggetto precedentemente deallocato. Non sono quindi necessari meccanismi di *lock* espliciti sui dati.

Il modello di programmazione *dataflow* permette a LabVIEW di supportare facilmente la creazione di applicazioni concorrenti. Tramite lo sdoppiamento di un filo contenente un oggetto viene infatti eseguita una copia di quell'oggetto, da quel momento in poi del tutto indipendente dal destino dell'oggetto di partenza. L'esecuzione di *VI* membri da parte di ciascuno dei due oggetti non va ad influenzare reciprocamente le due copie, ma comporta la creazione di due *thread* distinti e indipendenti, ciascuno operante sulla propria copia dell'oggetto: l'utilizzo di meccanismi di *lock* su dati manipolati *by-value* non sono quindi necessari. Tali meccanismi diventano tuttavia indispensabili qualora si acceda in scrittura a risorse condivise, rappresentate da *refnum* o *DVR* di oggetti: un *tree* di MDSplus, essendo una risorsa condivisa, dovrà perciò essere rappresentato *by-reference* e gestito di conseguenza. La nuova interfaccia LabVIEW del sistema deve quindi essere in grado di gestire correttamente l'accesso concorrente a dati e risorse che non possono essere utilizzati *by-value* secondo il modello *dataflow*.

4.2.5 Atomicità di operazioni composte

MDSplus permette di salvare i dati di uno specifico processo di acquisizione in un apposito *shot* di un *tree*. L'accesso concorrente in scrittura ad uno stesso *node* è un'operazione priva di significato essendo un *tree* una base di dati sperimentali progettata per salvare acquisizioni diverse in *node* appartenenti a *shot* progressivi distinti. Un meccanismo che permetta di implementare l'atomicità di sequenze di operazioni *read/modify/write* distinte per evitare di produrre inconsistenze nei dati è tuttavia una caratteristica desiderabile per la nuova interfaccia LabVIEW di MDSplus.

4.3 Architettura proposta

L'idea fondamentale alla base dell'architettura per la nuova interfaccia LabVIEW di MDSplus è la realizzazione di un *mapping* tra le classi della versione C++ del sistema e le corrispondenti classi *LVOOP* appositamente create in LabVIEW. In questa sezione si darà una descrizione ad alto livello delle caratteristiche del *mapping* per poi passare ad una descrizione a basso livello del collegamento tra classi C++ e classi *LVOOP*. Si illustreranno inoltre le scelte architetturali effettuate per soddisfare i vincoli progettuali illustrati in 4.2.

4.3.1 *Mapping* tra classi

Per ogni classe della versione C++ di MDSplus viene creata una classe omonima appartenente all'interfaccia LabVIEW del sistema. Considerando le differenze esistenti

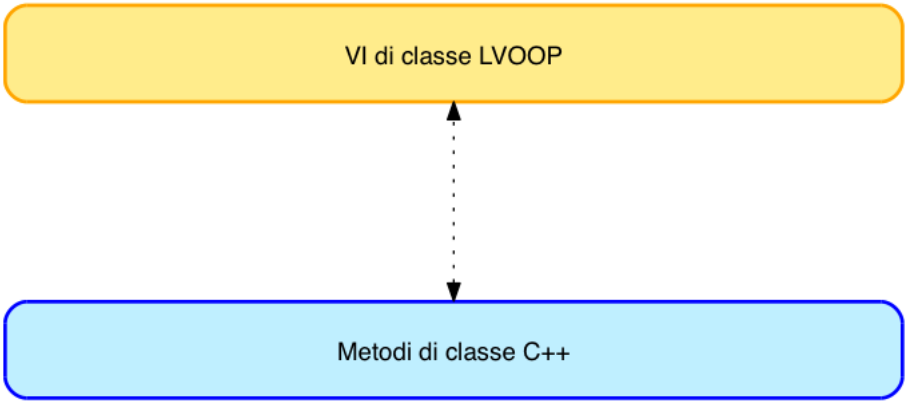


Figura 4.1: Ipotesi di *mapping* tra classi C++ e classi *LVOOP*.

tra *LVOOP* e C++ nella declinazione del paradigma *OOP*, in particolare l'obbligo di dichiarare privati tutti i dati di una classe *LVOOP*, non viene operata una ridefinizione pedissequa di tutte le proprietà delle classi C++ nelle corrispondenti classi *LVOOP*, ma viene definita un'unica proprietà, denominata *Object Pointer*, che contiene l'indirizzo di memoria di un oggetto della corrispondente classe C++ già istanziato dal relativo costruttore. Gli oggetti *LVOOP* rappresentano quindi un contenitore per i corrispondenti oggetti C++ e hanno il compito di ricreare in *LVOOP* la stessa gerarchia di classi definita in C++. In questo modo si ripropone la stessa logica di utilizzo delle funzionalità del sistema in *LVOOP*, soddisfando buona parte dei vincoli progettuali descritti in 4.2.1. L'utilizzo di *LVOOP* è inoltre compatibile con l'esecuzione su piattaforme *real-time*: il vincolo progettuale descritto in 4.2.3 risulta quindi soddisfatto. In Figura 4.1 viene illustrata questa ipotesi architetturale.

L'algoritmo 4.1 presenta una parziale definizione C++ della classe *Tree*: si possono notare le tre proprietà *protected* `name`, `shot` e `ctx`. In Figura 4.2 si può invece osservare che l'unica proprietà della corrispondente classe *Tree* definita in *LVOOP* è `Object Pointer`.

Tutti i metodi pubblici delle classi C++, compresi costruttori, distruttori e metodi di classe, vengono ridefiniti con lo stesso nome e la stessa firma in *LVOOP*. In Figura 4.3 sono raffigurati i *VI* immagine di alcuni dei metodi della classe *Tree*. Tutti i *VI* immagine di metodi pubblici diversi da costruttori e distruttori sono *dynamic dispatch*: in questo modo eventuali metodi di classi derivate che operano un *override* dei metodi omonimi della classe progenitrice verranno correttamente richiamati sulla base dell'oggetto presente nel filo in ingresso.

Per esemplificare il meccanismo di funzionamento del *dynamic dispatching* si riporta in Figura 4.4(a) l'esecuzione *step-by-step* dei *subVI* contenuti nel *block diagram* di un semplice *VI* che compie le seguenti operazioni:

Algoritmo 4.1 Definizione parziale della classe *Tree*

```
class EXPORT Tree {
    ...
  protected:
    ...
    char *name;
    int shot;
    void *ctx;
  public:
    ...
    Tree(char *name, int shot, char *mode);
    ~Tree();
    ...
    TreeNode *getNode(char *path);
    ...
};
```

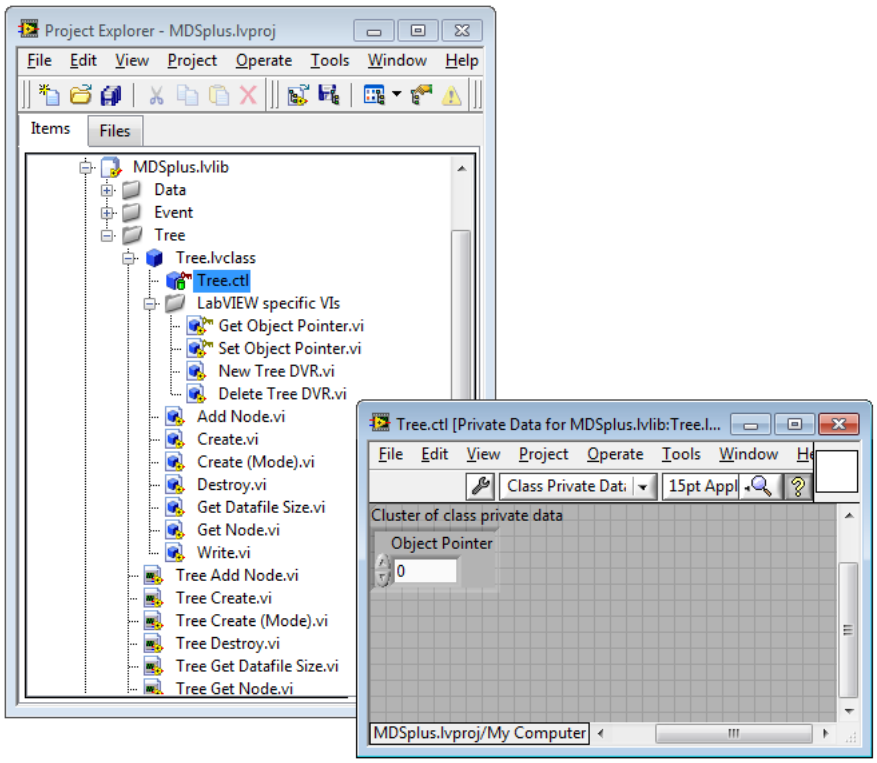


Figura 4.2: *Class library file* della classe *Tree* con relativo *cluster* contenente i controlli per i dati privati della classe.

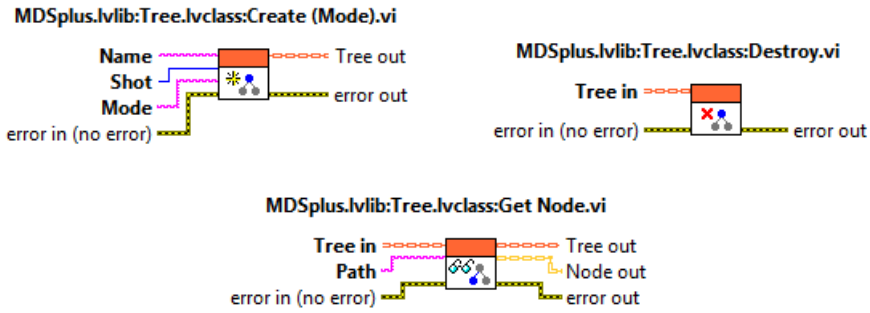


Figura 4.3: Illustrazione dei *connector pane* di alcuni dei *VI* membri della classe *Tree*: `Tree.lvclass:Create (Mode).vi` è l'immagine del costruttore `Tree::Tree(char *name, int shot, char *mode)`; `Tree.lvclass:Destroy.vi` è l'immagine del distruttore `Tree::~Tree()`; `Tree.lvclass:Get Node.vi` è l'immagine del metodo pubblico `TreeNode *Tree::getNode(char *path)`.

1. Apertura del *tree* `my_tree` in scrittura tramite `Tree.lvclass:Create (Mode).vi`[2].
2. Accesso al *node* `NUM0` del *tree* (aperto al punto 1) tramite `Tree.lvclass:Get Node.vi`[3], che ritorna in uscita un oggetto della classe *TreeNode*.
3. Creazione di un nuovo oggetto *Float32* tramite `Float32.lvclass:Create.vi`[4].
4. Scrittura del valore dell'oggetto *Float32* (creato al punto 3) nell'oggetto *TreeNode* (instanziato al punto 2) tramite `TreeNode.lvclass:Put Data.vi`[5].
5. Lettura del contenuto dell'oggetto *TreeNode* (instanziato al punto 2 e modificato al punto 4) tramite `TreeNode.lvclass:Get Data.vi`[6], che ritorna in uscita un oggetto *Data*.
6. Lettura del contenuto dell'oggetto *Data* (ritornato al punto 5) e successiva trasformazione in un valore in virgola mobile a precisione singola tramite `Data.lvclass:Get Float.vi`[7].

---

2 *Connector pane* di `Tree.lvclass:Create (Mode).vi`.

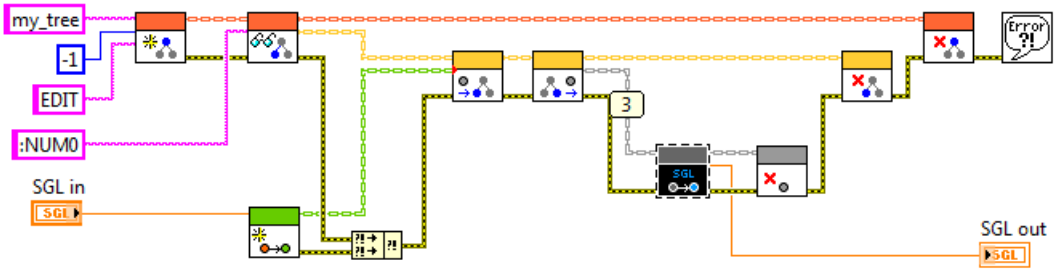
3 *Connector pane* di `Tree.lvclass:Get Node.vi`.

4 *Connector pane* di `Float32.lvclass:Create.vi`.

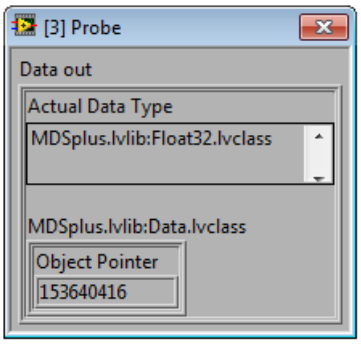
5 *Connector pane* di `TreeNode.lvclass:Put Data.vi`.

6 *Connector pane* di `TreeNode.lvclass:Get Data.vi`.

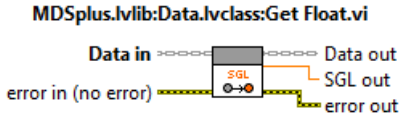
7 *Connector pane* di `Data.lvclass:Get Float.vi`.



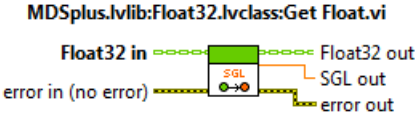
(a)



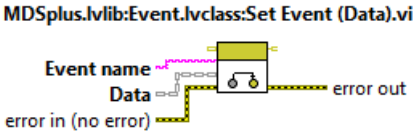
(b)



(c)



(d)



(e)

Figura 4.4: (a) *Block diagram* in esecuzione di un *VI* di esempio per mostrare il funzionamento del *dynamic dispatching*; (b) Finestra di *debug* che permette di conoscere il contenuto di un filo; (c) *Connector pane* di `Data.lvclass:Get Float.vi`; (d) *Connector pane* di `Float32.lvclass:Get Float.vi`; (e) *Connector pane* del *VI* di classe `Event.lvclass:Set Event (Data).vi`.

7. Distruzione dell'oggetto *Data* (ritornato al punto 5 e utilizzato al punto 6) tramite `Data.lvclass:Destroy.vi`[8].

8. Distruzione dell'oggetto *TreeNode* (ritornato al punto 2) tramite `TreeNode.lvclass:Destroy.vi`[9].

9. Distruzione dell'oggetto *Tree* (ritornato al punto 1) tramite `Tree.lvclass:Destroy.vi`[10].

Come si può vedere in Figura 4.4(a), il prossimo *subVI* da eseguire (evidenziato in nero) è `Data.lvclass:Get Float.vi`, rappresentato in Figura 4.4(c). La Figura 4.4(b) mostra il contenuto del filo grigio di ingresso di `Data.lvclass:Get Float.vi`: si può notare che il filo, pur essendo definito in modo tale da contenere un oggetto *Data*, in realtà contiene un oggetto *Float32*, situazione lecita in quanto la classe *Float32* annovera tra le proprie classi progenitrici la classe *Data*. Per il *dynamic dispatching* non verrà quindi eseguito `Data.lvclass:Get Float.vi`, ma `Float32.lvclass:Get Float.vi`, rappresentato in Figura 4.4(d). Si può notare che entrambi i *VI* hanno *connector pane* configurati allo stesso modo, ad eccezione dei tipi di oggetto in ingresso.

I metodi di classe sono mappati in *static dispatch VI* e non hanno bisogno di essere richiamati da un particolare oggetto presente nel terminale di ingresso, il cui collegamento risulta pertanto opzionale. In Figura 4.4(e) è rappresentato il *connector pane* di un *VI* di classe appartenente ad *Event*[11]: si può notare che il collegamento dell'oggetto in ingresso non è indicato come in Figura 4.4(c) e 4.4(d), differenza che lo identifica come opzionale.

Particolare attenzione viene prestata alla definizione in *LVOOP* dei *VI* immagine dei costruttori e dei distruttori C++. I costruttori vengono mappati in *static dispatch VI*, in quanto ogni classe deve avere il proprio specifico costruttore. I distruttori vengono invece mappati in *VI* con il terminale di ingresso che è *dynamic dispatch*, in quanto è possibile che venga invocato un *VI* distruttore di una classe progenitrice su di un oggetto che in realtà appartiene ad una classe derivata: in questo caso, grazie al *dynamic dispatching*, viene eseguito il distruttore della classe derivata invece che il distruttore della classe progenitrice. Questa situazione si verifica nell'esempio di Figura 4.4(a) al punto 7: essendo il contenuto del filo di ingresso di `Data.lvclass:Destroy.vi` di tipo *Float32*, il *VI* distruttore effettivamente eseguito è `Float32.lvclass:Destroy.vi`[12]. Nei *VI* distruttori il terminale di uscita per l'oggetto è assente, in quanto l'oggetto viene distrutto all'interno del *VI*.

---

[8]  *Connector pane* di `Data.lvclass:Destroy.vi`.

[9]  *Connector pane* di `TreeNode.lvclass:Destroy.vi`.

[10]  *Connector pane* di `Tree.lvclass:Destroy.vi`.

[11] *Event* è una classe di MDSplus.

[12]  *Connector pane* di `Float32.lvclass:Destroy.vi`.

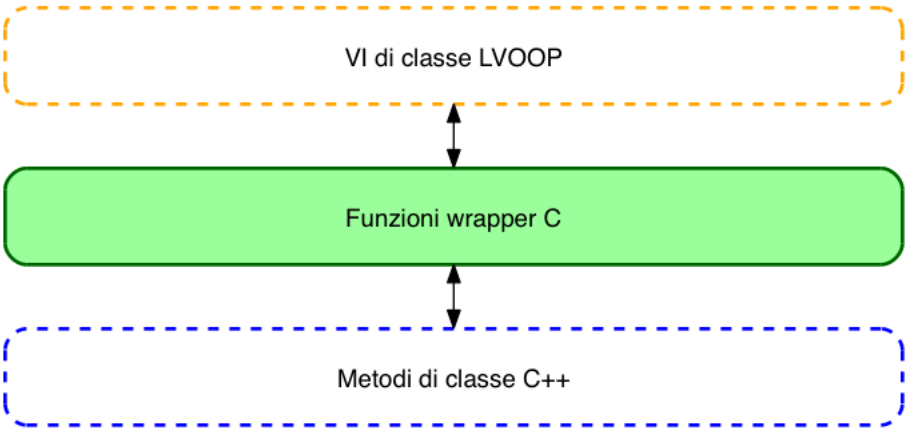


Figura 4.5: *Mapping* tra classi C++ e classi *LVOOP* realizzato attraverso l'utilizzo di funzioni *wrapper* e *CLFN*.

4.3.2 Collegamento tramite *CLFN*

Il collegamento tra i metodi C++ del sistema e i corrispettivi *VI* della nuova interfaccia si basa sull'utilizzo di *CLFN*, *subVI* che mette in collegamento una libreria esterna contenente tutti i metodi del sistema con il *block diagram* dei *VI* immagine degli stessi metodi. Come descritto in 3.3, *CLFN* si interfaccia con *Windows DLL*, *Mac OS X Framework* e *Linux Shared Library*: il supporto per librerie esterne eseguibili su piattaforme diverse soddisfa buona parte dei vincoli progettuali descritti in 4.2.2.

La libreria esterna è composta da funzioni *wrapper* C che incapsulano le chiamate ai metodi degli oggetti C++ del sistema, passando in ingresso il puntatore allo specifico oggetto sul quale un metodo va ad operare e ritornando in uscita i risultati. In caso di funzioni che incapsulano costruttori e distruttori, il puntatore all'oggetto verrà rispettivamente ritornato, in seguito all'avvenuta creazione, e annullato, in seguito all'avvenuta distruzione. La definizione di queste funzioni *wrapper* risulta necessaria in quanto non è possibile interfacciare direttamente le classi e i metodi C++ di MDSplus con *CLFN*, che accetta solamente librerie composte da funzioni C. In Figura 4.5 è illustrata l'implementazione tramite funzioni *wrapper* e *CLFN* dell'ipotesi architetturale proposta in Figura 4.1.

In Algoritmo 4.2 è riportata la struttura della funzione *wrapper* del costruttore della classe *Tree*: si può notare che viene creato un nuovo oggetto *Tree* tramite il costruttore `Tree::Tree(char *name, int shot, char *mode)`, il puntatore in memoria al nuovo oggetto viene quindi assegnato a `treePtrOut` e successivamente copiato in `lvTreePtrOut`, parametro che si interfaccia con *CLFN*. In Algoritmo 4.3 è riportata la struttura della funzione *wrapper* del metodo *getNode()* della classe *Tree*: si può notare che il puntatore all'oggetto *Tree*, che ha invocato il metodo, viene passato per valore alla funzione attraverso `lvTreePtr`, mentre il puntatore dell'oggetto *TreeNode*, ritornato dall'invocazione di `TreeNode *Tree::getNode(char *path)`, viene

Algoritmo 4.2 Funzione *wrapper* del costruttore della classe *Tree*

```
extern "C" __declspec(dllexport) void mdsplus_tree_constructor_mode(void **lvTreePtrOut, const
char *nameIn, int shotIn, const char *modeIn, ErrorCluster *error)
{
  Tree *treePtrOut = NULL;
  MgErr errorCode = noErr; char *errorSource = __FUNCTION__; char *errorMessage = "";
  try
  {
    treePtrOut = new Tree(const_cast<char *>(nameIn), shotIn, const_cast<char *>(modeIn));
    *lvTreePtrOut = reinterpret_cast<void *>(treePtrOut);
  }
  catch (MdsException *mdsE)
  {
    delete treePtrOut;
    errorCode = bogusError; errorMessage = const_cast<char *>(mdsE->what());
  }
  catch (exception *e)
  {
    delete treePtrOut;
    errorCode = bogusError; errorMessage = const_cast<char *>(e->what());
  }
  fillErrorCluster(errorCode, errorSource, errorMessage, error);
}
```

Algoritmo 4.3 Funzione *wrapper* del metodo *getNode()* della classe *Tree*

```
extern "C" __declspec(dllexport) void mdsplus_tree_getNode(const void *lvTreePtr, void
**lvTreeNodePtrOut, const char *pathIn, ErrorCluster *error)
{
  Tree *treePtr = NULL;
  TreeNode *treeNodePtrOut = NULL;
  MgErr errorCode = noErr; char *errorSource = __FUNCTION__; char *errorMessage = "";
  try
  {
    treePtr = reinterpret_cast<Tree *>(const_cast<void *>(lvTreePtr));
    treeNodePtrOut = treePtr->getNode(const_cast<char *>(pathIn));
    *lvTreeNodePtrOut = reinterpret_cast<void *>(treeNodePtrOut);
  }
  catch (MdsException *mdsE)
  {
    delete treeNodePtrOut;
    errorCode = bogusError; errorMessage = const_cast<char *>(mdsE->what());
  }
  catch (exception *e)
  {
    delete treeNodePtrOut;
    errorCode = bogusError; errorMessage = const_cast<char *>(e->what());
  }
  fillErrorCluster(errorCode, errorSource, errorMessage, error);
}
```

Algoritmo 4.4 Funzione *wrapper* del distruttore della classe *Tree*

```
extern "C" __declspec(dllexport) void mdsplus_tree_destructor(void **lvTreePtr)
{
  Tree *treePtr = reinterpret_cast<Tree *>(*lvTreePtr);
  delete treePtr;
  *lvTreePtr = NULL;
}
```

assegnato a `treeNodePtrOut` e successivamente copiato in `lvTreeNodePtrOut`, parametro che si interfaccia con *CLFN*, con le stesse modalità con cui l'oggetto *Tree* viene ritornato a *CLFN* dalla funzione *wrapper* descritta in Algoritmo 4.2. In Algoritmo 4.4 è riportata la struttura della funzione *wrapper* del distruttore della classe *Tree*: l'oggetto *Tree* viene passato per riferimento alla funzione, che lo elimina dalla memoria invocando il distruttore e ritornando a *CLFN* un valore `NULL`. Per ogni metodo di MDSplus esiste una corrispondente funzione *wrapper*, la cui struttura ricalca quella delle funzioni appena descritte, salvo alcune differenze introdotte dal numero e dal tipo di parametri di ingresso, ciascuno dei quali si interfaccia con i tipi di dati di LabVIEW in modo specifico.

Ogni funzione *wrapper* viene assegnata ad uno specifico *CLFN*, inserito a sua volta nel *block diagram* dell'immagine *LVOOP* del metodo incapsulato nella funzione stessa. In Figura 4.6 e in Figura 4.7 sono mostrate due pagine della finestra di dialogo di *CLFN*, che permette di specificare la funzione *wrapper* da collegare a *CLFN* e di impostare i terminali di ingresso e uscita di *CLFN* con i corrispondenti parametri della funzione *wrapper* selezionata. In Figura 4.8 è riportato il *block diagram* di `Tree.lvclass:Create (Mode).vi`, costruttore della classe *Tree*: si può vedere come i vari parametri di ingresso siano collegati a *CLFN* e come il puntatore all'oggetto *Tree*, appena creato dalla funzione *wrapper* descritta in Algoritmo 4.2 ed eseguita da *CLFN*, venga ritornato in uscita e vada a definire il nuovo oggetto *Tree*. In Figura 4.9 è riportato il *block diagram* di `Tree.lvclass:Get Node.vi`, *VI* della classe *Tree*: si può vedere come i vari parametri di ingresso siano collegati a *CLFN* e come il puntatore all'oggetto *TreeNode*, appena creato dalla funzione *wrapper* descritta in Algoritmo 4.3 ed eseguita da *CLFN*, venga ritornato in uscita e vada a definire un nuovo oggetto *TreeNode*. In questo caso viene passato in uscita anche l'oggetto *Tree* che ha eseguito il *VI*. In Figura 4.10 è riportato il *block diagram* di `Tree.lvclass:Destroy.vi`, distruttore della classe *Tree*: si può vedere come i vari parametri di ingresso siano collegati a *CLFN* e come il puntatore all'oggetto *Tree* venga passato alla funzione *wrapper* descritta in Algoritmo 4.2 ed eseguita da *CLFN*, che distrugge l'oggetto e ritorna un puntatore nullo. Per ogni funzione *wrapper* esiste un corrispondente *VI* membro, il cui *block diagram* ricalca quello dei *VI* membri della classe *Tree* appena descritti, salvo alcune differenze introdotte dalle caratteristiche peculiari di ciascuna funzione *wrapper*.

Il puntatore che si interfaccia in ingresso o in uscita con *CLFN* è rappresentato

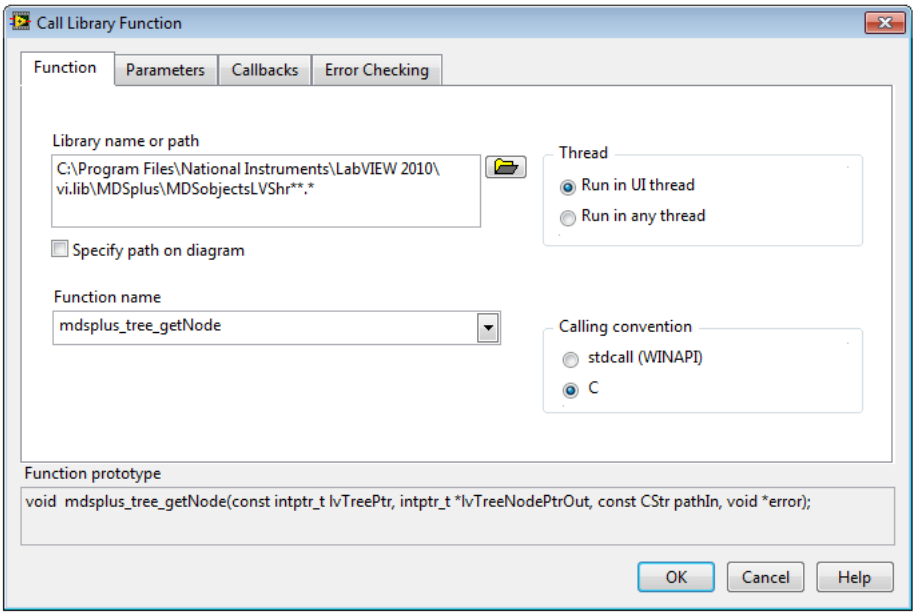


Figura 4.6: Finestra di dialogo di *CLFN* di `Tree.lvclass:Get Node.vi`: in questa scheda si possono interfacciare le funzioni *wrapper* della libreria con *CLFN*.

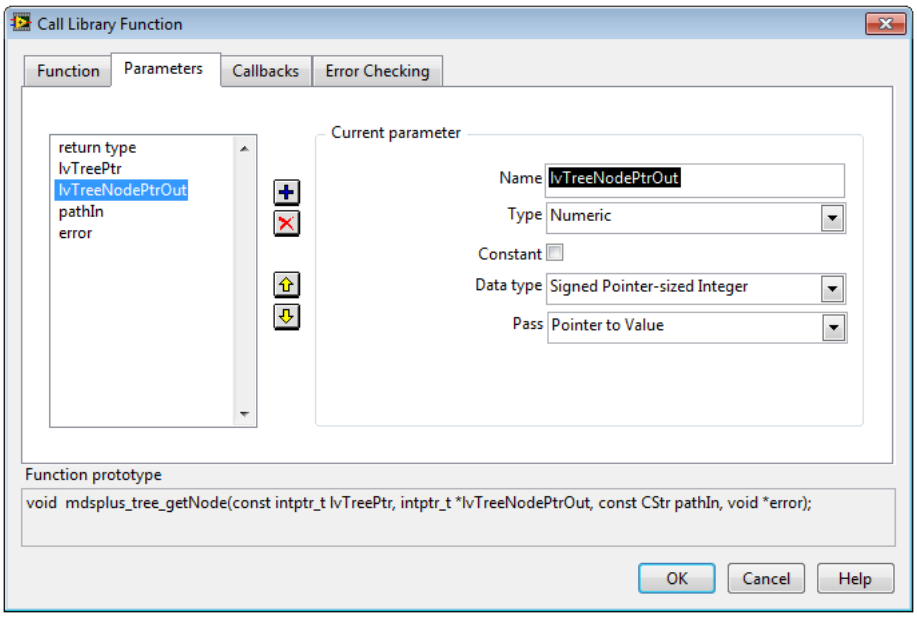


Figura 4.7: Finestra di dialogo di *CLFN* di `Tree.lvclass:Get Node.vi`: in questa scheda si possono collegare i terminali di *CLFN* con i parametri di ingresso e uscita della funzione *wrapper* selezionata.

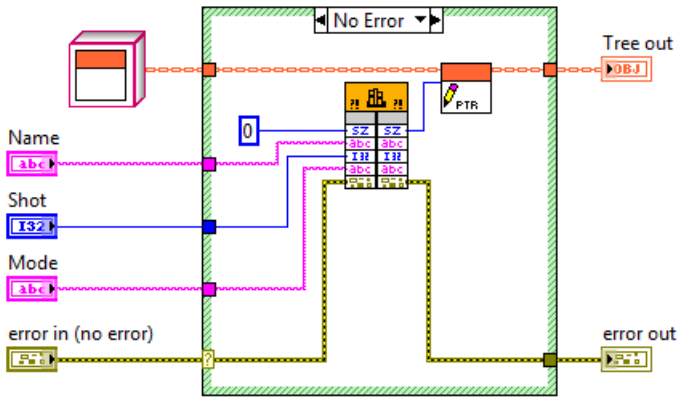


Figura 4.8: *Block diagram* di `Tree.lvclass:Create (Mode).vi`.

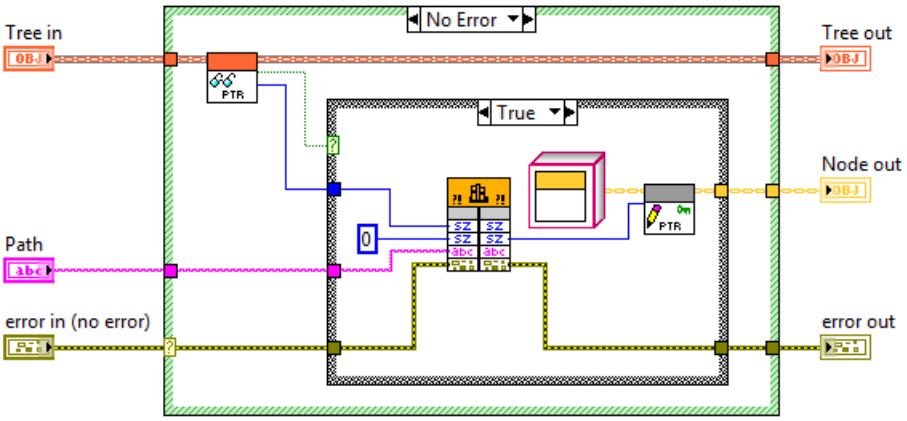


Figura 4.9: *Block diagram* di `Tree.lvclass:Get Node.vi`.

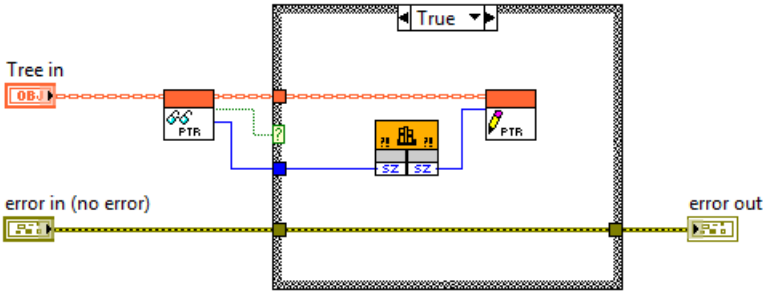
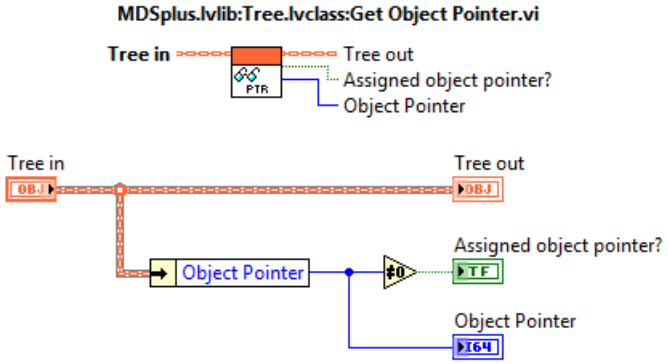
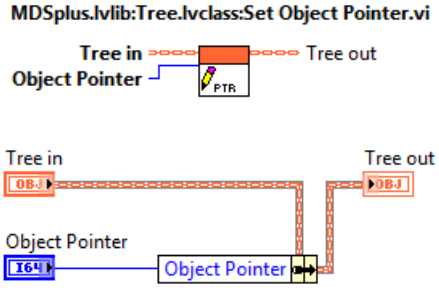


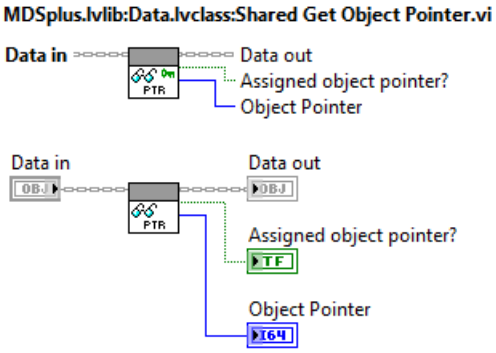
Figura 4.10: *Block diagram* di `Tree.lvclass:Destroy.vi`.



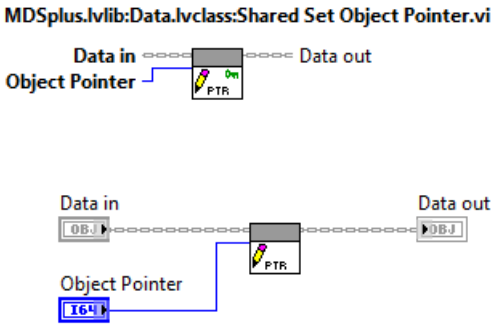
(a)



(b)



(c)



(d)

Figura 4.11: (a) *Connector* pane e *block diagram* di `Tree.lvclass:Get Object Pointer.vi`; (b) *Connector* pane e *block diagram* di `Tree.lvclass:Set Object Pointer.vi`; (c) *Connector* pane e *block diagram* di `Data.lvclass:Shared Get Object Pointer.vi`; (d) *Connector* pane e *block diagram* di `Data.lvclass:Shared Set Object Pointer.vi`.

dall'unica proprietà di ogni oggetto dell'interfaccia *LVOOP* di MDSplus, ovvero *Object Pointer*. *Object Pointer* è rappresentato come valore intero con segno a 64 bit: questa definizione permette di eseguire correttamente *CLFN* sia su architetture a 32 bit che su architetture a 64 bit, soddisfando i vincoli progettuali descritti in 4.2.2. Come si può vedere in Figura 4.7, l'argomento `lvTreeNodePtrOut`, che rappresenta il valore del puntatore del nuovo oggetto *TreeNode* ritornato dalla funzione *wrapper* corrispondente al metodo *getNode()* di *Tree*, è definito come `Signed Pointer-sized Integer`, ovvero come un puntatore a 32 bit o 64 bit, a seconda dell'architettura del sistema su cui è eseguito *CLFN*. Considerando che un'architettura a 32 bit rappresenta i puntatori come interi a 32 bit e che un'architettura a 64 bit rappresenta i puntatori come interi a 64 bit, la definizione di *Object Pointer* come intero con segno a 64 bit rende possibile il supporto di entrambe le architetture, senza bisogno di differenziare il codice sorgente: nel primo caso, infatti, *CLFN* utilizza solamente i 32 bit meno significativi di *Object Pointer*, mentre nel secondo caso, in modo speculare, *CLFN* utilizza tutti i 64 bit. Essendo per definizione privato, *Object Pointer* può essere letto e scritto solamente per mezzo di due appositi *VI* di accesso, dichiarati *protected* e quindi ereditati da eventuali classi derivate dalla classe che li definisce. In Figura 4.11(a) si possono osservare il *connector pane* e il *block diagram* di `Tree.lvclass:Get Object Pointer.vi`, *VI* di accesso in lettura per la proprietà *Object Pointer* della classe *Tree*. In Figura 4.11(b) si possono osservare il *connector pane* e il *block diagram* di `Tree.lvclass:Set Object Pointer.vi`, *VI* di accesso in scrittura per la proprietà *Object Pointer* della classe *Tree*. Essendo definiti *protected*, i *VI* di accesso ad *Object Pointer* possono essere utilizzati solamente nel *block diagram* di *VI* membri della classe di appartenenza o di classi derivate. Ci sono però dei casi in cui un *VI* di una classe ritorna in uscita un nuovo oggetto di un'altra classe, magari non derivata dalla prima, ed è quindi necessario poter accedere alla proprietà *Object Pointer* del nuovo oggetto per permetterne la corretta creazione. Vengono quindi ridefiniti i due *VI* di accesso in modo tale che siano in *community scope* e possano così essere dichiarati *friend* della classe che li dovrà utilizzare, rendendone possibile la corretta invocazione dal *block diagram* della stessa. In Figura 4.11(c) si possono osservare il *connector pane* e il *block diagram* di `Data.lvclass:Shared Get Object Pointer.vi`, *VI* in *community scope* di accesso in lettura per la proprietà *Object Pointer* della classe *Data* o di una classe derivata da *Data*. In Figura 4.11(d) si possono osservare il *connector pane* e il *block diagram* di `Data.lvclass:Shared Set Object Pointer.vi`, *VI* in *community scope* di accesso in scrittura per la proprietà *Object Pointer* della classe *Data* o di una classe derivata da *Data*. Il caso appena discusso si può ritrovare nel *block diagram* di `Tree.lvclass:Get Node.vi` rappresentato in Figura 4.9: per la costruzione dell'oggetto *TreeNode* viene infatti utilizzato `Data.lvclass:Shared Set Object Pointer.vi`, *VI* in *community scope* della classe *Data*, ereditato dalla classe *TreeNode*, derivata da *Data*. La classe *Data* ha come *friend* la classe *Tree*: tutti i *VI* di *Data* in *community scope* possono quindi essere utilizzati nel *block diagram* dei *VI* di *Tree*.

4.3.3 Sintassi *by-reference* e *dataflow*

Uno dei principali problemi incontrati durante la progettazione della nuova interfaccia è l'incompatibilità tra la sintassi *by-reference*, utilizzata per la manipolazione degli oggetti dell'interfaccia C++ del sistema, e la sintassi *by-value*, utilizzata per la gestione dei corrispondenti oggetti *LVOOP*. Il *mapping* tra oggetti C++ e oggetti *LVOOP*, descritto ad alto livello in 4.3.1, realizza di fatto una sintassi *by-reference* pur mostrando in apparenza una sintassi *by-value*: lo sdoppiamento di un filo contenente un oggetto non comporta infatti la creazione di una copia indipendente dell'oggetto, ma determina solamente la copia del valore del puntatore all'oggetto, contenuto in *Object Pointer*. Questa situazione è in netto contrasto con i principi del *dataflow* e può generare confusione nell'utilizzo della nuova interfaccia da parte di sviluppatori che non conoscono tutti i dettagli implementativi introdotti dall'operazione di *mapping*. In Figura 4.12 è rappresentato il *block diagram* di un *VI* ottenuto aggiungendo un *thread* al *VI* di Figura 4.4(a). Il nuovo *thread* si compone delle stesse operazioni descritte ai punti 5, 6, 7 e 8 di 4.3.1. A causa dello sdoppiamento del filo operato prima dell'esecuzione delle operazioni descritte al punto 4 di 4.3.1, il nuovo *thread* agisce sullo stesso oggetto *TreeNode* ritornato al punto 2 di 4.3.1. L'esecuzione da parte di uno dei due *thread* del *VI* distruttore dell'oggetto *TreeNode*, descritto al punto 8 di 4.3.1, pregiudica il corretto funzionamento del programma: quando infatti il secondo *thread* esegue a sua volta il *VI* distruttore dell'oggetto *TreeNode* oppure un altro *VI* che agisce sull'oggetto *TreeNode*, *CLFN* termina prematuramente l'esecuzione della funzione *wrapper* associata, che va in *crash* nel tentativo di eliminare un oggetto situato in una locazione di memoria già liberata dalla precedente operazione di distruzione oppure di accedere in memoria ad un oggetto inesistente. Per il *dataflow* gli oggetti contenuti in due fili che hanno un'origine comune, dal momento dello sdoppiamento in poi, sono del tutto indipendenti e non c'è modo di far passare l'informazione dell'avvenuta distruzione dell'oggetto dal primo al secondo filo. L'errore è dovuto al fatto che il *block diagram* in questione espone una sintassi *by-value*, ma in realtà funziona con una sintassi *by-reference*.

Questo problema può essere risolto uniformando la sintassi esposta dall'interfaccia con la sintassi realmente utilizzata. Dovendo operare un *mapping* delle classi C++ del sistema, gestite con una sintassi *by-reference*, in classi immagine *LVOOP*, la soluzione più ragionevole è dotare di una sintassi *by-reference* anche le classi immagine *LVOOP*. Uno dei modi più semplici ed efficaci per ottenere classi *LVOOP* gestite *by-reference* prevede la creazione di *VI wrapper* che incapsulino i *VI* membri delle classi *LVOOP* gestite *by-value*. I *VI wrapper* in questione hanno lo stesso *connector pane* e gli stessi terminali dei *VI* membri incapsulati, ad eccezione dei terminali preposti al passaggio degli oggetti, sostituiti da terminali che accettano *DVR* a quegli stessi oggetti. All'interno dei *VI wrapper* vengono dereferenziati i *DVR* in ingresso, che restituiscono degli oggetti sui quali i *VI* membri incapsulati possono agire. In Figura 4.13 si può osservare un *block diagram* che esegue *by-reference* gli stessi metodi delle classi *LVOOP* di Figura 4.12: in questo caso, non appena uno dei due *thread* esegue il distruttore di *TreeNode*, il *DVR* associato all'oggetto *TreeNode* appena distrutto

viene segnalato come inesistente dal sistema di gestione della memoria di LabVIEW e tutte le successive esecuzioni di *VI wrapper* che agiscono su quel *DVR* ritorneranno un errore, senza però mandare in *crash* l'applicazione, che riconosce che il *DVR* associato all'oggetto *TreeNode* è già stato distrutto. L'informazione riguardante la modifica o la distruzione di un *DVR* viene cioè istantaneamente propagata su tutti i fili che contengono una copia dello stesso *DVR*. Le differenze grafiche fra il *block diagram* di Figura 4.13 e il *block diagram* di Figura 4.12 consistono nel maggiore spessore dei fili contenenti gli oggetti *by-reference* (Figura 4.13) rispetto ai fili contenenti oggetti *by-value* (Figura 4.12) e nella presenza di un simbolo rappresentante una freccia sulla parte superiore colorata del *connector pane* dei *VI wrapper* (Figura 4.13), assente nella corrispondente area del *connector pane* dei *VI* membri (Figura 4.12).

Tutte le operazioni di modifica e distruzione di un oggetto dereferenziato da un *VI wrapper* sono *thread safe*: ogni *VI wrapper* utilizza infatti un meccanismo di *lock* e *unlock* che garantisce l'accesso in mutua esclusione all'oggetto contenuto nel *DVR*. I costrutti del linguaggio G che permettono di realizzare questo meccanismo di accesso in mutua esclusione sono *In Place Element* (*IPE*) e *Data Value Reference Read/Write Element Border Node* (*EBN*): la combinazione di questi due costrutti permette di ottenere un accesso serializzato ai *DVR* ad essa collegati. Eventuali accessi concorrenti vengono gestiti tramite una coda. Vengono riportati in Figura 4.14 il *connector pane* e il *block diagram* di `Tree Create (Mode).vi`, *VI wrapper* che incapsula il *VI* membro `Tree.lvclass:Create (Mode).vi`, costruttore della classe *Tree*: osservando il *block diagram* si può notare che `Tree.lvclass:Create (Mode).vi` passa l'oggetto *Tree* appena creato al *VI* membro in *public scope* `Tree.lvclass:New Tree DVR.vi`, responsabile della creazione del *DVR* per il nuovo oggetto *Tree*. Il *connector pane* e il *block diagram* di `Tree.lvclass:New Tree DVR.vi` sono riportati in Figura 4.17(a). Vengono riportati in Figura 4.15 il *connector pane* e il *block diagram* di `Tree Get Node.vi`, *VI wrapper* che incapsula il *VI* membro `Tree.lvclass:Get Node.vi`: osservando il *block diagram* si può identificare nel riquadro giallo che circonda `Tree.lvclass:Get Node.vi` il costrutto *IPE*, che ha lo scopo di rendere atomiche tutte le operazioni eseguite sull'oggetto *Tree* appena dereferenziato dalla parte del costrutto *EBN* agganciata al bordo sinistro di *IPE* stesso. La parte del costrutto *EBN* agganciata al bordo destro di *IPE* ha invece la funzione di ricostruire il *DVR* all'oggetto precedentemente dereferenziato. Il *VI* membro `Tree.lvclass:Get Node.vi` ritorna un nuovo oggetto *TreeNode*, di cui verrà creato un *DVR* dal *VI* membro in *public scope* `TreeNode.lvclass:New TreeNode DVR.vi`[13], l'analogo di `Tree.lvclass:New Tree DVR.vi` per la classe *TreeNode*. Tale *DVR* rappresenta l'oggetto *TreeNode* ritornato *by-reference* da `Tree Get Node.vi`. Vengono riportati in Figura 4.16 il *connector pane* e il *block diagram* di `Tree Destroy.vi`, *VI wrapper* che incapsula il *VI* membro `Tree.lvclass:Destroy.vi`, distruttore della classe *Tree*: osservando il *block diagram* si può notare che il *DVR* all'oggetto *Tree*, passato *by-reference* al terminale di ingresso di `Tree Destroy.vi`, viene eliminato dal *VI* membro in *public scope*

[13]  *Connector pane* di `TreeNode.lvclass:New TreeNode DVR.vi`.

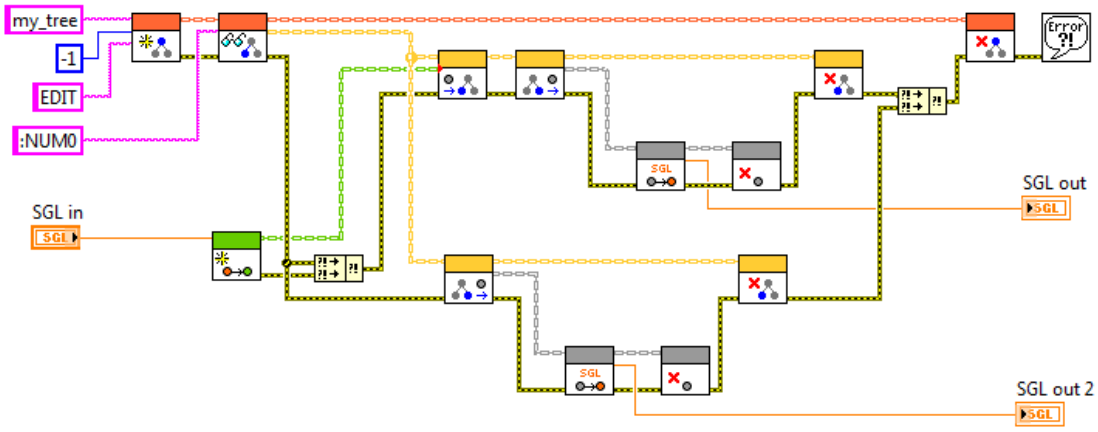


Figura 4.12: *Block diagram* di un *VI* di esempio che illustra i limiti della sintassi *by-value* nella gestione delle classi immagine *LVOOP*.

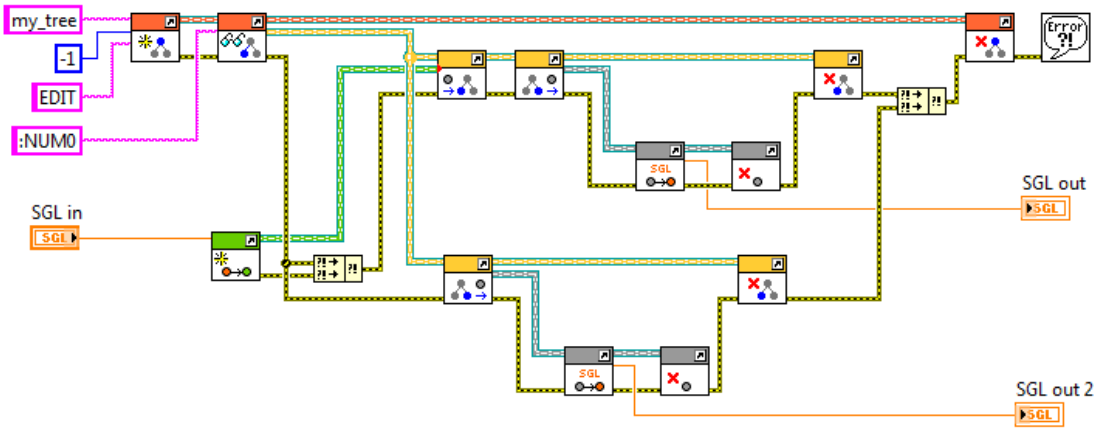


Figura 4.13: *Block diagram* del *VI* di esempio di Figura 4.12 trasformato secondo la sintassi *by-reference* per superare i limiti della sintassi *by-value* nella gestione delle classi immagine *LVOOP*.

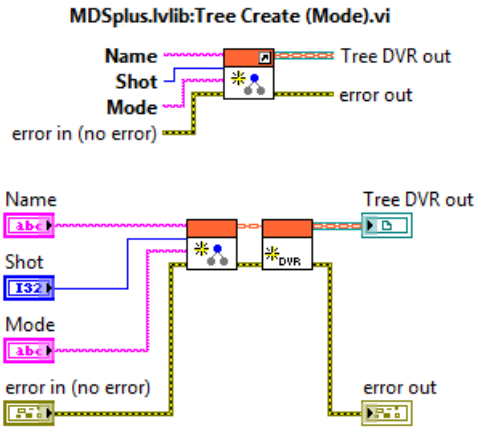


Figura 4.14: *Connector pane* e *block diagram* di `Tree Create (Mode).vi`.

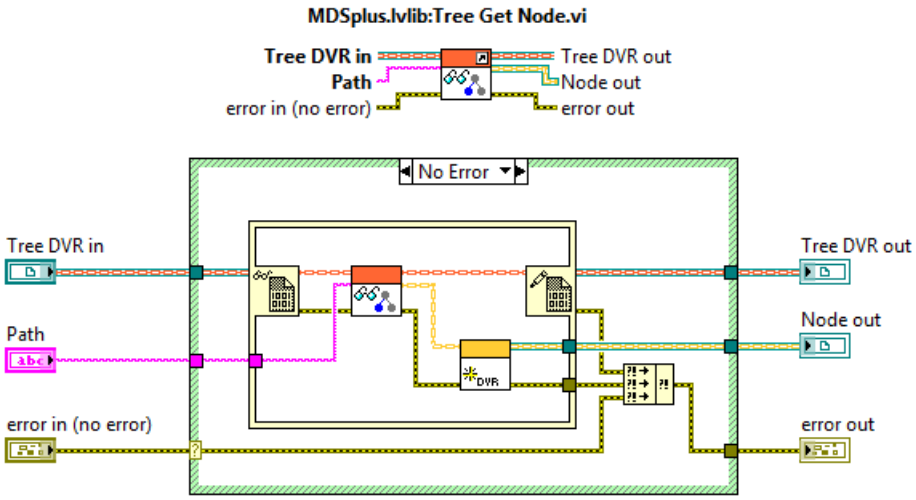


Figura 4.15: *Connector pane* e *block diagram* di `Tree Get Node.vi`.

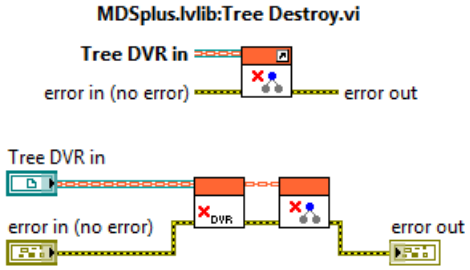
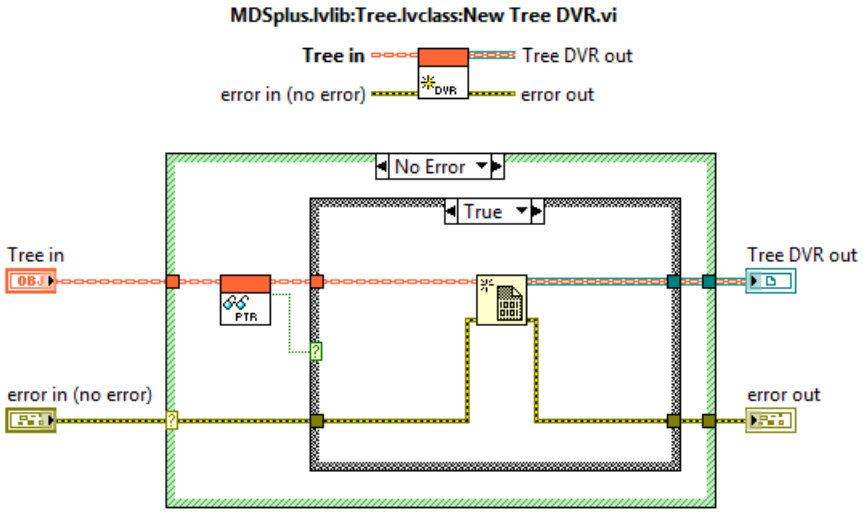
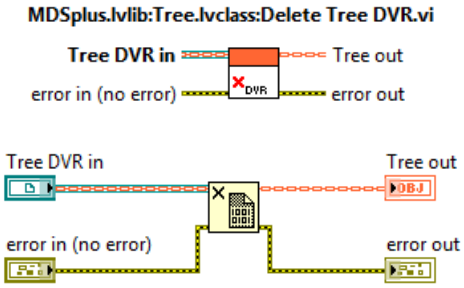


Figura 4.16: *Connector pane* e *block diagram* di `Tree Destroy.vi`.



(a)



(b)

Figura 4.17: (a) *Connector* pane e *block diagram* di `Tree.lvclass:New DVR.vi`; (b) *Connector* pane e *block diagram* di `Tree.lvclass:Delete DVR.vi`.

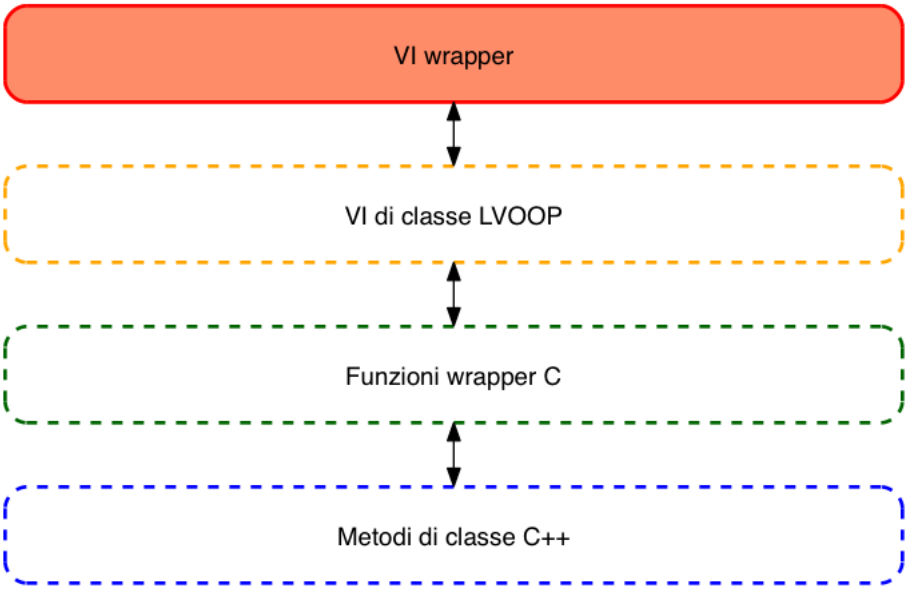


Figura 4.18: *Mapping* tra classi C++ e classi *LVOOP* *by-reference* realizzato attraverso l'utilizzo di *VI wrapper* e *DVR*.

`Tree.lvclass:Delete Tree DVR.vi`, che passa l'oggetto *Tree* appena dereferenziato al *VI* distruttore `Tree.lvclass:Destroy.vi`, che provvede all'effettiva eliminazione dell'oggetto. Il *connector pane* e il *block diagram* di `Tree.lvclass:Delete Tree DVR.vi` sono riportati in Figura 4.17(b). Si sottolinea che la cancellazione del *DVR* all'oggetto *Tree* non comporta l'eliminazione dell'oggetto, ma corrisponde ad un'operazione di dereferenziazione del *DVR* all'oggetto combinata con la cancellazione definitiva dal sistema di gestione della memoria di LabVIEW del *DVR* stesso. Attraverso l'utilizzo combinato di *IPE* ed *EBN* viene garantito il rispetto dei vincoli progettuali descritti in 4.2.4 e in 4.2.5: impiegando questi due costrutti è infatti possibile rendere *thread-safe* l'esecuzione di uno o più *VI* che agiscono su oggetti gestiti *by-reference*.

Ogni *VI* membro delle classi *LVOOP* di MDSplus viene incapsulato nel corrispondente VI *wrapper*, con le stesse modalità utilizzate per i *VI* della classe *Tree* appena descritti. La nuova interfaccia, illustrata in Figura 4.18, è quindi rappresentata dai *VI wrapper*, che realizzano un *mapping* coerente e completo dei metodi delle classi della versione C++ del sistema, utilizzando una sintassi *by-reference* per la creazione, la distruzione e la gestione degli oggetti.

4.3.4 Sintesi architetturale

In Figura 4.21 viene fornita una rappresentazione sintetica dell'architettura complessiva della nuova interfaccia LabVIEW di MDSplus, riassumendo graficamente le principali caratteristiche descritte in 4.3.1, 4.3.2 e 4.3.3. In Figura 4.20 viene illustrato il

mapping del metodo `TreeNode *Tree::getNode(char *path)` nel *VI wrapper* `Tree Get Node.vi` operato secondo l'architettura proposta, mostrando le interfacce coinvolte nei vari livelli architetturali. In Figura 4.21 viene riproposto il *mapping* di Figura 4.20, mettendo però in evidenza l'implementazione in ogni livello dell'architettura.

4.4 Dettagli implementativi

In questa sezione si descrivono alcuni degli aspetti più importanti che riguardano l'implementazione della nuova interfaccia, fornendo degli esempi specifici per rendere più facile la comprensione di quanto trattato.

4.4.1 *CLFN* e funzioni *wrapper*

La maggior parte delle funzioni *wrapper* collegate a *CLFN* ritorna dati di tipo semplice, come puntatori o valori numerici. Questi dati vengono creati in LabVIEW, passati per riferimento alla funzione *wrapper* e infine ritornati a LabVIEW. La funzione *wrapper* si limita ad elaborare e modificare il dato già allocato, senza preoccuparsi della creazione e della distruzione di nuovi dati in memoria. Quando una funzione *wrapper* elabora e ritorna elementi come array o stringhe deve invece provvedere al ridimensionamento del dato, solitamente vuoto, passato a *CLFN* come *handle*[14]. Una funzione *wrapper* che ritorni in uscita tipi di dati composti, come array di stringhe, deve in aggiunta gestire l'allocazione di nuovo spazio in memoria. In questi ultimi due casi non è possibile utilizzare funzioni standard C per l'allocazione e il rilascio della memoria: LabVIEW ha infatti un proprio sistema di gestione della memoria, che permette a *CLFN* di essere eseguibile su piattaforme diverse. Per compiere operazioni di gestione della memoria e di accesso al disco all'interno del codice di funzioni *wrapper* collegate a *CLFN* è quindi necessario utilizzare le specifiche funzioni di gestione della memoria e gli specifici tipi di dati previsti da LabVIEW. Una descrizione esaustiva di tutte le funzioni e i tipi di dati disponibili si può trovare in [34]. In Algoritmo 4.5 viene illustrata la funzione *wrapper* di `char **Array::getStringArray(int *numElements)`, metodo della classe *Array* che ritorna un array di stringhe in uscita. Si può notare l'utilizzo delle funzioni *DSNewHandle()*, *NumericArrayResize()* e *MoveBlock()*, utilizzate rispettivamente per la creazione, il ridimensionamento e lo spostamento di dati in memoria. L'array di stringhe di uscita `lvStrArrHdlOut`, di tipo `LStrArrHdl`, è passato come *handle* alla funzione *wrapper* per essere elaborato dalla stessa e poi ritornato. Il tipo di dato `LStrArrHdl` è definito in Algoritmo 4.6, dove sono definiti anche tutti gli altri tipi di dati compositi utilizzati per interfacciare *CLFN* e funzioni *wrapper*. La distruzione dei dati allocati nelle funzioni *wrapper* e ritornati da *CLFN* viene gestita autonomamente da LabVIEW secondo i principi del *dataflow*.

[14] Un *handle* di un dato in memoria è un doppio puntatore del dato stesso.

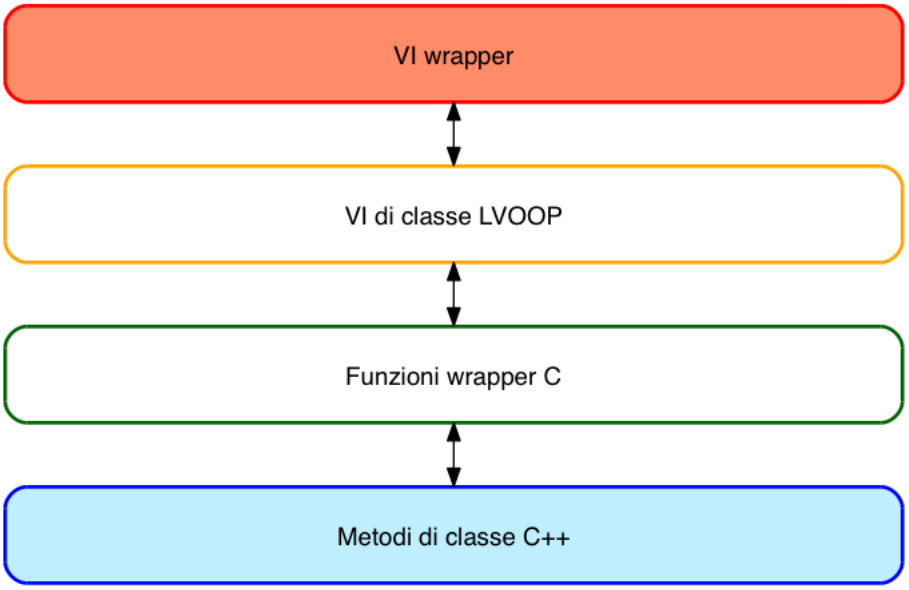


Figura 4.19: Architettura della nuova interfaccia LabVIEW di MDSplus.

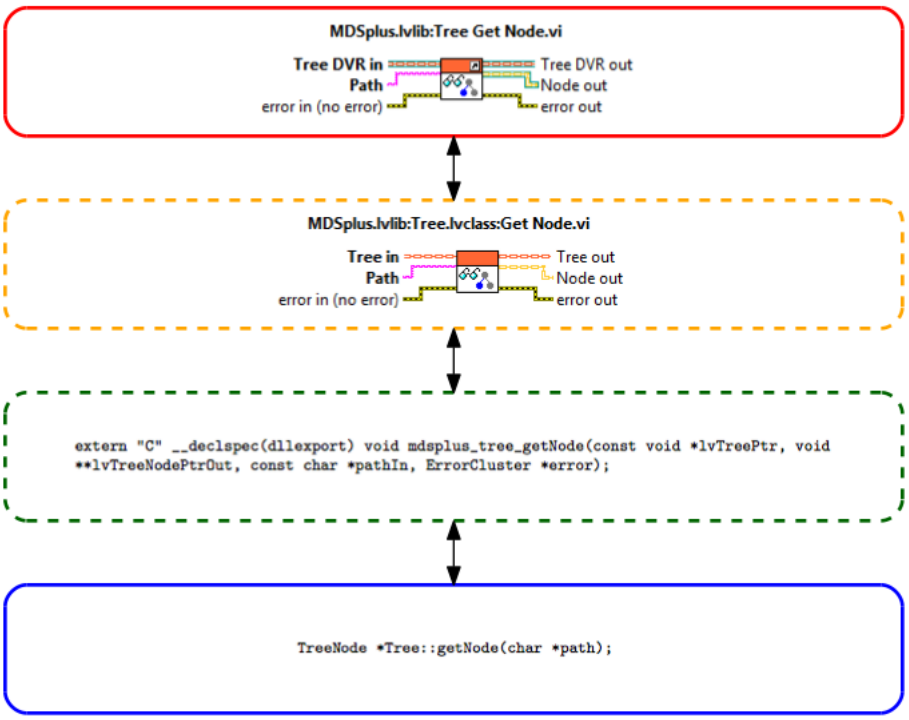


Figura 4.20: Illustrazione dei vari passaggi del *mapping* del metodo `TreeNode *Tree::getNode(char *path)` nel *VI wrapper* `Tree Get Node.vi`: vengono riportate le trasformazioni dell'interfaccia per ogni livello architetturale.

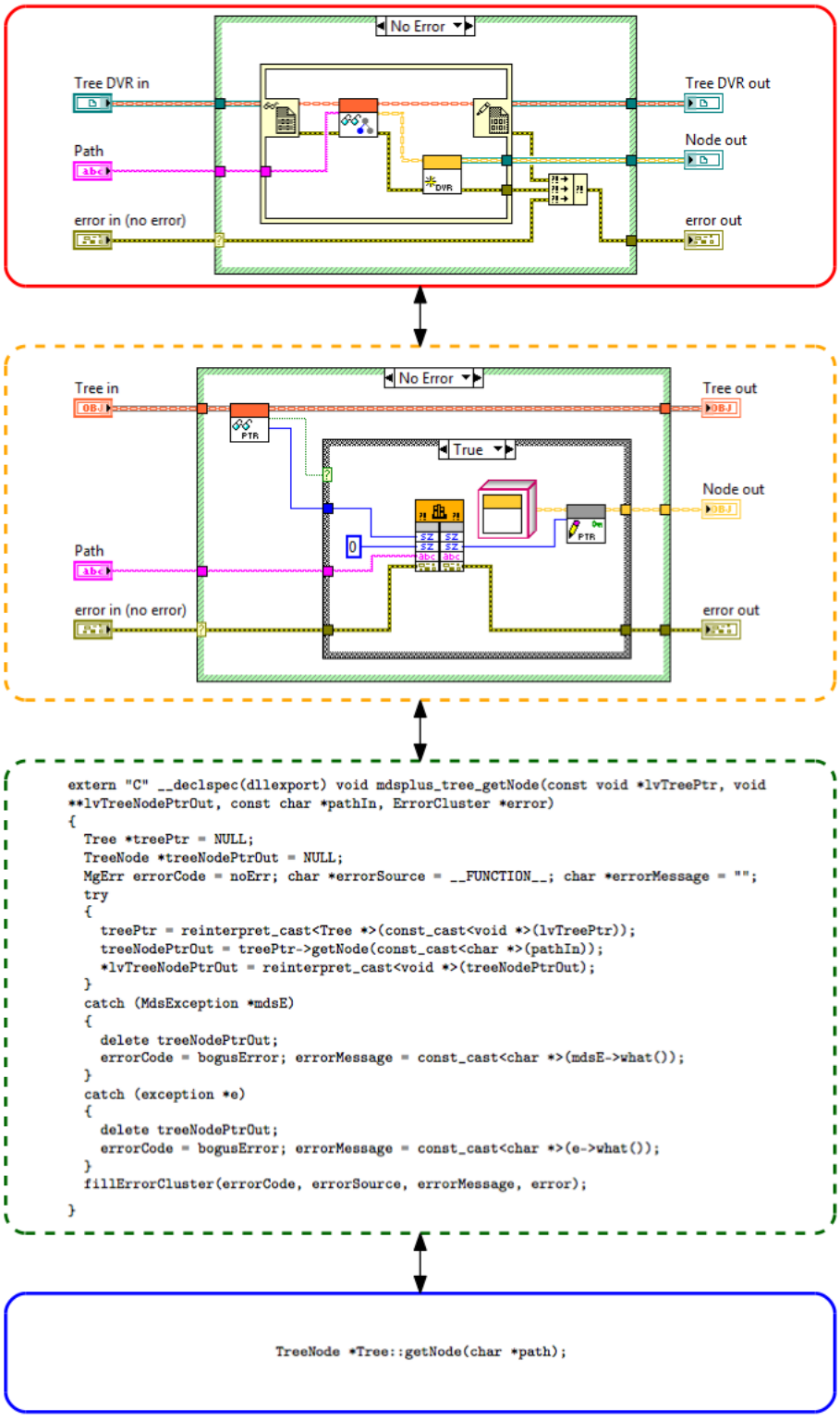


Figura 4.21: Illustrazione dei vari passaggi del *mapping* del metodo `TreeNode *Tree::getNode(char *path)` nel *VI wrapper* `Tree Get Node.vi`: vengono riportate le implementazioni dell'interfaccia per ogni livello architetturale.

Algoritmo 4.5 Funzione *wrapper* del metodo *getStringArray()* della classe *Array*

```
extern "C" __declspec(dllexport) void mdsplus_array_getStringArray(const void *lvArrayPtr, LStrArrHdl lvStrArrHdlOut,
ErrorCluster *error)
{
  Array *arrayPtr = NULL;
  char **stringArrOut = NULL;
  int stringArrLen = 0;
  MgErr errorCode = noErr; char *errorSource = __FUNCTION__; char *errorMessage = "";
  try
  {
    arrayPtr = reinterpret_cast<Array *>(const_cast<void *>(lvArrayPtr));
    stringArrOut = arrayPtr->getStringArray(&stringArrLen);
    // checks whether the size of a pointer is 32 bit or 64 bit depending upon the system architecture
    int32 typeCode = (sizeof(LStrHandle) > sizeof(int32)) ?  iQ : iL;
    // resizes string array
    errorCode = NumericArrayResize(typeCode, 1, reinterpret_cast<UHandle *>(&lvStrArrHdlOut),
      static_cast<int32>(stringArrLen));
    if (!errorCode)
    {
      int i = 0;
      // creates LStrHandle strings and fills them with the stringArrOut corrispondent strings
      while (!errorCode && i < stringArrLen)
      {
        char *currStr = stringArrOut[i];
        int32 currStrLen = static_cast<int32>(strlen(currStr));
        LStrHandle currLStrHdl = reinterpret_cast<LStrHandle>(DSNewHandle(sizeof(LStrHandle)));
        errorCode = currLStrHdl == NULL;
        if (!errorCode)
        {
          errorCode = NumericArrayResize(uB, 1, reinterpret_cast<UHandle *>(&currLStrHdl), currStrLen);
          if (!errorCode)
          {
            MoveBlock(reinterpret_cast<uChar *>(currStr), LStrBuf(*currLStrHdl), currStrLen);
            (*currLStrHdl)->cnt = currStrLen;
            (*lvStrArrHdlOut)->elm[i++] = currLStrHdl;
          }
          else
            errorMessage = "NumericArrayResize error";
        }
        else
          errorMessage = "DSNewHandle error";
      }
      // keeps only well allocated string handles, till the ith string
      (*lvStrArrHdlOut)->dimSize = static_cast<int32>(i);
    }
    else
      errorMessage = "NumericArrayResize error";
    // frees memory
    for (int i = 0; i < stringArrLen; i++)
      deleteString(stringArrOut[i]);
    deleteNativeArray(stringArrOut);
  }
  catch (MdsException *mdsE)
  {
    for (int i = 0; i < stringArrLen; i++)
      deleteString(stringArrOut[i]);
    deleteNativeArray(stringArrOut);
    errorCode = bogusError;
    errorMessage = const_cast<char *>(mdsE->what());
  }
  catch (exception *e)
  {
    for (int i = 0; i < stringArrLen; i++)
      deleteString(stringArrOut[i]);
    deleteNativeArray(stringArrOut);
    errorCode = bogusError;
    errorMessage = const_cast<char *>(e->what());
  }
  fillErrorCluster(errorCode, errorSource, errorMessage, error);
}
```

Algoritmo 4.6 Definizione *handle* di array contenenti valori numerici e stringhe

```
#include <extcode.h>

/* lv_prolog.h and lv_epilog.h set up the correct alignment for LabVIEW data.  */

#include <lv_prolog.h>

typedef struct {
  LVBoolean status;
  int32 code;
  LStrHandle source;
} ErrorCluster;

typedef struct {
  int32 dimSize;
  int8 elt[1];
} LByteArr;
typedef LByteArr **LByteArrHdl;

typedef struct {
  int32 dimSize;
  uInt8 elt[1];
} LUByteArr;
typedef LUByteArr **LUByteArrHdl;

typedef struct {
  int32 dimSize;
  double elt[1];
} LDblArr;
typedef LDblArr **LDblArrHdl;

typedef struct {
  int32 dimSize;
  float elt[1];
} LFltArr;
typedef LFltArr **LFltArrHdl;

typedef struct {
  int32 dimSize;
  int32 elt[1];
} LIntArr;
typedef LIntArr **LIntArrHdl;

typedef struct {
  int32 dimSize;
  uInt32 elt[1];
} LUIntArr;
typedef LUIntArr **LUIntArrHdl;

typedef struct {
  int32 dimSize;
  int64 elt[1];
} LLngArr;
typedef LLngArr **LLngArrHdl;

typedef struct {
  int32 dimSize;
  uInt64 elt[1];
} LULngArr;
typedef LULngArr **LULngArrHdl;

typedef struct {
  int32 dimSize;
  int16 elt[1];
} LShtArr;
typedef LShtArr **LShtArrHdl;

typedef struct {
  int32 dimSize;
  uInt16 elt[1];
} LUShtArr;
typedef LUShtArr **LUShtArrHdl;

typedef struct {
  int32 dimSize;
  LStrHandle elm[];
} LStrArr;
typedef LStrArr **LStrArrHdl;

typedef struct {
  int32 dimSize;
  void *elt[1];
} LPtrArr;
typedef LPtrArr **LPtrArrHdl;

#include <lv_epilog.h>
```

4.4.2 Politica di distruzione degli oggetti

La nuova interfaccia LabVIEW di MDSplus prevede che un oggetto venga creato *by-reference* dal proprio costruttore, esegua una serie di operazioni per mezzo dei *VI wrapper* e venga infine eliminato dal proprio distruttore. La mancata distruzione di un oggetto gestito *by-reference* avrebbe come effetto collaterale la permanenza in memoria dell'oggetto C++ associato. Ogni oggetto gestito *by-reference* va quindi esplicitamente distrutto, a meno che la distruzione non venga gestita implicitamente dai *VI wrapper* che ne fanno uso. Si prenda in considerazione il *block diagram* di Figura 4.13: si può osservare che ogni filo ha origine da uno specifico *VI wrapper* e che quasi tutti i fili terminano in *VI wrapper* distruttori. Un filo può tuttavia morire nel terminale di ingresso di un *VI wrapper* che non è un distruttore, ma un semplice utilizzatore dell'oggetto contenuto nel filo: se il *VI wrapper* utilizzatore non è provvisto di un terminale di uscita corrispondente al terminale di ingresso per quello specifico oggetto, allora l'oggetto in questione non sarà più disponibile dalla fine dell'esecuzione del *VI* in poi: la sua distruzione sarà infatti eseguita implicitamente dallo stesso *VI wrapper* oppure da uno dei metodi di MDSplus successivamente invocati. Questa situazione si presenta con l'esecuzione di `TreeNode Put Data.vi`[15], che prende in ingresso il filo contenente l'oggetto *Float32* creato da `Float32 Create.vi`[16]: si può vedere che il filo va a morire nel terminale di ingresso di `TreeNode Put Data.vi`, che gestisce la distruzione dell'oggetto in esso contenuto. La politica di distruzione degli oggetti della nuova interfaccia LabVIEW di MDSplus si può riassumere nella seguente asserzione.

Asserzione. Se un *VI wrapper* è dotato di un terminale di uscita per un un dato oggetto, allora quell'oggetto sicuramente non verrà deallocato. In caso contrario l'oggetto verrà automaticamente deallocato dal sistema nel momento più opportuno.

4.4.3 Gestione di eccezioni ed errori

La nuova interfaccia LabVIEW del sistema prevede che anche le eventuali eccezioni sollevate dai metodi C++ vengano mappate in errori gestibili a livello di *VI* membri *LVOOP* e di *VI wrapper*. Il *mapping* delle eccezioni C++ in errori LabVIEW è gestito a livello di funzioni *wrapper* tramite il costrutto *try/catch*: un'eventuale eccezione sollevata da un metodo del sistema viene catturata dai blocchi *catch* della funzione *wrapper* immagine del metodo e viene elaborata in modo tale da essere trasformata in un *cluster* di errore. Tale *cluster* costituisce uno dei parametri di uscita per ogni funzione *wrapper*. La costruzione del *cluster* di errore viene eseguita in ogni funzione *wrapper* tramite *fillErrorCluster()*, funzione riportata in Algoritmo 4.7. Il *cluster* di errore ritornato da *fillErrorCluster()* è di tipo `ErrorCluster`, tipo di dato definito in Algoritmo 4.6 e pienamente compatibile con il contenuto del filo

[15]  *Connector pane* di `TreeNode Put Data.vi`.

[16]  *Connector pane* di `Float32 Create.vi`.

Algoritmo 4.7 Costruzione del *cluster* di errore

```
void fillErrorCluster(MgErr code, const char *source, const char *message, ErrorCluster
*error)
{
  if (code)
  {
    char *errMsg = new char[strlen(source) + strlen(message) + strlen("<ERR>..")];
    strcpy(errMsg, source);
    strcat(errMsg, ".");
    strcat(errMsg, "<ERR>");
    strcat(errMsg, message);
    strcat(errMsg, ".");
    int32 errMsgLen = static_cast<int32>(strlen(errMsg));
    error->status = LVBooleanTrue;
    error->code = static_cast<int32>(code);
    if (!NumericArrayResize(uB, 1, reinterpret_cast<UHandle *>(&(error->source)), errMsgLen))
    {
      MoveBlock(reinterpret_cast<uChar *>(errMsg), LStrBuf(*error->source), errMsgLen);
      (*error->source)->cnt = errMsgLen;
    }
    delete[] errMsg;
  }
}
```

dell'errore LabVIEW. In Algoritmo 4.3 si può osservare l'utilizzo del blocco *try/catch* e di *fillErrorCluster()* per le gestione degli errori nella funzione *wrapper* del metodo *getNode()* di *Tree*.

4.5 Test incrementale

Durante la fase di progettazione e implementazione della nuova interfaccia sono state costruite numerose applicazioni di prova, finalizzate a testare le nuove funzionalità via via codificate. Un esempio di tali applicazioni di prova sono i *block diagram* illustrati in Figura 4.4(a), in Figura 4.12 e in Figura 4.13. L'applicazione che tuttavia costituisce il banco di prova per la verifica del funzionamento delle principali funzionalità di MDSplus è descritta in dettaglio nel capitolo 5.

5 Applicazione di test

L'obiettivo fondamentale del progetto di tesi è la realizzazione di una nuova interfaccia LabVIEW di MDSplus in grado di supportare l'intero insieme di funzionalità del sistema, in particolare l'acquisizione continua di dati sperimentali. In questo capitolo viene descritta un'applicazione di test che realizza in LabVIEW il processo di acquisizione continua dei dati: la corretta esecuzione dell'acquisizione costituirà la verifica della correttezza delle scelte progettuali effettuate e dell'implementazione realizzata.

5.1 Descrizione

L'applicazione di acquisizione deve essere in grado di salvare su disco in modo continuo i dati prodotti dagli apparati sperimentali, secondo quanto descritto in 2.3.4. In termini pratici si può pensare a questa applicazione di test come ad una generalizzazione del processo di acquisizione continua dei dati descritto in Algoritmo 2.1. Per rendere riproducibile l'esecuzione dell'applicazione su macchine non collegate con hardware National Instruments, la produzione dei segnali sperimentali viene simulata da un *VI* in grado di generare specifiche forme d'onda e di aggiungere rumore. Per verificare la correttezza delle operazioni eseguite dalla nuova interfaccia viene operato un confronto tra la visualizzazione del segnale acquisito fornita dall'applicazione di test e la visualizzazione dello stesso segnale fornita da *jScope*. La corrispondenza tra le due visualizzazioni decreterà la correttezza delle operazioni eseguite tramite la nuova interfaccia.

5.2 Implementazione

In questa sezione viene trattata l'implementazione dell'applicazione di test attraverso una descrizione dei controlli posizionati sul relativo *front panel* e dei *VI* utilizzati nel relativo *block diagram*.

5.2.1 *Front panel*

In Figura 5.1 si può osservare il *front panel* del *VI* principale, ovvero l'interfaccia utente dell'applicazione di test. Il riquadro in alto a sinistra contiene i controlli per impostare i parametri relativi al *tree* e al *node* utilizzati per il test, ovvero:

- `Name`: nome del *tree* da aprire (in questo caso `my_tree`).
- `Shot`: numero di *shot* considerato (in questo caso `-1`).

- `Mode`: modalità di apertura del *tree* (necessariamente `EDIT`).
- `Path`: percorso del *node* del *tree* in cui verrà salvato il segnale acquisito (in questo caso `:SIG_0`).

Il riquadro in basso a sinistra contiene invece i valori relativi alla simulazione del segnale, che andranno a costituire i parametri di ingresso per *Simulate Signal Express VI*, il *VI* utilizzato per la generazione di forme d'onda. In Figura 5.2 è rappresentato il *connector pane* di *Simulate Signal Express VI*, mentre in Figura 5.3 è riportata la finestra di dialogo utilizzata per l'impostazione dei parametri. In questo caso *Simulate Signal Express VI* è stato impostato per produrre un segnale sinusoidale disturbato da rumore. I parametri di ingresso sono i seguenti:

- `frequency`: frequenza in Hz della sinusoide (in questo caso `1,1` Hz).
- `amplitude`: ampiezza della sinusoide (in questo caso `5`).
- `phase`: fase iniziale della sinusoide (in questo caso `0°`).

All'interno del riquadro in basso a sinistra si può notare la presenza aggiuntiva di un *cluster* denominato `sampling info`, composto dai seguenti controlli:

- `Fs`: frequenza di campionamento in Hz del segnale prodotto (in questo caso `1` kHz).
- `#s`: numero di campioni per ogni *segment* (in questo caso `1000`).

Per il corretto funzionamento dell'applicazione il valore del controllo `Samples per second (Hz)` di Figura 5.3 deve corrispondere al valore di `Fs`, mentre il valore del controllo `Number of samples` di Figura 5.3 deve corrispondere a `#s`. *Simulate Signal Express VI* deve cioè produrre in uscita solamente `#s` valori, ottenuti campionando la sinusoide con frequenza `Fs`. In questo caso *Simulate Signal Express VI* è configurato per produrre dati in uscita per `1` secondo.

Il segmento di dati prodotto da *Simulate Signal Express VI* verrà visualizzato nell'indicatore `Waveform Graph` posizionato in alto a destra. L'indicatore `Waveform Graph 2`, posizionato in basso a destra, ha invece la funzione di visualizzare il segnale complessivamente acquisito al termine dell'applicazione. Per concludere, il pulsante `STOP`, posizionato in basso a sinistra, ha la funzione di interrompere l'acquisizione continua dei dati.

5.2.2 *Block diagram*

In Figura 5.4 si può osservare il *block diagram* dell'applicazione di test, composto da un *thread* produttore denominato *Producer* e da un *thread* consumatore denominato *Consumer*.

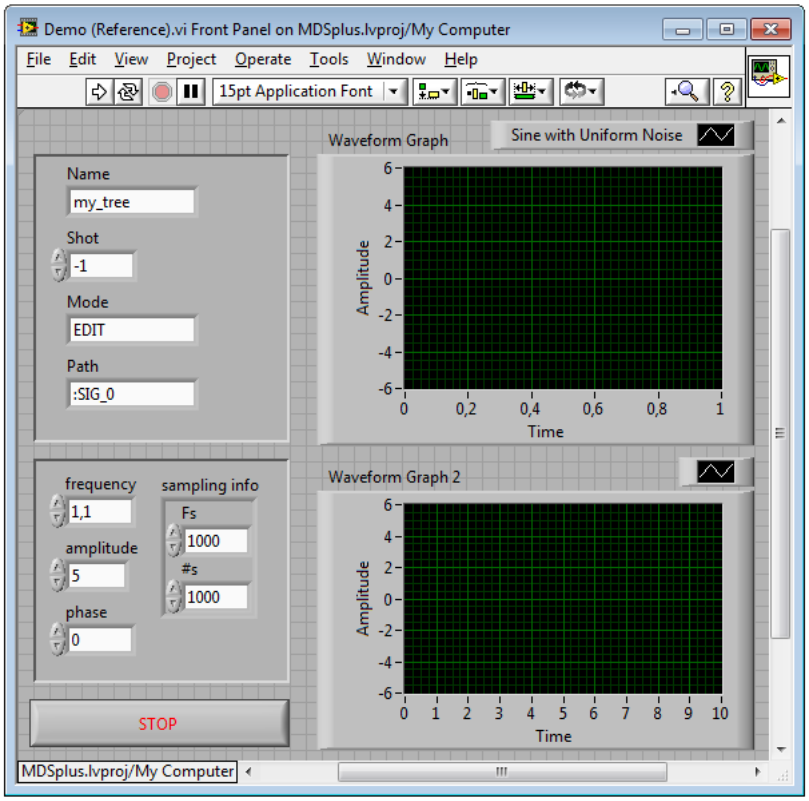


Figura 5.1: *Front panel* dell'applicazione di test.

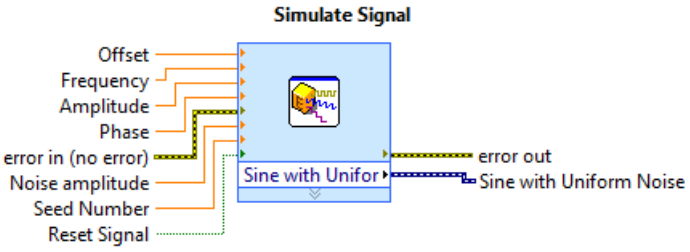


Figura 5.2: *Connector pane* di *Simulate Signal Express VI* impostato per generare una sinusoide disturbata da rumore.

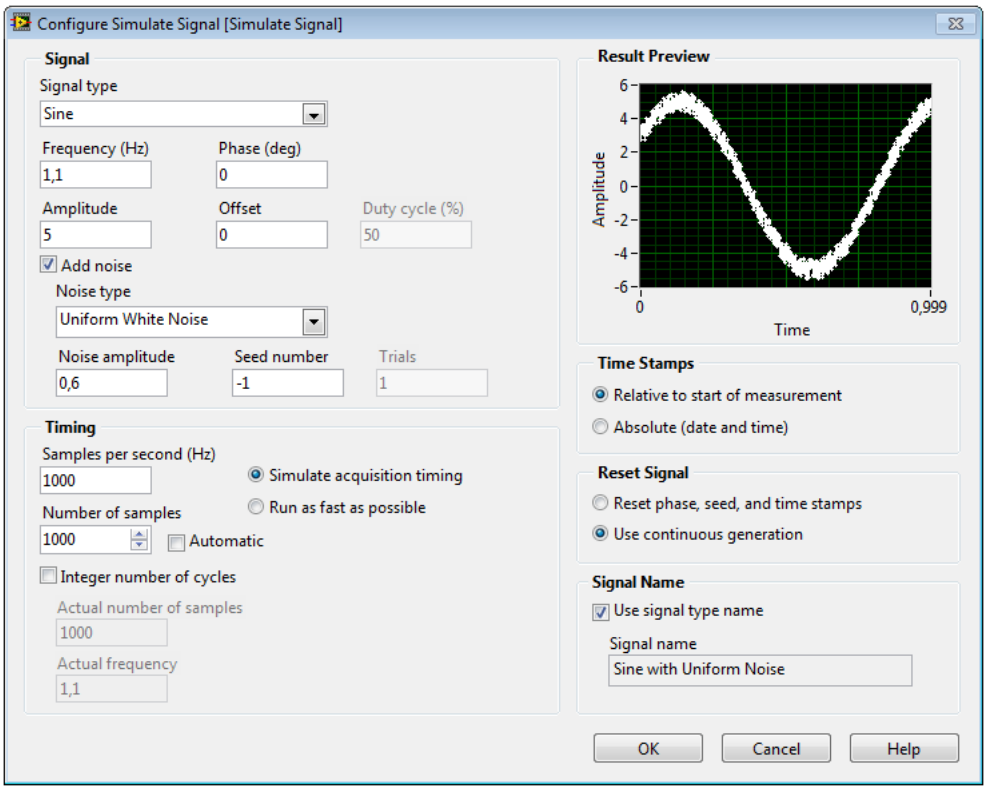


Figura 5.3: Finestra di dialogo per l'impostazione dei parametri di *Simulate Signal Express VI*.

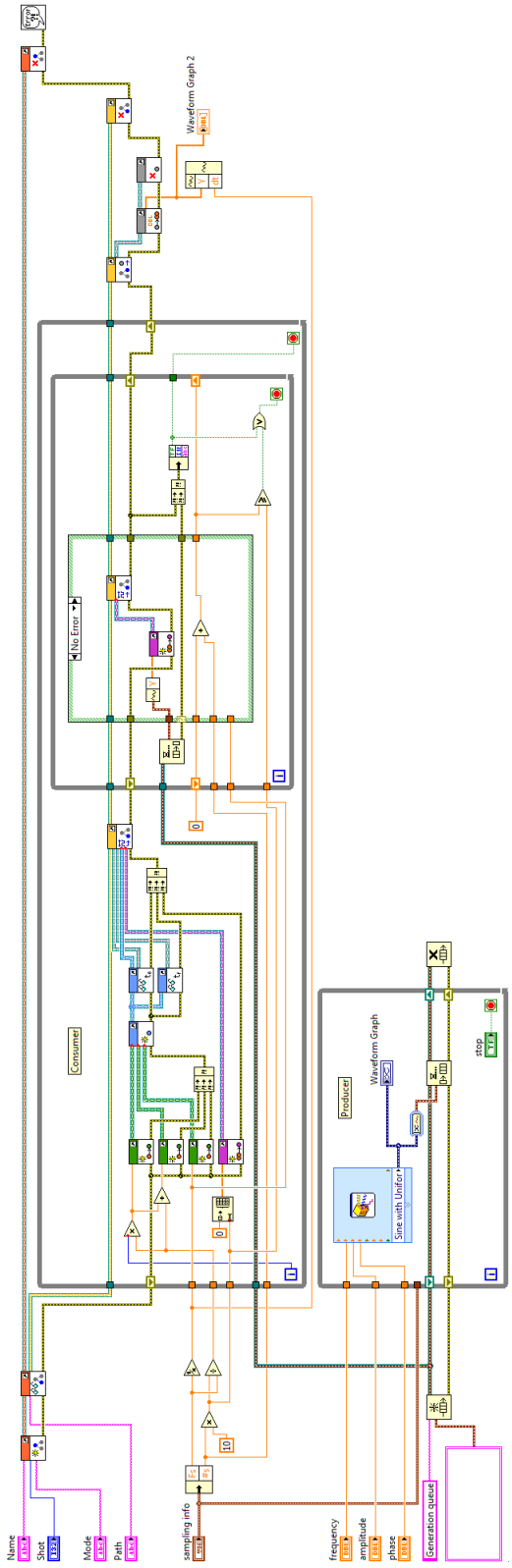


Figura 5.4: *Block diagram* dell'applicazione di test.

Producer

Producer è il *thread* posizionato in basso, di dimensioni minori, che esegue ininterrottamente un ciclo in cui avviene la produzione di un segmento di segnale e il salvataggio dello stesso in una coda appositamente creata. La produzione del segnale viene gestita da *Simulate Signal Express VI* secondo le modalità descritte in 5.2.1. Il segmento di segnale creato ad ogni iterazione viene inoltre visualizzato in `Waveform Graph`.

Consumer

Consumer è il *thread* posizionato in alto, di dimensioni maggiori, che esegue ininterrottamente un ciclo in cui avviene il salvataggio dei segmenti prodotti da *Producer* nel *node* di percorso `Path` appartenente allo *shot* `-1` del *tree* `my_tree`. Il ciclo termina quando il produttore non produce più segmenti. I segmenti prodotti da *Producer* vengono passati a *Consumer* tramite la coda definita in *Producer*. Il concetto di segmento è approfondito in 2.3.4. È interessante analizzare più in dettaglio le operazioni compiute da *Consumer*, che si possono suddividere in tre fasi:

1. *Inizializzazione*: viene aperto il *tree* tramite `Tree Create (Mode).vi`[1], viene ritornato il *node* al percorso `:SIG_0` tramite `Tree Get Node.vi`[2] e vengono fatte delle operazioni su `Fs` e `#s`, che restituiscono il periodo di campionamento (in questo caso `1` ms), la dimensione in numero di campioni dell'array contenente i dati da salvare su disco ad ogni iterazione del ciclo di acquisizione (in questo caso `10000`) e il numero di segmenti contenuti in tale array (in questo caso `10`).

2. *Acquisizione*: ad ogni iterazione del ciclo di acquisizione tramite `TreeNode Begin Segment.vi`[3] viene allocato lo spazio in memoria per contenere `10` segmenti. `TreeNode Begin Segment.vi` prende in ingresso l'oggetto *TreeNode* rappresentante il *node* `SIG_0`, due oggetti *Data* ritornati da `Range Get Begin.vi`[4] e da `Range Get Ending.vi`[5] e rappresentanti rispettivamente il tempo di inizio e il tempo di fine in secondi dell'attuale serie di `10` segmenti da acquisire, un oggetto *Range* creato da `Range Create.vi`[6] a partire dal tempo di inizio e dal tempo di fine in secondi dell'attuale serie di `10` segmenti da acquisire e dal

---

1  *Connector pane* di `Tree Create (Mode).vi`.

2  *Connector pane* di `Tree Get Node.vi`.

3  *Connector pane* di `TreeNode Begin Segment.vi`.

4  *Connector pane* di `Range Get Begin.vi`.

5  *Connector pane* di `Range Get Ending.vi`.

6  *Connector pane* di `Range Create.vi`.

periodo di campionamento, rispettivamente ritornati da `Float64 Create.vi`[7], e infine un oggetto *Float64Array*, creato da `Float64Array Create.vi`[8], rappresentante l'array di campioni da allocare in memoria. A questo punto inizia l'esecuzione del ciclo interno: ad ogni iterazione viene estratto dalla coda un segmento, che viene trasformato in un oggetto *Float64Array* da `Float64Array Create.vi` e quindi scritto su disco da `TreeNode Put Segment.vi`[9]. In questo caso il ciclo interno è composto di `10` iterazioni.

3. *Visualizzazione*: `TreeNode Get Data.vi`[10] ritorna un oggetto *Data* contenente l'intera sequenza di campioni scritti su disco nella fase precedente. Tramite `Data Get Double Array.vi`[11] la stessa sequenza viene visualizzata su `Waveform Graph 2`. Seguono le distruzioni degli oggetti utilizzati.

5.3 Risultati ottenuti

Eseguendo l'applicazione per una decina di secondi si ottiene in uscita il segnale riportato dall'indicatore `Waveform Graph 2` di Figura 5.5. Confrontando questo segnale con il segnale visualizzato tramite *jScope* (Figura 5.6) e relativo allo stesso *node* `SIG_0` di `my_tree`, si può osservare una sostanziale coincidenza tra i due. Il successo dell'applicazione di test porta a concludere che la nuova interfaccia LabVIEW di MDSplus esegue correttamente l'acquisizione continua di segnali, banco di prova per il test complessivo della nuova interfaccia, e si può quindi considerare in grado di supportare tutte le principali funzionalità del sistema.

---

7 *Connector pane* di `Float64 Create.vi`.

8 *Connector pane* di `Float64Array Create.vi`.

9 *Connector pane* di `TreeNode Put Segment.vi`.

10 *Connector pane* di `TreeNode Put Data.vi`.

11 *Connector pane* di `Data Get Double Array.vi`.

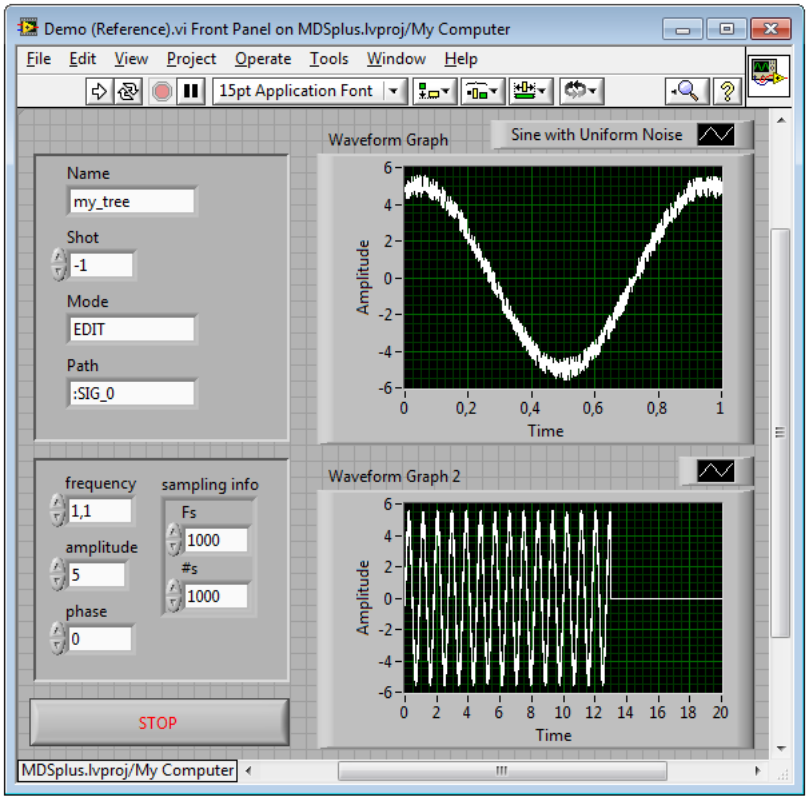


Figura 5.5: *Front panel* dell'applicazione di test ad esecuzione terminata.

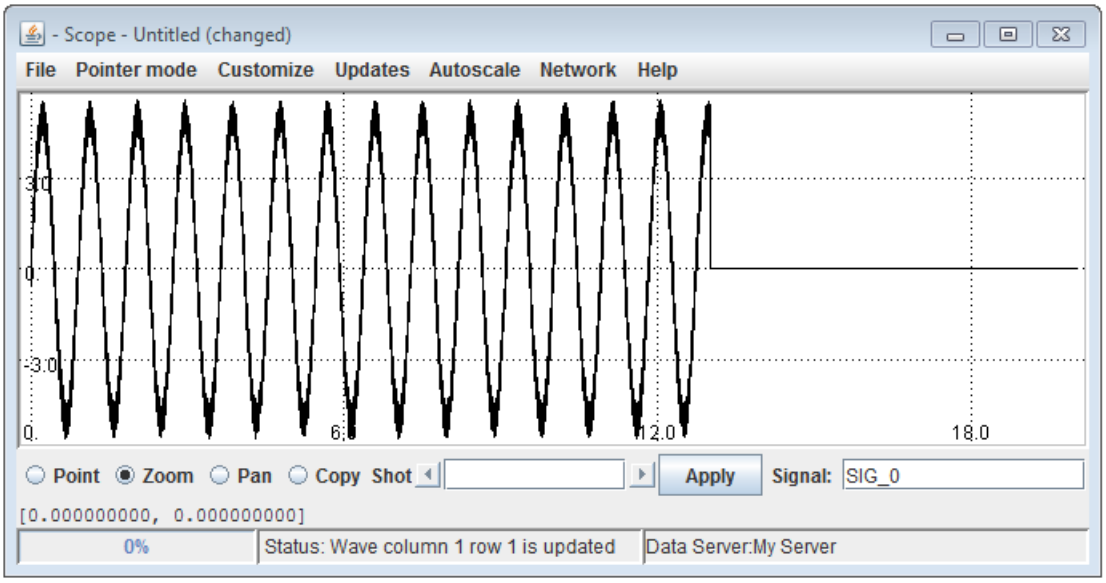


Figura 5.6: Visualizzazione del segnale acquisito dall'applicazione di test tramite *jScope*.

6 Conclusioni

Il lavoro di tesi ha portato alla definizione di un'architettura per la nuova interfaccia LabVIEW di MDSplus che permetta di utilizzare tutte le funzionalità del sistema nell'ambiente di sviluppo grafico di National Instruments. Il *mapping* tra classi e metodi C++ e classi e *VI* LabVIEW è stato già applicato a buona parte delle funzioni del sistema e i test effettuati hanno dato risultati positivi, a conferma della correttezza delle scelte architetturali messe in atto. Il percorso che ha portato alla definizione dell'architettura presentata è risultato tutt'altro che semplice: è stato infatti necessario approfondire i concetti di programmazione orientata agli oggetti nelle diverse declinazioni C++ e *LVOOP* e ricreare in LabVIEW la stessa filosofia di utilizzo del sistema nella sua versione C++. L'aspetto più delicato e allo stesso tempo più interessante del *mapping* è stato sicuramente il processo di integrazione della sintassi *by-reference*, tipicamente utilizzata per la gestione degli oggetti in C++, in un modello di programmazione *dataflow*, peculiarità di LabVIEW.

6.1 Sviluppi futuri

La versione corrente dell'interfaccia è ora in utilizzo presso IGI per un'applicazione di acquisizione di dati termici relativi all'esperimento RFX. In futuro verrà sicuramente completata l'implementazione delle parti mancanti e verranno effettuati nuovi test prima del rilascio definitivo. La versione *Beta* della nuova interfaccia sarà presto disponibile, unitamente alla relativa documentazione di utilizzo e di installazione, sul sito `www.mdsplus.org`. Questo lavoro di tesi costituirà inoltre la base per un articolo scientifico che sarà presentato alla ventisettesima conferenza *Symposium On Fusion Technology* di Liegi per poi essere sottoposto alla rivista *Fusion Engineering and Design* per la pubblicazione.

6.2 Considerazioni sul tirocinio

Questo lavoro di tesi mi ha permesso di mettere alla prova le mie conoscenze e capacità per cercare di trovare una soluzione valida ad un problema reale non banale. La collaborazione con il Prof. Gabriele Manduchi e con le altre persone interessate al progetto è stata molto positiva e mi ha permesso di entrare in contatto con il mondo della ricerca del CNR, dove ho trovato un ambiente molto stimolante dal punto di vista scientifico e umano. Determinante per la buona riuscita del lavoro di tesi è stato anche il periodo di formazione iniziale, voluto dall'Ing. Augusto Mandelli, e il supporto ricevuto da National Instruments Italy.

6.3 Ringrazimenti

Per l'Università degli Studi di Padova ringrazio il Prof. Michele Moro, mio relatore, per la disponibilità dimostrata e l'interesse con cui ha seguito le varie fasi del lavoro. Per l'IGI ringrazio il Prof. Gabriele Manduchi, responsabile del progetto per il CNR e mio correlatore, per la grande disponibilità e competenza, nonché per la fiducia dimostrata nell'assegnarmi questo progetto; ringrazio inoltre Diego Ravarotto per la collaborazione. Per National Instruments Italy ringrazio l'Ing. Augusto Mandelli per la fiducia dimostrata nel sottoscrivere il mio progetto di tesi e per aver promosso i corsi di formazione che ho seguito; ringrazio inoltre l'Ing. Guido Claudio, l'Ing. Luigi Tremolada, l'Ing. Alex Lollio, l'Ing. Matteo Calcagno.

Un ringraziamento speciale alla mia famiglia per avermi sempre sostenuto e incoraggiato: Luigi, Anna e Federica. Ringrazio infine tutti i miei compagni di corso, i miei amici e i miei coinquilini.

Bibliografia

[MDS] J. Stillerman, T. Fredian, *The MDSplus data acquisition system, current status and future directions*, Fusion Engineering and Design 43 (1999) 301-308.

[MLP] T. Fredian, J. Stillerman, G. Manduchi, *MDSplus extensions for long pulse experiments*, Fusion Engineering and Design 83 (2008) 317-320.

[MOO] G. Manduchi, T. Fredian, J. Stillerman, *A new object-oriented interface to MDSplus*, Fusion Engineering and Design 85 (2010) 564-567.

[1] Pagina wiki su LabVIEW:
http://en.wikipedia.org/wiki/Labview

[2] Pagina wiki su LVOOP:
http://labviewwiki.org/LVOOP

[3] Documentazione su RFX:
https://www.consorziorfx.eu/www/sites/default/files/opuscolo_web.pdf

[4] Pagina wiki introduttiva su MDSplus:
http://mdsplus.org/index.php/Introduction

[5] Documentazione su IGI:
http://www.pd.cnr.it/index.php/it/informazioni/istituti/27-igi-it

[6] Documentazione su National Instruments:
http://www.ni.com/company/i/

[7] Pagina wiki su National Instruments:
http://en.wikipedia.org/wiki/National_Instruments

[8] Documentazione su LabWindows/CVI:
http://www.ni.com/lwcvi/i/

[9] Pagina wiki su MDSplus:
http://www.mdsplus.org/index.php?title=Introduction:Approach&open=258312063107164064975 9&page=Introduction%2FApproach

[10] Documentazione sul paradigma di programmazione *data-driven*:
http://www.faqs.org/docs/artu/ch09s01.html

[11] Pagina wiki sull'acronimo *API*:
http://en.wikipedia.org/wiki/Application_programming_interface

[12] Pagina wiki su MDSplus:
http://www.mdsplus.org/index.php?title=Introduction:Capability&open=258312063107164064975 9&page=Introduction%2FCapabilities+and+Concepts

[13] Pagina wiki su MDSplus:
http://www.mdsplus.org/index.php?title=Introduction:Platforms&open=258312063107164064975 9&page=Introduction%2FPlatforms+and+Language

[14] Pagina wiki su AIX:
http://en.wikipedia.org/wiki/IBM_AIX

[15] Pagina wiki su Tru64 Unix:
http://en.wikipedia.org/wiki/Tru64_UNIX

[16] Pagina wiki su HP-UX:
http://en.wikipedia.org/wiki/HP-UX

[17] Pagina wiki su IRIX:
http://en.wikipedia.org/wiki/Irix

[18] Pagina wiki su OpenVMS:
http://en.wikipedia.org/wiki/Openvms

[19] Pagina wiki su SunOs:
http://en.wikipedia.org/wiki/SunOS

[20] Pagina wiki su IDL:
http://en.wikipedia.org/wiki/IDL_(programming_language)

[21] Pagina wiki sull'implementazione orientata agli oggetti di MDSplus:
http://www.mdsplus.org/index.php?title=Documentation:Tutorial:MdsObjects&open=258362503862801989 6319&page=Documentation%2FThe+MDSplus+tutorial%2FThe+Object+Oriented+interface+of+MDSPlus

[22] Pagina wiki su MDSplus:
http://www.mdsplus.org/index.php?title=Documentation:Tutorial:CreateTrees%20&open=258362503862801989 6319&page=Documentation%2FThe+MDSplus+tutorial%2F+Creating+and+populating+MDSplus+trees

[23] Documentazione su *jScope*:
http://www.mdsplus.org/documentation/tutorial/jScope.html

[24] Documentazione su *LVOOP*:
http://www.ni.com/white-paper/3574/en

[25] FAQ su *LVOOP*:
http://www.ni.com/white-paper/3573/en

[26] Documentazione su *LVOOP*:
http://zone.ni.com/reference/en-XX/help/371361F-01/lvconcepts/creating_classes/

[27] Documentazione su *LVOOP*:
http://zone.ni.com/reference/en-XX/help/371361H-01/lvconcepts/using_classes/

[28] Pagina wiki sui *template* in C++:
http://en.wikipedia.org/wiki/Template_(C%2B%2B)

[29] Pagina wiki sulle funzioni *pure virtual*:
http://en.wikipedia.org/wiki/Pure_virtual_method

[30] Pagina wiki sull'ereditarietà multipla:
http://en.wikipedia.org/wiki/Multiple_inheritance

[31] Documentazione su *Call Library Function Node*:
http://zone.ni.com/reference/en-XX/help/371361H-01/lvexcodeconcepts/configuring_the_clf_node/

[32] Documentazione su *Call Library Function Node*:
http://zone.ni.com/reference/en-XX/help/371361H-01/glang/call_library_function/

[33] Documentazione su *Call Library Function Node*:
http://www.ni.com/white-paper/3009/en

[34] Guida sull'utilizzo di codice esterno in LabVIEW:
http://www.ni.com/pdf/manuals/370109b.pdf

[35] Documentazione sull'utilizzo di *DVR*:
http://www.ni.com/white-paper/9386/en

[36] Pagina wiki su VxWorks:
http://en.wikipedia.org/wiki/VxWorks

Elenco degli algoritmi

Elenco delle figure

Elenco delle tabelle