POLITECNICO DI BARI

Dipartimento di Ingegneria Elettrica e dell'Informazione

Corso di Laurea in
Ingegneria Informatica Magistrale - Information Systems

Tesi di laurea in Advanced Software Engineering

Validazione dei Risultati di Test per Attività in Sala Prova Motore con Algoritmi di Intelligenza Artificiale

Relatore:
Prof.ssa Ing. Marina Mongiello

Correlatore:
Ing. Vincenzo Ficarella

Laureando:
Alberto Giulio Cassano

Anno Accademico: 2018-2019

Indice

Indice Immagini

Indice Tabelle

Introduzione

Il mondo dei propulsori è un'ampia sfera che abbraccia svariate discipline scientifiche ed è fortemente condizionato dall'evoluzione tecnologica che scandisce il progresso della nostra civiltà. Con il susseguirsi di nuove scoperte, spinte da ricerche che esplorano i limiti della realtà che conosciamo, si aprono nuove strade e si istituiscono nuovi metodi per migliorare gli strumenti di cui già disponiamo, con i quali è possibile tendere a orizzonti prima irraggiungibili. In tal senso, anche la realtà dei motori a combustione interna è in continua trasformazione, grazie all'utilizzo di combustibili alternativi, tecniche innovative e nuovi materiali. Su questi aspetti è decisamente molto sensibile l'azienda **FPT Industrial** (**Fiat Powertrain Technologies**) del gruppo **CNH Industrial**, di cui faccio parte, che si è affermata come punto di riferimento nel mercato dei propulsori per applicazioni veicolari industriali on-road, off-road, marine e power generation. In modo particolare lo studio di nuove soluzioni che tutelino l'ambiente, senza rinunciare a buone prestazioni e affidabilità del prodotto, è prerogativa dei centri R&D di FPT Industrial, sparsi nel mondo. É proprio in uno di questi *Testing Centre*, situato a Foggia, che si conduce una buona fetta della attività di ricerca sui motori alimentati da NG (Gas Naturale, che nel linguaggio comune è semplicemente chiamato Metano), che costituisce una delle sfide cruciali del mercato dei propulsori in questo particolare periodo storico ed è in questa realtà pugliese che svolgo la mia attività lavorativa. I ritmi di evoluzione delle tecnologie, tuttavia, sembrerebbero più lenti rispetto a quelli del preoccupante deterioramento del nostro ecosistema, a cui stiamo tragicamente assistendo con maggiore enfasi negli ultimi anni. Per questa ragione è indispensabile concentrare il più possibile gli sforzi perché i risultati dei test condotti nelle nostre **sale prova motore** siano allo stesso tempo tempestivi e affidabili.
A tale scopo è stato concepito questo progetto, che nasce dall'intenzione di validare i risultati dei test, approcciando ai dati raccolti in quanto tali. Spesso le acquisizioni fatte in sala prova nascondono molte informazioni di difficile accesso, dal momento che sono disperse in un quantitativo molto grande di dati. Nell'ottica di semplificare il lavoro di interpretazione o di condurre un'operazione di **Data Mining**, può essere utile usare algoritmi di **Machine Learning**. Quest'idea è stata applicata alla necessità di un immediato controllo di plausibilità sui risultati dei test: **lo scopo di questo progetto è l'implementazione del-**

lo strumento di validazione AITV (**A**rtificial **I**ntelligence **T**est **V**alidator), basato su un modello predittivo con **Intelligenza Artificiale**, che ha il fine di validare la consistenza dei risultati raccolti. **AITV** apprende su un *training set* costituito dai dati relativi ai test effettuati, sistematicamente raccolti. Essi consistono di parametri generali di input, che possono caratterizzare e influenzare il test, e di risultati calcolati in fase di post-elaborazione. Il modello così ottenuto è in grado di stimare il risultato di una nuova prova a partire dai parametri di input, permettendo così di confrontare la predizione con il risultato realmente calcolato dalla post-elaborazione e l'esito di questo confronto può quindi contribuire a determinare la validità del risultato ottenuto.
Per rendere possibile un'operazione del genere, è necessario dunque realizzare un ampio sistema che abbracci e coinvolga l'intera piattaforma di test, dall'**automazione di sala prova** al **post-processing dei dati**, passando per le **aree di archiviazione**. Realizzare un'architettura del genere è il solo modo per fruire delle potenzialità di AITV, che vanno ad aggiungersi a diverse altre tecniche di validazione comunemente adoperate per i risultati delle attività di ricerca in sala prova.
Nel primo capitolo di questo elaborato è offerta una breve panoramica dei concetti fondamentali che regolano l'attività di test sui motori e degli strumenti di cui ci si è serviti per realizzare questo progetto. Il secondo capitolo introduce invece l'architettura del sistema creato e illustra alcuni dei concetti che saranno maggiormente approfonditi nel successivo terzo capitolo, dedicato alla dettagliata descrizione tecnica della struttura implementata. Il quarto capitolo è infine dedicato ad un caso di studio che concerne il *Lambda Step*, una tipologia di test effettuato in sala prova, tramite il quale è valutata la *Oxygen Storage Capacity* (OSC) dei catalizzatori in prova. Nell'ultima sezione, dedicata alle conclusioni, saranno brevemente discussi i risultati di questo progetto e saranno dettagliate diverse idee per futuri sviluppi.
In conclusione, lo strumento proposto in questa tesi è progettato per offrire un supporto agli ingegneri responsabili della valutazione dei risultati ottenuti dalle attività di ricerca svolte, senza pretendere di sostituire le preesistenti tecniche di validazione basate su studi dei fenomeni chimici e fisici che regolano emissioni, consumi e prestazioni dei nostri motori. Lo spettro d'applicazione di questo sistema è potenzialmente ampio e sebbene l'interesse primario sia ottenere una sempre maggiore confidenza nei risultati dei test, il suo impiego potrebbe fare luce su correlazioni finora sottovalutate tra grandezze protagoniste delle nostre ricerche.

Capitolo 1

Attività R&D in Sala Prova Motori

Dal 26/04/2017 occupo la posizione di Automation Engineer presso il Testing Centre FPT di Foggia[1]. La mia azienda, FPT (FIAT Powertrain Technologies) Industrial, è un brand del gruppo CNH Industrial, che progetta e produce motopropulsori per applicazioni *on-road*, *off-road*, *marine* e *power generation*. La posizione che ricopro trova collocamento nel ramo d'ingegneria[2], responsabile di progettazione e test dei motori, dalla fase prototipale alle prove su veicolo.
Tali attività di ricerca e sviluppo - finalizzate per esempio ad incrementare le prestazioni o a ridurre emissioni e consumi o a provare l'affidabilità nel tempo di nuovi componenti - sono condotte principalmente in appositi laboratori, noti come *sale prova motore*, dotati di strumentazione di misura avanzata e serviti da numerosi e complessi impianti.
Il Testing Center di Foggia, uno dei 6 poli R&D di FPT Industrial distribuiti nel mondo, è ubicato all'interno di un ampio stabilimento di produzione FPT e sebbene sia piccolo riveste attualmente un ruolo di particolare rilievo: è principalmente a Foggia infatti che sono sviluppati - e omologati - i motori **NG**[3] di FPT, un'applicazione di cruciale importanza nel mercato odierno, segnato da una stigmatizzazione dei motori a combustione interna tradizionali.

1.1 FPT Industrial

La società, nata nel 2011 da una scissione parziale di Fiat Group, è considerato uno dei principali attori nel mercato dei motopropulsori agricoli, industriali e marini: produce motori, cambi e trasmissioni e vende non solo ai brand dello

[1] Loc. Incoronata.
[2] In particolare nella divisione **Product Engineering - Engineering Services & Prototype**.
[3] Natural Gas, combustibile composto principalmente da metano (CH4)

Figura 1.1: Logo di FPT Industrial.

stesso gruppo CNH Industrial[4], ma anche ad altre aziende esterne. Un altro settore di spicco per i prodotti FPT è costituito dalle competizioni sportive marine, in modo particolare la disciplina **offshore**, che prende il nome proprio dall'omonimo tipo di imbarcazione, in grado di raggiungere elevate velocità.
Proponendo una breve descrizione ristretta al settore in cui è coinvolto il centro R&D di Foggia, la gamma di motori prodotti da FPT spazia tra taglia leggera, media e pesante e sono diverse le tecnologie all'avanguardia sviluppate e utilizzate per questi propulsori. Le applicazioni sono principalmente:

- **On-road**, che fa riferimento ai trasporti su strada e prevede l'impiego delle famiglie **F1**, **NEF** e **CURSOR**, con una cilindrata che va da 2.3 a 12.9 litri, power rate da 97 a 570 cavalli e coppia massima da 240 a 2500 Nm. Comprende le seguenti tipologie di veicolo:
  - LCV (Light Commercial Vehicles)
  - Trucks
  - Buses
- **Agricolture**, che prevede diverse soluzioni per ogni applicazione agricola, dai trattori alle macchine di mietitura e raccolta.
- **Construction**, che impiega 5 famiglie di motori e prevede una grande flessibilità, adattandosi a escavatori, pale caricatrici, pale compatte, motolivellatrici, dozers, carrelli elevatori, spazzaneve e gru.

Concentrando inoltre l'attenzione sulle tecnologie Alternative Fuels, i motori Natural Gas di FPT presentano emissioni di inquinanti nocivi estremamente basse, in modo particolare riducono notevolmente le emissioni di CO2, importanti responsabili del riscaldamento globale. Questo *eco-benefit*, insieme all'abbattimento della rumorosità dei motori e a costi del carburante molto bassi, fanno

[4]Iveco, Iveco Bus, Case Construction Equipment, New Holland Agricolture, solo per citarne alcuni.

di questi propulsori una valida alternativa al Diesel per veicoli leggeri, medi e pesanti, da diversi punti di vista:

- **Prestazioni**: questi motori hanno derivazione Diesel e possono garantire performance confrontabili (range di power rate da 136 a 400 cavalli). Grazie alle tecniche impiegate, garantiscono una combustione stabile e una risposta immediata nei transitori.

- **Efficienza**: il risparmio sul carburante rispetto al Diesel è stimato intorno al 30% ed è massimizzata la flessibilità in un'ampia gamma di qualità del gas.

- **Affidabilità**: l'esperienza di FPT nel Natural Gas è ventennale e può vantare il primissimo impiego della tecnologia stechiometrica, divenuta poi uno standard. É previsto per questi motori un cambio olio ogni 75000 km (best in class).

- **Sistema After-Treatment**: non è necessario l'impiego di EGR, SCR o DPF per ottenere emissioni al di sotto dei limiti di normativa, perciò è previsto l'impiego di un semplice *3-way catalyst*, rendendo estremamente competitivo il TCO (Total Cost of Ownership) dei prodotti Natural Gas.

Infine va specificato che questi motori possono essere alimentati dal combustibile Natural Gas nelle forme Compressa (CNG), Liquida (LNG) e rinnovabile (biometano), quest'ultimo in particolare costituisce la migliore alternativa *"green"*, in grado di portare le emissioni di CO2 a valori più bassi addirittura del 100% rispetto al Diesel.
Come anticipato nel paragrafo precedente, nel centro R&D di Foggia sono condotte attività di ricerca e sviluppo esclusivamente su motori NG: questo è uno stimolo enorme per me e per i miei colleghi, per via delle numerose innovazioni messe in campo e per la rilevanza cruciale del Natural Gas nella promozione del rispetto dell'ambiente.

1.2 FPT Testing Foggia - R&D Center

Il centro Testing FPT di Foggia ospita 6 sale prova motore ed un banco a rulli, che rappresentano i due principali tipi di laboratorio per prove motoristiche:

- Nelle sale prova il motore è installato in una configurazione il più possibile vicina a quella veicolare, ma allo stesso tempo controllato con sistemi di attuazione tali da ottenere nei test una ripetibilità, in termini di condizioni dell'ambiente e del motore, non facilmente garantibile altrimenti. Anche le misurazioni effettuate in sala prova, direttamente su componenti spesso prototipali e strumentati in modo altamente specifico, sono difficilmente

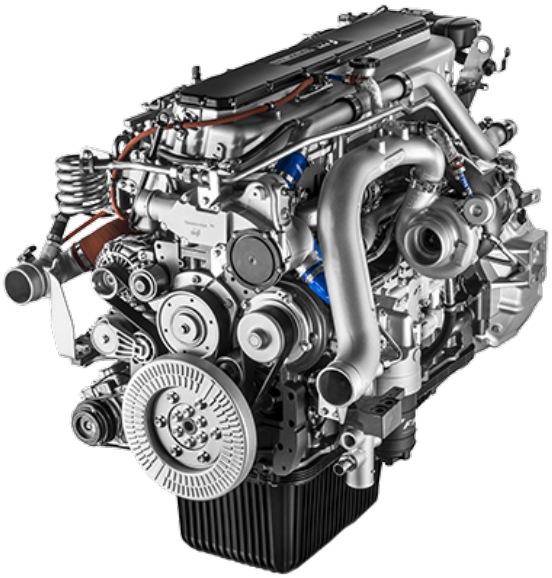

Figura 1.2: Cursor 13 CNG di FPT Industrial.

ottenibili su un motore direttamente montato sul veicolo. La stessa strumentazione adottata in sala prova, dotata di precisione e affidabilità molto elevate, è impossibile da installare su veicolo, per dimensioni e condizioni di funzionamento[5].

- Il banco a rulli è un particolare laboratorio in cui è testato l'intero veicolo, grazie ad un rullo posto in rotazione dalle ruote motrici. É così possibile l'utilizzo praticamente della stessa strumentazione adoperata in sala prova ed è garantito un vantaggio di ripetibilità, fatta eccezione per la variabilità nella guida del veicolo strettamente condizionata dal *driver*[6] incaricato del test.

Il personale che svolge attività lavorativa in questo centro è suddiviso in due team:

- PWT PE EE - CNG P&E BASE CALIBRATION (CNG Performance & Emissions Base Calibration), ovvero il team *application*, responsabile delle attività condotte sul motore e che dunque stabilisce e commissiona i test da effettuare nelle sala prova motore, con lo scopo di sviluppare il prodotto e validarne consumi, emissioni, prestazioni e affidabilità.

- PWT PE - ES&P Foggia (Testing Operations & Prototype - Foggia Site), ovvero il team *facilities*. Si occupa della gestione dei laboratori sia dal

[5]Va fatta menzione comunque di strumenti come AVL PEMS [11], utilizzato per prove su veicolo direttamente su strada (RDE, Real Driving Emissions)

[6]Si tratta comunque di personale qualificato ed esperto e le prove prevedono un numero massimo di errori consentiti, in riferimento alla traccia prefissata.

punto di vista degli impianti sia da quello degli strumenti di misura e attuazione, della loro manutenzione, ma soprattutto è responsabile della realizzazione delle prove commissionate dal team application e fornisce loro strumenti d'analisi delle prove. Questo team è suddiviso in due componenti, una dedicata alle sale prova - di cui faccio parte - e una al banco a rulli. Inoltre a questo gruppo afferiscono gli operai specializzati, le cui mansioni complesse e variegate costituiscono un aspetto fondamentale della vita del centro.

Con particolare riferimento alle sale prova, alle quali è dedicata la mia figura, è proposta brevemente un'introduzione alle due principali attività che mi competono: la programmazione di test automatici (e più in generale la gestione dell'**automazione** di sala prova) e la **post-elaborazione** dei dati raccolti. Per questi compiti mi è risultata indispensabile la preparazione assunta con la laurea triennale in Ingegneria Informatica e dell'Automazione, così come tutti gli esami svolti in questo corso di Ingegneria Informatica Magistrale.

1.2.1 Automazione di sala prova

All'interno della sala prova, il motore è fissato su un supporto apposito (spesso realizzato su misura), il quale è adagiato su una base sismica. All'albero motore è collegato per mezzo di un giunto un **freno dinamometrico elettrico** a correnti parassite (noto come *freno stazionario*), in grado di applicare al motore in prova una coppia resistente, o in alternativa un **motore elettrico** con inverter (noto come *freno dinamico*), in grado di fornire una coppia motrice (trascinando il motore) o resistente (lavorando di fatto come un freno a correnti parassite). Queste due modalità di funzionamento del freno sono solitamente indicate con la seguente denominazione:

- Modalità **Passiva**: consiste nell'applicazione esclusivamente di coppia resistente ed è di solito utilizzata per impostare un dato valore limite di velocità[7], infatti si indica spesso anche come "modalità *tetto di giri*". In questo modo è possibile lavorare a giri costanti, al variare del carico (e quindi della coppia fornita dal motore in prova).

- Modalità **Attiva**: permette di applicare sia coppia motrice che resistente ed è di solito utilizzata per garantire una determinata velocità, anche trascinando il motore in prova quando esso non fornisce coppia. Questo avviene durante il **cut-off**, cioè il taglio d'iniezione, una circostanza molto frequente durante il normale funzionamento dei propulsori quando sono montati su veicolo. Pilotare il motore con un banco in modalità attiva permette quindi di simulare nella maniera migliore possibile il reale comportamento

[7] Per velocità si intende la velocità angolare del motore in prova, misurata in rpm. É nota quindi anche con il nome di **giri**.

Figura 1.3: Freno a correnti parassite Apicom.

Figura 1.4: Sala prova dinamica (con motore AC) Apicom.

Figura 1.5: Cella di carico Revere Transducers 9363.

su strada dei motori, caratterizzato da transitori con variazioni rapide di velocità e coppia erogata.

La dualità di utilizzo garantita dai freni dinamici, li rende di fatto più preziosi e adatti ad attività di test di livello avanzato.
Alla **UUT** (Unit Under Test) sono connessi vari sensori in grado di misurare le principali grandezze d'interesse durante i test, ad esempio:

- La **coppia** erogata dal motore, mediante celle di carico o flange torsiometriche, queste ultime calettate al rotore del freno dinamico;
- I **giri** o la **velocità** del motore, mediante encoder o pick-up magnetici;
- Le varie **pressioni**, mediante trasduttori piezoresistivi;
- Le varie **temperature**, mediante termocoppie (solitamente di tipo K) o termoresistenze (solitamente PT100 in platino).

Inoltre sono installati in sala prova numerosi **strumenti di misura**, cioè sofisticati sistemi di acquisizione in grado di effettuare misure molto complesse con un'elevata precisione e velocità. Sono riportati di seguito alcuni esempi.

- Il **banco analisi emissioni** misura in tempo reale la concentrazione degli inquinanti nel gas di scarico del motore. Le sale prova del centro Testing FPT di Foggia sono equipaggiate con gli **AVL AMA i60**, che sono dotati di diversi **analizzatori** raggruppati in uno o due treni, ciascuno dei quali analizza il gas catturato in un qualsiasi punto di prelievo sulla linea di scarico. Gli analizzatori presenti in un banco AMA i60 sono i seguenti:

Figura 1.6: Torsiometro HBM T40B.

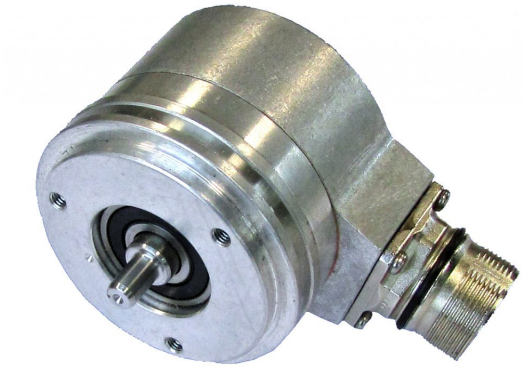

Figura 1.7: Encoder Heidenhain ROD 430.

Figura 1.8: Pickup magnetico Red Lion MP-62TA.

Figura 1.9: Trasduttore di pressione GE Sensing PTX 1000.

Figura 1.10: Termocoppia di tipo K ItalCoppie TCK.

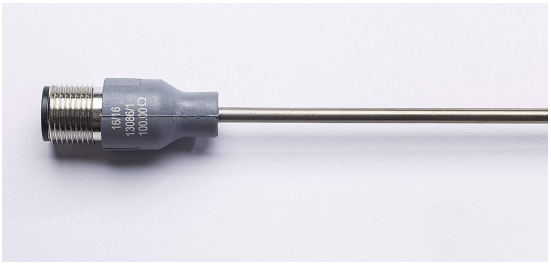

Figura 1.11: Termoresistenza PT100 ItalCoppie PT100.

Figura 1.12: Banco Analisi AVL AMA i60.

  - NDIR, analizzatore non dispersivo a raggi infrarossi. Misura monossido di carbonio (CO) e anidride carbonica (CO2).
  - HFID, analizzatore a ionizzazione di fiamma con rivelatore, valvole e condotti riscaldati (a 463 K ca.). Misura idrocarburi totali (THC).
  - CLD, analizzatore di tipo a chemiluminescenza con convertitore NOx / NO. Misura ossidi di azoto (NOx).
  - FID, analizzatore a ionizzazione di fiamma con dispositivo di eliminazione di idrocarburi non metanici (cutter). Misura metano (CH4).

- Separatamente viene misurata la concentrazione di ammoniaca (NH3), un altro inquinante nocivo, sottoposto a limite di normativa, che si forma a valle dei catalizzatori. Lo strumento tipicamente utilizzato per questa misura è **AVL SESAM FTIR**, che basa l'analisi sulla spettroscopia agli infrarossi.

- I sistemi di **indicating** permettono di condurre un'approfondita analisi della combustione effettuata dal motore, ciclo per ciclo. Nel nostro centro è utilizzato il sistema **AVL IndiCom**, che misura e calcola tutte le grandezze d'interesse con una velocità elevatissima. Le varie generazioni di questo sistema prevedono hardware di diverso tipo (Indimodul 622 o X-ion).

Figura 1.13: Banco Analisi AVL SESAM FTIR.

Figura 1.14: HW per acquisizone Indicating AVL Indimodul 622.

Figura 1.15: HW per acquisizone Indicating AVL X-ion.

Alla UUT è collegato inoltre il sistema after-treatment e sono naturalmente presenti anche gli scambiatori e gli impianti di raffreddamento, la tubazione che approvvigiona il combustibile (gas di rete proveniente dalla linea o gas certificato stoccato in apposite bombole) e tutti i sistemi di sicurezza in caso di incendi o esplosioni, che non si possono escludere in un contesto di ricerca su motori a gas naturale compresso.
Infine è collegata al cablaggio motore una "centralina" o, più propriamente, una **ECU** (Engine Control Unit) adatta allo sviluppo: differentemente dalle ECU di produzione, questi dispositivi prevedono una particolare interfaccia che permette di accedere e modificare in modalità real-time parametri e mappe della ECU. Per gestire questa interfaccia è utilizzato un software apposito, tipicamente è adoperato ETAS INCA [5].
Per coordinare un banco prova così strutturato è necessario un potente **sistema di automazione**, che faccia da collettore di tutta la strumentazione di misura e che sia in grado di manovrare programmaticamente il motore e l'intera sala prova. Si tratta di sistemi suddivisi generalmente in una componente di più basso livello (dotata di HW dedicato, tipicamente installata su SO di tipo Real Time) e in una componente di alto livello (installata su un classico SO Windows). La prima garantisce brevissimi tempi di input e output (talvolta nell'ordine di 1 ms), mentre la seconda gestisce l'interfaccia utente. Nel centro di Foggia sono adoperati due prodotti di questo genere:

- **EURINS AdaMo**[8], basato su tecnologia HW e SW National Instruments[9], è adoperato in 4 sale prova. La sua architettura, interamente realizzata in linguaggio LabVIEW [4], si basa su un'applicazione Windows connessa ad un CompactRIO[10] ed è sviluppata su 3 livelli:
  - un dispositivo **FPGA**, integrato nel cRIO, che gestisce input e output a basso livello;
  - un software **Real Time** installato sul SO LinuxRT [15] del cRIO, che rappresenta il vero nucleo di AdaMo: gestisce tutte le schede NI (9264, 9205, 9219) a bordo del cRIO;
  - un client **Windows** installato su un classico PC Desktop, che costituisce l'interfaccia utente e - attualmente - l'ambiente di programmazione per task e prove automatiche.

  I prodotti National Instruments sono un riferimento nel merito della misura e del controllo, ma un'ulteriore e ancor più importante qualità del sistema AdaMo è la **modularità** hardware e software, che lo rende uno strumento molto valido per attività di ricerca, non solo in ambito automotive.

[8] `www.eurins.com`
[9] `www.ni.com`
[10] `www.ni.com/it-it/shop/compactrio.html`.

Permette infatti un alto grado di personalizzazione dell'applicazione per il cliente, grazie all'integrazione di altri software (senza vincoli sul linguaggio utilizzato) e grazie anche alla possibilità di utilizzare qualsiasi scheda NI compatibile con il cRIO.

- **AVL PUMA Open** [1], presente in 2 sale prova nella versione 1.5.1, è considerato lo standard mondiale per i banchi prova motore ed è molto facilmente integrabile con tutta la strumentazione prodotta dalla stessa AVL, la più largamente utilizzata in questo campo. Sono citati di seguito alcuni esempi di strumenti prodotti da AVL, utilizzati nel centro Testing FPT di Foggia:

  - AVL IndiCom per il controllo combustione
  - AVL AMA i60, AMA i60 LDD e AVL SESAM FTIR per l'analisi emissioni
  - AVL 489 APC e AVL 478/472 SPC per misurare rispettivamente il numero di particelle emesse (Particulate Number, PN) e il loro peso (Particulate Matter, PM)

  PUMA Open è una suite software che offre una serie di strumenti per la parametrizzazione, il monitoraggio e il controllo dei banchi prova motore. É dotato di un'interfaccia per gli operatori che prende il nome di PUMA Operator Interface (POI) che mette a disposizone tutte le funzionalità del sistema d'automazione, in base all'implementazione effettuata da parte dell'automation engineer. Il sistema operativo real time di PUMA prende il nome di **INtime** e si occupa della gestione a basso livello del banco prova e anche del coordinamento di alcune categorie di strumenti, tra cui i banchi analisi. Infine un banco PUMA può essere dotato della piattaforma di certificazione (AVL iGEM [8]), che costituisce il principale riferimento per l'esecuzione e la post-elaborazione delle prove di certificazione dei motori.

Questi sistemi permettono dunque, in tempo reale, di pilotare il motore e analizzarne la risposta nei termini delle grandezze misurate, che possono essere mostrate a video o scritte su un file di salvataggio. Il controllo del motore è effettuato dal banco per mezzo dei seguenti output:

- Set di giri trasmesso al freno (modalità passiva) o al motore elettrico (modalità passiva o attiva).
- Set di pedale (alpha), trasmesso alla ECU come segnale analogico in tensione. Consiste nel nostro caso in una richiesta di coppia al motore, espressa come percentuale di pedale in base alla coppia massima raggiungibile dal motore alla velocità attuale.

- Set mappe o parametri della ECU, attraverso un protocollo di comunicazione (tipicamente ASAP3 [7]) tra il software di automazione e il software di gestione ECU. É quest'ultimo a trasmettere infine il set alla ECU.

Il set di giri e alpha (o coppia) può essere effettuato in modo stazionario, mantenendo cioè il motore stabile nel *punto operativo* richiesto, oppure possono essere effettuati dei transitori (gradini o rampe). Il software di banco permette inoltre di coordinare valvole, regolatori e altri dispositivi di condizionamento dell'ambiente all'interno della sala prova.
Un altro compito fondamentale del sistema di automazione consiste nell'integrazione dei vari strumenti di misura adoperati, ciascuno dei quali espone metodi di controllo e dati misurati attraverso uno specifico protocollo (tipicamente AK Protocol [13]). Attraverso l'implementazione di opportuni *driver* è possibile trasmettere agli strumenti comandi (misura, standby, reset, pulizia, etc.) e ricevere i valori misurati con una frequenza dipendente dallo strumento). Ciascuna di queste operazioni può essere effettuata tramite meccanismi manuali o alternativamente per mezzo dell'interfaccia utente del software, che l'*automation engineer* realizza in base alle proprie necessità. Le tecniche di salvataggio infine prevedono l'utilizzo di file di testo o record di database strutturati (solitamente ASAM ODS [3][9]).
É possibile considerare dunque il SW d'automazione come un ampio ambiente di programmazione, in cui le variabili (tipicamente è adoperata una tipizzazione forte) sono note come **canali** e possono avere diversa natura:

- **input** attribuibili al valore attuale di una grandezza fisica (ottenuto da un sensore o da uno strumento di misura) o immessi dall'operatore durante l'utilizzo del banco
- **costanti** prefissate e definite staticamente
- **formule** implementabili con metodi di calcolo avanzati, per ottenere grandezze non direttamente misurabili
- **output** associati ad uscite fisiche del sistema o variabili utilizzate come contatori, buffer, etc.

Inoltre possono essere configurati ed eseguiti **task** (controllo del motore, gestione strumenti, set di output, scrittura su file), **automi** a stati finiti, allarmi e relative reazioni. AVL PUMA permette inoltre l'esecuzione di VBScript per realizzare strumenti software più complessi e strutturati.
L'impiego di questi sistemi tuttavia non si limita ad un utilizzo "manuale" del banco, che richiede la presenza di personale qualificato, ma permette l'implementazione di **test automatici** che il SW esegue appunto in autonomia. La programmazione di queste prove è realizzata secondo le tecniche ed i linguaggi

(ad esempio VB[11]) che i diversi sistemi d'automazione mettono a disposizione, tuttavia la disciplina e gli algoritmi da adoperare sono a totale discrezione dell'utilizzatore: è infatti cruciale riuscire a programmare test in grado di restituire lo stesso risultato per formato e contenuto, a prescindere dal software di automazione utilizzato.
Le prove automatiche permettono inoltre di effettuare sul motore operazioni deterministicamente ripetibili, in maniera reiterata. Questo risulta fondamentale per attività di **durata**, che consistono nel replicare per un gran numero di volte lo stesso *profilo* di giri e coppia (o alpha) con lo scopo di validare l'affidabilità del prodotto o di determinate sue componenti. Tuttavia nel nostro centro sono svolte più frequentemente prove di **sviluppo**, spesso finalizzate a raggiungere la calibrazione (ovvero la configurazione dei parametri della ECU) che permetta di ottenere specifici risultati preposti, tipicamente in termini di emissioni, consumi o prestazioni. Se da un lato le prove manuali costituiscono sempre il punto di partenza per qualsiasi sperimentazione motoristica di questo genere, dall'altro lato i test automatici possono garantire, per prove di sviluppo riconducibili a precisi algoritmi, ripetibilità e velocità d'attuazione impossibili da ottenere manualmente. Inoltre garantiscono lo svolgimento di complesse attività di ricerca, anche quando il personale qualificato per svolgerle ha terminato l'orario lavorativo.
Lo scopo delle sperimentazioni condotte consiste quasi esclusivamente nel mantenere le emissioni dei nostri motori al di sotto dei limiti di normativa. Per validare la maturità di un motore e della relativa calibrazione, sono spesso effettuati **cicli omologativi**[12] anche durante le attività di sviluppo. Si tratta di particolari prove automatiche implementate ed eseguite secondo le prescrizioni delle normative vigenti e sono effettuate durante le omologazioni in presenza di personale della Motorizzazione Civile, che certifica la conformità del motore in base appunto all'esito di queste prove. Infine sta assumendo sempre più importanza la simulazione al banco di profili provenienti da acquisizioni su veicolo, che danno un riscontro pratico del quantitativo di inquinanti emessi durante il percorrimento di un normale tratto stradale. L'implementazione e l'esecuzione di queste particolari prove richiede dunque un approccio meticoloso nell'osservazione delle specifiche di normativa.
Generalmente la realizzazione delle prove si snoda nelle seguenti fasi:

1. **Progettazione**: in cooperazione con gli application engineer sono analizzati gli obiettivi dell'attività e le specifiche necessarie, in termini di caratteristiche del banco e disponibilità di eventuali strumenti di misura non standard (in condivisione tra le sale prova). A tale scopo si consultano le Norme (interne FPT o normative di legge) che dettano le linee guida per lo svolgimento delle prove.

[11] `docs.microsoft.com/it-it/dotnet/visual-basic/`

[12] I principali test omologativi per motori On-road sono WHTC (World Harmonized Transient Cycle) e WHSC (World Harmonized Stationary Cycle), descritti dalla normativa ECE-R49 [10].

2. **Gestione degli strumenti**: come già accennato, ciascuno strumento di misura è dotato di una propria interfaccia che permette al SW di banco di acquisire le grandezze misurate e in molti casi consente anche il controllo remoto del device; va curata quindi la configurazione di questi protocolli e la creazione di librerie per la gestione degli strumenti nell'ambiente d'automazione.

3. **Implementazione**: gli algoritmi di test possono essere realizzati come veri e propri programmi, strutturati in sotto-procedure (o *subroutines*) rese il più possibile parametrizzabili dinamicamente e quindi riutilizzabili in altre sequenze di prova. Questo approccio permette di organizzare le procedure in librerie, dalle quali attingere durante l'implementazione di una nuova prova. I diversi step delle sotto-procedure possono consistere in azioni di controllo del motore (statiche o dinamiche), nell'invio di comandi agli strumenti connessi, nell'esecuzione di salvataggi (avvio o interruzione di *recorder* e la scrittura di misurazioni mediate), etc. Non meno importante è la messa a punto dell'interfaccia grafica per gli operatori che dovranno eseguire la prova ed eventualmente immettere alcuni input preliminari.

4. **Test**: prima di deliberare che la prova richiesta sia pronta per essere eseguita, è opportuno verificare il funzionamento - ed eventualmente effettuare un "debug" - del test implementato, simulandone lo svolgimento a motore spento. Si tratta di un'operazione fondamentale per non causare danni su motori e componenti prototipali.

1.2.2 Post-Elaborazione dei dati

La quantità e la complessità dei dati acquisiti durante le prove richiedono un'importante attività di analisi successiva, da condurre per mezzo di strumenti adeguati a tale scopo. Effettuare valutazioni sui risultati dei test comporta infatti una conoscenza profonda delle normative, del funzionamento dei nostri motori e di tutti i fenomeni fisici e chimici associati alla combustione e allo scarico. D'altro canto è contestualmente necessario maneggiare una grande mole di dati, dai quali non è sempre immediato ottenere le informazioni desiderate: per questo motivo è necessario associare alle competenze tecniche del settore, proprie del team application, competenze nell'ambito della programmazione offerte dall'automation engineer del team facilities.
Piattaforme tipiche per effettuare quest'attività di **post-elaborazione** possono essere individuate nei classici fogli di lavoro Microsoft Excel[13], anche grazie alle Macro VBA[14] implementabili, o nell'IDE di Mathworks MATLAB[15]. Il software

[13] `docs.microsoft.com/en-us/office/client-developer/excel/excel-home`
[14] `docs.microsoft.com/it-it/office/vba/api/overview/excel`
[15] urlwww.mathworks.com/products/matlab.html

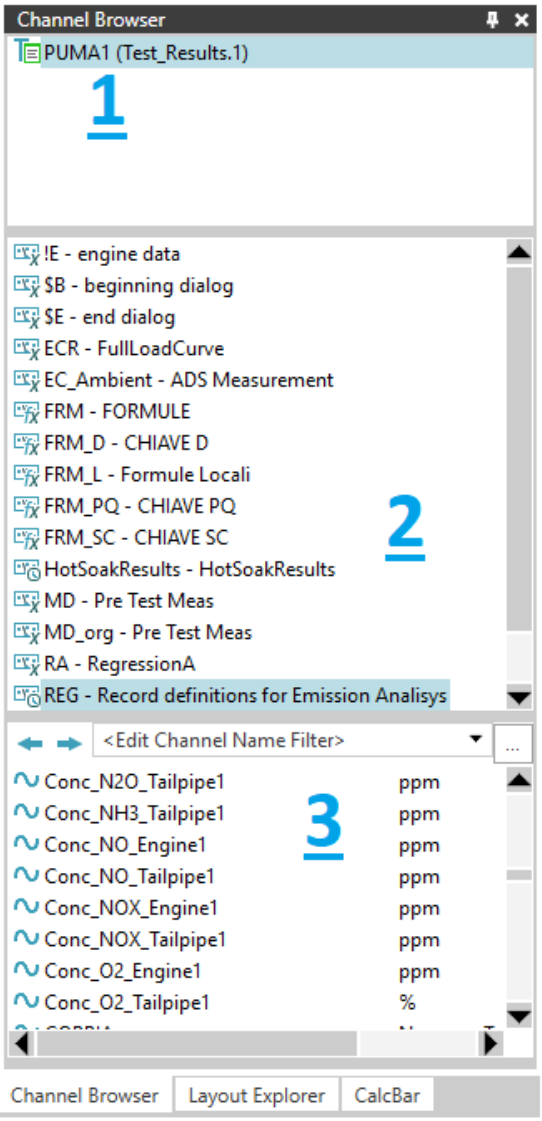

Figura 1.16: Struttura di un File dati in Concerto.

di maggior impiego nella nostra azienda per questo genere di lavoro è tuttavia **AVL Concerto**[16], largamente utilizzato nel settore automotive, per il quale è stato essenzialmente ideato. Permette di connettersi a database remoti, importare ed esportare dati gestendo numerosi formati diversi, realizzare e manipolare molto facilmente grafici di diverse tipologie, effettuare calcoli molto articolati o impossibili da ottenere "online" sul software di automazione di sala prova (ad esempio il calcolo del lavoro durante un ciclo WHTC, espresso come l'integrale della potenza positiva durante il test). Un insieme di grafici - o finestre di altro tipo - prende il nome di **layout**, vale a dire una "maschera" che rappresenta in maniera esplicita le informazioni estratte dai dati contenuti nei *test result*; un esempio tipico può essere rappresentato dal layout applicato ai dati raccolti da un ciclo omologativo, che costituisce il vero e proprio **report** della prova di certificazione.
La grande potenzialità di questa piattaforma è però nell'ambiente di programmazione che mette a disposizione, per realizzare formule più complesse o script che manipolano dinamicamente il layout. In tal senso tecniche e metodologie di programmazione più strutturate permettono la realizzazione di layout adatti a effettuare elaborazioni molto complesse in maniera immediata. Per fare questo, è fondamentale che la struttura e il contenuto del test result sia noto a priori e dunque a monte devono essere acquisiti durante il test in sala prova tutti i dati necessari, nei tempi e modi opportuni. Ciò è facilitato dal fatto che sia lo stesso automation engineer a occuparsi del post-processing, di conseguenza i test automatici e i task d'automazione sono implementati in modo coerente con la struttura di elaborazione realizzata su Concerto (e viceversa).
Una volta importati i dati dai test, Concerto lavora su una struttura fissa, avulsa dal formato di provenienza, che permette in ambiente di programmazione di accedere ai canali registrati. Tale organizzazione può essere così schematizzata (Fig. 1.16):

1. **File**: una volta aperto un test result, ad esso è associato un **Alias** tramite il quale è possibile esplicitare un riferimento. É possibile definire diversi **Datasource**, che assegnano uno specifico formato ad una determinata locazione in DB e filesystem locali o remoti, con il vantaggio di poter assegnare a tali file formule già implementate, regole di traduzione per i nomi dei canali e altri meccanismi di preconfigurazione automatica.

2. **Chiavi**: ogni file è suddiviso in chiavi, che rappresentano determinate fasi o porzioni dell'acquisizione fatta in sala prova. Possono essere ricondotte a due tipologie:

   (a) **Time Based**, anche note come *recorder*, che consistono in acquisizioni effettuate in sala prova in modo "continuo", cioè con la scrit-

[16] urlwww.avl.com/web/guest/-/avl-concerto-5-

tura appunto continua dei valori attuali dei canali con una frequenza impostata.

(b) **LogPoint Based**, anche note come *measurement*, che rappresentano acquisizioni "puntuali", cioè di un singolo valore dei canali molto spesso ottenuto da una media in un dato intervallo di tempo.

3. **Canali**: infine nelle chiavi sono contenuti diversi canali, cui è associato un tipo di dato e un'unità di misura, esattamente come configurato in sala prova prima dell'acquisizione.

Il linguaggio previsto per la programmazione su Concerto, molto simile a VB, è proprietario AVL[17] e presenta molte classi adatte alla gestione dei dati importati. Nelle versioni più recenti[18] inoltre è stato integrato un supporto per script e formule **Python**[19], che apre al vasto mondo della community di utilizzatori di questo linguaggio e permette di fruire dei vantaggi dettati dalla sua versatilità e riusabilità. L'integrazione dei due linguaggi, associata ad una gestione avanzata dell'automazione di banco, costituisce la chiave per realizzare architetture molto potenti che coinvolgono ogni aspetto della sperimentazione motoristica.

[17] É utilizzato anche su AVL PUMA (sistema d'automazione, vedi paragrafo precedente) per l'implementazione delle formule.

[18] Per questo progetto è impiegato Concerto 5 R3.1 [2].

[19] urlhttps://docs.python.org/3/

Capitolo 2

Descrizione del progetto

Il contesto di ricerca descritto a grandi linee nel capitolo precedente, lascia intendere l'importanza cruciale che riveste l'affidabilità di tutti gli strumenti adoperati e delle metodologie adottate in ogni fase delle attività condotte sui motori. Dai dati ottenuti dalle prove dipende non solo l'evoluzione del prodotto, ma soprattutto la sua idoneità alla produzione e alla vendita: è necessario che su tali dati si abbia la maggiore confidenza possibile, perché siano attendibili e difendibili. Uno degli aspetti chiave del nostro lavoro consiste proprio nella **validazione dei risultati**, sia nel caso in cui siano misure dirette, sia nel caso in cui siano prodotto di calcoli. Oltre a basare naturalmente su un metodo documentato il processo che porta all'ottenimento di questi dati, è necessario dunque corroborare il processo con una fase di verifica, che scovi eventuali errori difficili da rilevare. Si tratta di un'attività di controllo fondamentale non solo per l'importanza dei dati in questione (per esempio consumi ed emissioni dichiarate) ma soprattutto per individuare eventuali vulnerabilità nei metodi adoperati permettendo così di far evolvere qualitativamente l'attività di ricerca.
Per corredare il concetto appena esposto di un semplice esempio, è utile fare menzione dei risultati dei test omologativi in termini di emissioni, per comprendere la complessità del procedimento di misura e calcolo. Per ciascun inquinante sottoposto a limiti dalla normativa vigente, è presente nel report ufficiale un valore normalizzato al lavoro effettuato dal motore durante il test (riportato in g/kWh). Tale grandezza risultante dipende dunque da:

- Il lavoro, in kWh, calcolato come integrale della potenza (kW) durante il ciclo, a sua volta calcolata a partire dalle misure istante per istante di velocità (rpm) e coppia (Nm);
- Il quantitativo di inquinante emesso, in g, calcolato a partire dalle misure istante per istante di concentrazione dell'inquinante (ppm o Vol%), portata allo scarico (kg/h), umidità e temperatura dell'aria, oltre che dalla composizione chimica del carburante.

Si evince dunque quanto articolato sia il procedimento per il calcolo dei dati di output delle prove, ma soprattutto quanto numerosi siano i nodi critici, potenzialmente vulnerabili rispetto a eventuali errori o malfunzionamenti strumentali. Avere a disposizione un ulteriore strumento di validazione permetterebbe così di basare le analisi motoristiche su dati consolidati e più affidabili e quindi focalizzare ulteriormente l'attenzione sullo sviluppo del prodotto.

2.1 Obiettivi

L'idea alla base di questo progetto nasce dalla volontà di validare i risultati di una prova proponendo un approccio ai dati raccolti in quanto tali. Molto spesso le acquisizioni fatte in sala prova celano informazioni di ardua accessibilità, poiché disperse in un quantitativo molto elevato di dati. Nell'ottica di snellire il lavoro di interpretazione o effettuare un'operazione di *data mining*, può essere utile utilizzare algoritmi di **machine learning**. É stato applicato questo concetto alla necessità di un controllo di plausibilità delle prove: l'obiettivo di questo progetto è l'implementazione di uno strumento di validazione dei risultati ottenuti da test effettuati al banco, sulla base dei dati relativi a test simili effettuati precedentemente. Nel caso in cui il processo di test, misura ed elaborazione sia stato affetto da errori, l'esito di tale controllo potrebbe rivelare un'anomalia statistica, qualora il risultato ottenuto sia troppo distante dalla proiezione ottenuta a partire dai dati pregressi. Questo strumento prende il nome di **AITV**: **A**rtificial **I**ntelligence **T**est **V**alidator.
Uno dei requisiti fondamentali, su cui si basa il sistema che si vuole realizzare, è la sistematica raccolta in un **Database** unico delle informazioni relative alle prove eseguite. Ciascuna entry del DB deve riportare tanto i parametri generali di input del test, che possano caratterizzare e condizionare la prova, quanto ciascuno dei relativi risultati. Questo implica un coinvolgimento in questo sistema dell'intera piattaforma di test, dall'automazione al post-processing, passando per le aree di archiviazione.
Il controllo di plausibilità è effettuato, come sopra accennato, attraverso un modello predittivo implementato tramite tecniche di **intelligenza artificiale**, più nello specifico di **machine learning**. Tale modello è allenato su un **training set** costituito dalle prove presenti nel DB ed è utilizzato per predire il risultato selezionato, dati gli input della prova in esame. Il valore così ottenuto può essere confrontato con il dato estratto realmente dal test.
Lo strumento proposto è stato concepito per offrire un supporto agli ingegneri responsabili delle valutazioni sulle attività condotte, con uno spettro di applicazioni potenzialmente molto ampio. Naturalmente i dati più sensibili sono quelli prodotti dai test omologativi e su questi è molto importante effettuare il maggior numero possibile di controlli di plausibilità, in fase di sviluppo del prodotto. Allo stesso modo tuttavia è conveniente accertare i risultati di qualsiasi altra prova

di sviluppo o durata, i quali, una volta condivisi con i restanti centri R&D FPT sparsi nel mondo, condizionano il successivo percorso di evoluzione del prodotto.

2.2 Architettura del sistema

Lo strumento introdotto nel paragrafo precedente, che prende il nome di **AITV** (**A**rtificial **I**ntelligence **T**est **V**alidator), coinvolge diversi ambienti che compongono il *testfield* e di conseguenza ciascuno di essi entra a far parte dell'architettura del sistema realizzato. É stato infatti necessario tanto agire su strutture d'automazione e test automatici in **PUMA** e **AdaMo**, quanto implementare la piattaforma di elaborazione su **Concerto**. Inoltre sono stati definiti un formato comune e un'area di storage centralizzata per i dati provenienti dai due diversi sistemi di automazione, il che ha comportato un lavoro di adattamento significativo.
In Fig. 2.1 è schematizzata l'architettura generale del sistema realizzato, che ne sintetizza l'intero processo di funzionamento ed evidenzia tutti gli ambiti interessati dal lavoro di implementazione compiuto in questo progetto. La descrizione che segue è organizzata proprio secondo la procedura della sperimentazione, dalla teorizzazione del test in sala prova, alla post-elaborazione e validazione del risultato.

2.2.1 Definizione delle prove

Come illustrato nel capitolo precedente, le diverse attività sperimentali condotte sui nostri motori comportano la progettazione di numerosi test che la sala prova è in grado di eseguire autonomamente, a seguito di un'opportuna implementazione. La gestione di sale prova di natura diversa (PUMA e AdaMo) impone tuttavia una **definizione dell'algoritmo** avulsa da ciascuna piattaforma software su cui sarà realizzato: per questa ragione risulta particolarmente utile una trascrizione in pseudocodice della procedura di prova automatica. É stato possibile in questo modo strutturare un database di prove (Fig. 2.2) disponibili, grazie al quale diventa più semplice il lavoro di consultazione ed eventuale aggiornamento dei cicli.
In questo contesto è semplice individuare, per ogni test già implementato in passato, in quale punto dell'algoritmo inserire il codice necessario per adeguare le strutture preesistenti al sistema realizzato in questo progetto: è infatti necessario che le definizioni dei test presentino uno step nel quale sono raccolti e salvati **gli specifici dati preliminari della prova**, che costituiranno i potenziali input della previsione effettuata in fase di validazione (ultimo blocco dell'architettura proposta in Fig. 2.1). In particolare i valori di queste variabili saranno memorizzati all'interno di una chiave predefinita del test result, a cui è stato assegnato il nome **V_IN**.

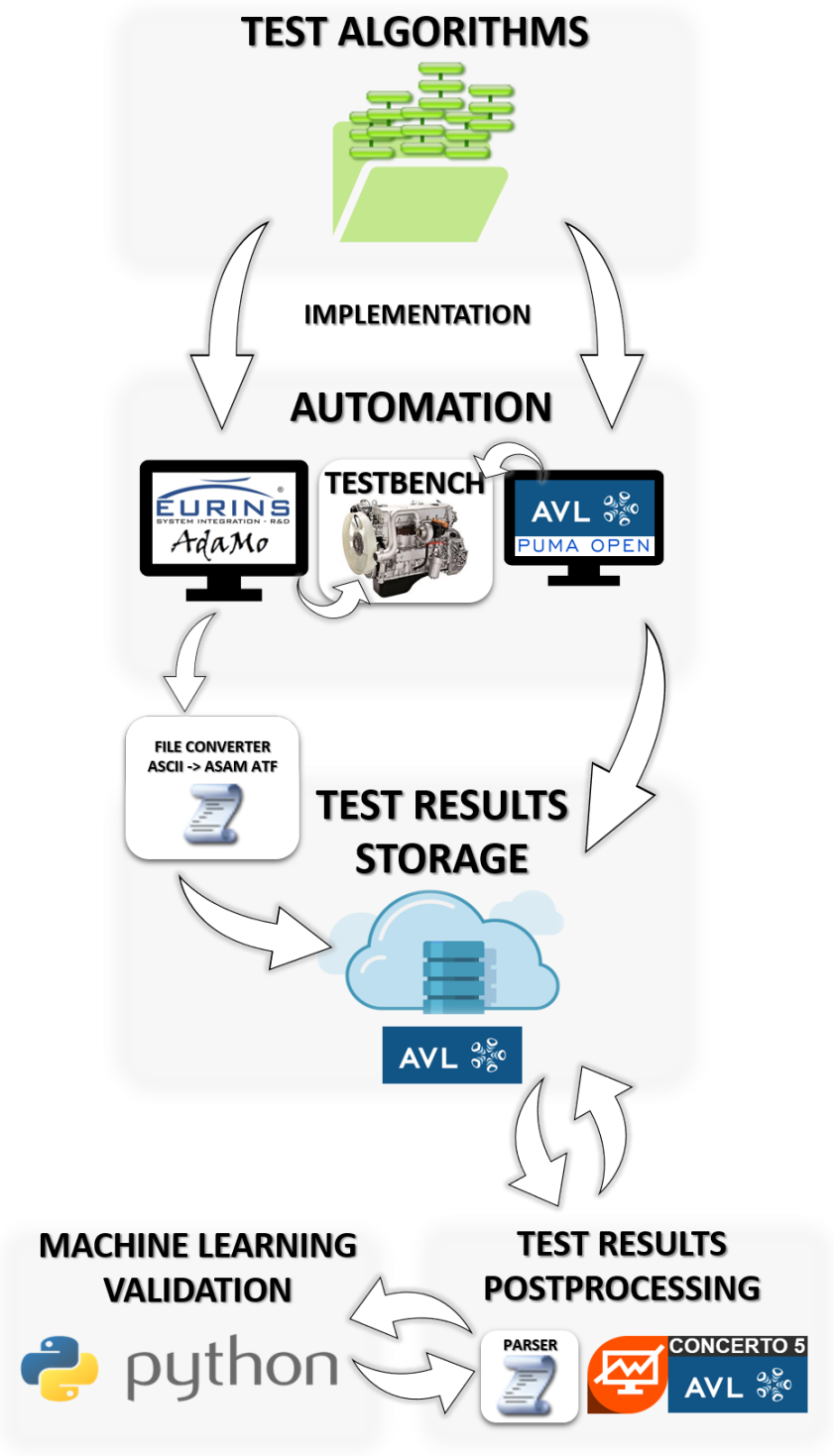

Figura 2.1: Architettura del sistema.

Figura 2.2: Database degli algoritmi di test.

2.2.2 Implementazione del test

Le procedure di test teorizzate ed espresse in pseudocodice sono implementate dall'automation engineer nei due sistemi d'automazione (Fig. 2.3), secondo le diverse tecniche e linguaggi di programmazione. Una corretta gestione di ciascuna sala prova, dal punto di vista dell'attuazione meccanica sul motore e delle misurazioni con sensori e strumenti, può garantire una solida base per ottenere test confrontabili anche se effettuati su banchi diversi. Per giungere a questo obiettivo è tuttavia necessario che anche la programmazione dei test sia realizzata in maniera corretta e coerente, il che richiede una profonda consapevolezza delle caratteristiche dei due sistemi d'automazione utilizzati nel centro di Foggia. Il **test result** della prova effettuata su un banco deve dunque presentarsi in modo analogo e quasi indistinguibile rispetto a quello ottenuto da un altro banco, anche nel caso in cui adoperi un sistema differente.
Il formato adottato da AVL per l'immagazzinamento dei dati raccolti da PUMA durante un test è ASAM ODS (Open Data Services) [3] [9]: ciascun test result costituisce un record di un database opportunamente strutturato ed è organizzato secondo la suddivisione illustrata nel capitolo precedente, in chiavi e canali stabiliti dall'utente. D'altro canto sulla piattaforma AdaMo è possibile produrre diversi file di testo (ASCII) durante l'esecuzione di una prova. Da qui nasce l'idea di unificare il formato del test result per entrambi i sistemi di automazione, adoperando sui file di testo prodotti da AdaMo un passaggio di traduzione per mezzo del **FileConverter**, un componente software opportunamente realizzato in linguaggio Concerto/Python. Data una particolare prova, la sua implementazione su PUMA, detta **TST**, prevederà il salvataggio in momenti opportuni delle diverse chiavi *recorder* o *measurement*; l'analogo programma di test realizzato su AdaMo, detto **prv**, produrrà tanti file di testo quante sono le chiavi richieste dalla prova. La folder contenente tutti i file generati da AdaMo sarà infine data in input al **FileConverter**, il quale produrrà un file ATF (ASAM Transport Format) strutturato esattamente come il record ASAM ODS restituito dal TST di PUMA.

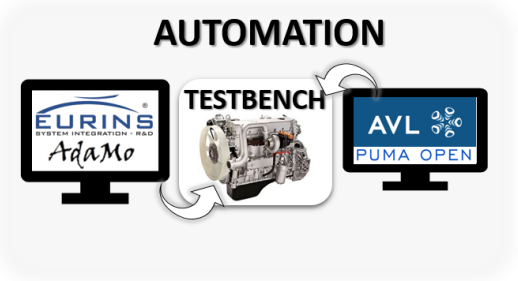

Figura 2.3: Automazione di sala prova.

2.2.3 Archiviazione dei risultati

Diverse sono le soluzioni per archiviare i test result prodotti dai sistemi d'automazione. Naturalmente l'utilizzo di un **database** centralizzato è considerata quasi sempre la soluzione migliore. Le sale prova AVL tendenzialmente permettono uno storage centralizzato - per mezzo di un database AVL Santorin ASAM ODS [1] - per tutti i banchi PUMA che compongono lo stesso *testfield* e di conseguenza un solo database raccoglie tutti i test result provenienti da ciascuna sala prova PUMA, oltre che ogni file di parametrizzazione, inclusi i TST. Almeno attualmente, questo non è il caso delle nostre due sale AVL, che sono dotate ciascuna del proprio server ASAM ODS locale. Allo stesso modo ogni sala AdaMo prevede un'area di archiviazione dei file di risultati nel filesystem locale. Tuttavia tutti i PC Windows di sala prova, che ospitano la parte di interfaccia utente dei sistemi d'automazione, hanno una scheda di rete riservata e adoperata per la connessione alla rete LAN aziendale, alla quale accedono anche tutti i laptop degli ingegneri.
Una delle funzionalità più utili di Concerto, come accennato nel capitolo 2, è la possibilità di definire dei **Datasource**, cioè dei *data provider* che associano a unità di archiviazione remote un determinato formato di file e altre preferenze di configurazione. Per questo progetto è stata adoperata una semplice organizzazione dei dati: ciascuna delle sei sale prova dispone della propria area di storage - in alcuni casi direttamente sul PC di sala prova e in altri casi su una risorsa di archiviazione condivisa - ma ciascuna mette in condivisione tale struttura attraverso un Datasource; di conseguenza è stato sufficiente creare su Concerto **un Datasource per ogni banco**, garantendo così l'accesso a tutti i test result (fig. 2.4). Naturalmente nel caso delle sale PUMA, il Datasource è configurato in modo tale da accedere direttamente al server ASAM ODS, mentre per le sale AdaMo è stato associato ad un determinato path nel filesystem di una risorsa di archiviazione condivisa nella rete aziendale, in cui sono caricati i file in formato ASAM ATF. Lo step di traduzione della folder di risultati di AdaMo, contenente i

[1] `www.avl.com/testing-solutions-for-batteries/-/asset_publisher/gYjUpY19vEA8/content/avl-santorin-asam-ods-server`.

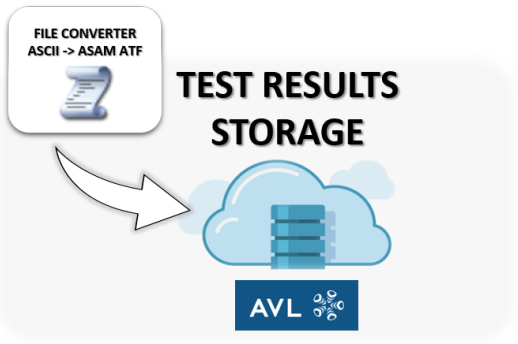

Figura 2.4: Archiviazione dei file di risultati dei test.

diversi file di testo raccolti, è performato preliminarmente alla post-elaborazione della singola prova, per mezzo dell'esecuzione del **Job Concerto**[2] relativo al FileConverter precedentemente menzionato, che genera un ATF file accessibile attraverso il Datasource.
In definitiva, una volta che la prova è stata eseguita su un sistema PUMA, i risultati sono archiviati localmente ma disponibli attraverso il corrispondente Datasource per essere post-processati con appositi layout. Diversamente, nel caso di AdaMo è necessario eseguire la conversione con il FileConverter, perché il test result sia disponibile per il post-processing in un formato adeguato allo standard AVL. Il risultato è un unico Database centralizzato per tutte le sale prova, che fisicamente è dislocato in tre diverse aree d'archiviazione, ma logicamente rende accessibili i dati di ciascun banco attraverso un solo strumento e una sola tecnica.

2.2.4 Post-Elaborazione dei dati raccolti

I dati raccolti dai banchi, resi disponibili in Concerto dal Database centralizzato appena descritto, sono infine elaborati con layout e formule specifici per ogni tipologia di test. Questo processo permette di determinare i risultati finali, anch'essi strettamente correlati alla prova di riferimento: risultati finali di un test omologativo WHTC sono ad esempio i valori di inquinanti emessi (UoM mg/kWh), il lavoro compiuto dal motore (UoM kWh) e così via. Le formule da implementare, la disposizione delle finestre nel layout e la struttura del report generato da Concerto sono tutti dettagli definiti in fase di preparazione dell'attività di test. É bene osservare quanto sia importante che i dati salvati dall'automazione siano completi e consistenti: la messa a punto di maschere di elaborazione efficaci va molto spesso di pari passo con l'implementazione del test al banco.
Uno degli elementi fondamentali su cui si basa questo progetto è il già denso ar-

[2]Il Job è un particolare task eseguibile nell'ambiente di Concerto, implementato e configurato dall'utente. Questo concetto sarà approfondito nel prossimo capitolo.

chivio dei layout di post-elaborazione utilizzati in diversi dei nostri centri R&D, per attività di ogni genere. Ad esso se ne aggiungono continuamente di nuovi, naturalmente in concomitanza con l'implementazione dei relativi test nell'automazione. Ciascuno di questi layout, una volta eseguito, produce una serie di **risultati** del test effettuato, che potranno essere oggetto di validazione mediante AITV, come sarà descritto nel prossimo paragrafo. Qualora tali dati siano ritenuti affidabili, potranno essere **aggiunti all'insieme dei risultati associati alla tipologia di prova elaborata**, che costituisce il **training set** con cui è addestrato il modello di validazione. Per fare questo, è eseguito un **Job** di Concerto che prende il nome di **Parser**, implementato in linguaggio Concerto/Python, invocabile dopo l'esecuzione del layout. Questo componente software ha il duplice scopo di fare un **parsing** sul file di risultati per ottenere tutti i dati preliminari del test immagazzinati nella chiave **V_IN** e di raccogliere i relativi risultati specifici del test elaborato. Queste informazioni sono trascritte su un file di testo associato alla specifica tipologia di prova, considerato il vero e proprio **training set**: esso contiene una serie di campioni corrispondenti a tutti i test eseguiti che afferiscono alla stessa tipologia, per i quali sono riportati input e output e su cui effettuerà l'apprendimento il modello di validazione basato su machine learning. Facilmente si evince dunque che è previsto **un file di training set per ciascun tipo di prova**, ognuno dei quali è popolato opportunamente dal **Parser** nel momento in cui è eseguito per raccogliere i dati ottenuti da nuovi test. Esso è dotato della necessaria flessibilità grazie alla quale discrimina il tipo di prova per cui è stato eseguito e di conseguenza seleziona quanti e quali risultati raccogliere dall'elaborazione popolando il corrispondente training set.
Si possono così riassumere i passaggi descritti finora: la prova è teorizzata, implementata e infine eseguita sul sistema d'automazione e i dati raccolti sono condivisi dalle sale attraverso i Datasource, con cui è possibile importare in Concerto il test ed elaborarlo con l'opportuno layout. A questo punto è possibile validare i risultati mediante l'esecuzione di AITV (come vedremo nel prossimo paragrafo) o aggiungerli - insieme ai parametri di input - al training set mediante l'esecuzione del Parser, che aggiorna la struttura su cui si basa il modello di validazione. Per questo motivo **post-elaborazione e validazione costituiscono due elementi dell'architettura paralleli e connessi**, che sono dunque posizionati sullo stesso livello. L'utente gode della libertà di scegliere se e quale Job eseguire dopo aver caricato il file di risultati e applicato su di esso il layout, tuttavia la logica di utilizzo di questi strumenti prevede che i risultati siano naturalmente validati prima di utilizzarli per aggiornare il training set. Questa sequenzialità delle due operazioni è tuttavia affidata all'utente, che in sintesi dovrebbe teoricamente:

1. identificare il file di risultati del test eseguito in sala prova;
2. eventualmente convertirlo con il **FileConverter** qualora provenga da una sala prova AdaMo;
3. caricarlo in Concerto e applicare il layout di post-elaborazione opportuno;

Figura 2.5: Post-elaborazione dei dati.

4. validare i risultati con **AITV**;

5. usare il **Parser** per aggiungere al training set associato alla tipologia di prova nuove entry contenenti i parametri di input della chiave `V_IN` e i relativi risultati calcolati dalla maschera di post-elaborazione.

Avere caricato in Concerto il file di risultati e applicato su di esso il layout costituisce dunque l'unico requisito per l'esecuzione tanto di AITV quanto del Parser e, a seconda dello step che si sta effettuando, è possibile eseguire l'uno o l'altro.

2.2.5 Validazione dei risultati

L'elemento che completa l'architettura è **AITV**, cioè il validatore dei risultati realizzato in linguaggio Python e sviluppato per l'ambiente di Concerto, di cui sono sfruttate le potenzialità già analizzate nel primo capitolo, in particolare per accedere ai dati e presentare graficamente il risultato della validazione.
Il ritmo da sostenere in un centro Testing FPT è molto sostenuto, le attività seguono pianificazioni ben scandite ed è cruciale che la post-elaborazione estragga dai dati raccolti un'interpretazione il più possibile rapida, univoca, chiara e **affidabile**. Per validare i risultati ottenuti sono generalmente adoperate metodologie di analisi basate su studi approfonditi dei fenomeni fisici e chimici che regolano emissioni, consumi e prestazioni dei motori. Nei modi proposti da questo progetto, il supporto dell'Intelligenza Artificiale e dunque del Machine Learning non è mirato assolutamente a sostituire le metodologie di validazione preesistenti, ma vuole anzi costituire un ulteriore strumento di analisi degli esiti delle prove, anche sul piano qualitativo. Il sistema è in grado di mostrare l'evidenza di un risultato "fuori statistica" secondo una predizione basata sui parametri selezionati dall'utente: si tratta quindi di un'analisi dipendente dagli input scelti e di conseguenza può essere valutata anche la stessa correlazione tra la quantità predetta (il risultato) e quelle di input (i parametri preliminari del test).

Figura 2.6: AITV: Validatore dei risultati con Intelligenza Artificiale.

Il validatore **AITV** (fig. 2.5) può essere eseguito - analogamente al Parser e al FileConverter - come un **Job Concerto**, appena dopo aver effettuato l'elaborazione del test con il relativo layout. Anche in questo caso si sfrutta il fatto che il test result d'interesse sia già caricato e aperto[3] e dunque, a seconda del tipo di prova, il validatore raccoglie i risultati appena calcolati e li sottopone all'utente perché ne selezioni uno. Determinato il risultato da validare, cioè l'output della predizione, l'utente può ora scegliere una o due variabili di input tra i parametri preliminari raccolti nella chiave **V_IN** e avviare la predizione. Queste operazioni possono essere ripetute più volte, finché non si ottengono le informazioni desiderate. L'apprendimento del modello avviene per mezzo della tecnica del **Gradient Descent** [12], che sarà descritto più nello specifico nel prossimo capitolo. L'ultimo step è la predizione finale del risultato da validare, che è quindi possibile confrontare con quello realmente calcolato dalla post-elaborazione per determinarne la distanza. Come vedremo, una buona predizione può essere motivo di rassicurazione sulla validità del test, ma in caso contrario si possono aprire diversi scenari: qualora il modello generato non sia ritenuto opportuno, può essere effettuata nuovamente l'esecuzione del validatore con una diversa parametrizzazione; qualora invece la stima di AITV sia in ogni caso troppo lontana dal valore ottenuto dal layout, è plausibile che qualcosa sia andato storto in uno dei tanti step della procedura di test.

[3] É comunque possibile sostituire il file di risultati attualmente caricato in Concerto con un altro relativo alla stessa tipologia di prova, per mezzo dell'ambiente Data Expoler, con cui si accede ai diversi Datsource.

Capitolo 3

Implementazione

La teorizzazione del sistema descritto nel capitolo precedente nasce da una consapevolezza della necessità di consolidare i dati raccolti dai test, che può essere maturata solo confrontandosi praticamente con gli aspetti più critici dell'attività di ricerca sui motori. Analogamente la realizzazione di tale architettura richiede una conoscenza adeguata delle diverse tecnologie impiegate nello svolgimento dei test in sala prova. Questo sistema non è solamente frutto del lavoro fatto in questo progetto, ma anche dell'efficienza dell'ormai matura piattaforma di test su cui si poggia, che è di conseguenza idonea ad accogliere la struttura proposta. Nei paragrafi che seguono sono approfondite le fasi che hanno scandito l'implementazione dell'architettura fin qui delineata.

3.1 Automazione

L'automazione è stato il punto di partenza per l'implementazione del sistema così come per la descrizione dell'architettura fornita nel capitolo precedente. É infatti in sala prova che nascono i dati che AITV valida e solo in sala prova è possibile acquisire e determinare i valori di tutte le grandezze su cui si basa la predizione che AITV effettua.
Come descritto nel paragrafo 1.2.2, il prodotto dei test condotti in sala prova è un file di risultati o **test result**, strutturato in chiavi che raccolgono gruppi di canali con i relativi valori acquisiti puntualmente o continuativamente. É bene specificare che ciascun tipo di chiave dispone di uno o più **measurement ID**[1], cioè porzioni della chiave ugualmente strutturate, ma acquisite in istanti diversi. Per ciascun measurement ID vengono aggiunte nuove righe, a mano a mano che sono condotte misure puntuali o per ogni campione acquisito in una misura continua.
Il lavoro che è stato necessario effettuare sull'automazione riguarda principal-

[1]Si tratta di una nomenclatura tipica di AVL Concerto [2].

mente l'acquisizione dei dati preliminari per la predizione. La scelta migliore per organizzare questi dati consiste nella creazione della chiave apposita **V_IN**, che si aggiunge così alla struttura del test result specifica di ogni tipologia di prova. In molti casi i valori raccolti in `V_IN` sono comunque presenti in altre chiavi del file di risultati, ma il metodo adottato, sebbene introduca una ridondanza, garantisce ordine e una facile fruibilità di questi dati in fase di elaborazione e validazione. Ad ogni modo, tale ripetitività nell'acquisizione non costituisce certamente un motivo di inefficienza per quanto riguarda gli spazi d'archiviazione, data la modesta dimensione della chiave `V_IN`.
A ciascuna tipologia di prova sono associati i caratterizzanti parametri preliminari da acquisire. In alcuni casi si tratta di valori noti già prima dell'avvio del test, ma talvolta è necessario acquisire determinate grandezze durante la prova stessa. Nell'esempio costituito dal caso di studio proposto nel capitolo 4, il *Lambda Step*, è necessario salvare il valore di temperatura del catalizzatore prima dell'esecuzione delle transizioni di lambda. In questi casi il lavoro compiuto sull'automazione potrebbe comportare anche l'inserimento - nell'algoritmo di test - di procedure apposite volte all'ottenimento dei dati preliminari: potrebbe essere necessario ad esempio portare il motore in un particolare punto operativo, che magari non è teoricamente richiesto dalla prova, o anche mettere in misura uno strumento che altrimenti non sarebbe indispensabile. Queste operazioni non devono comunque compromettere la validità della prova, rispettando tutti i vincoli descritti dalla normativa associata.
Facendo riferimento alla classificazione fatta nel paragrafo 1.2.2, si sottolinea che la chiave `V_IN` è di tipo **logpoint**, cioè è frutto di un salvataggio puntuale, che si differenzia dal quello continuo dei *recorder*. Tendenzialmente questo tipo di acquisizione è detta **measurement**, poiché è tipica delle misure stazionarie in cui ad ogni grandezza richiesta e associata una media dei valori accumulati in una data finestra temporale. Nelle prove stazionarie, come ad esempio il cosiddetto *Engine Map* (o *Piano Quotato*), le chiavi logpoint di misura sono sufficienti a contenere tutti i dati significativi del test. Oltre a questo impiego, le acquisizioni di tipo puntuale sono anche adoperate per l'immagazzinamento di dati, quali costanti o input generici dell'utente, che sono semplicemente archiviati nella chiave target senza calcolare una media.
É stato scelto questo tipo di chiave per `V_IN` perché perfettamente idoneo a raccogliere i dati preliminari che, come già detto, si presentano come costanti note o come valori da acquisire puntualmente. Non è infine raro trovare nella chiave `V_IN` più righe: alcune prove consistono nel ripetere ciclicamente una procedura su diversi punti operativi del motore e per ciascuno di essi i parametri caratterizzanti da inserire in `V_IN` variano anche sensibilmente. É esattamente il caso del *Lambda Step*, descritto nel capitolo 4: le transizioni di lambda sono effettuate su 15 punti operativi e per ciascuno di essi è riportato una riga in `V_IN`.
Definita la struttura desiderata per i dati preliminari e stabilito che ciascuna prova prevede uno o più punti dell'algoritmo di test in cui acquisire e salvare

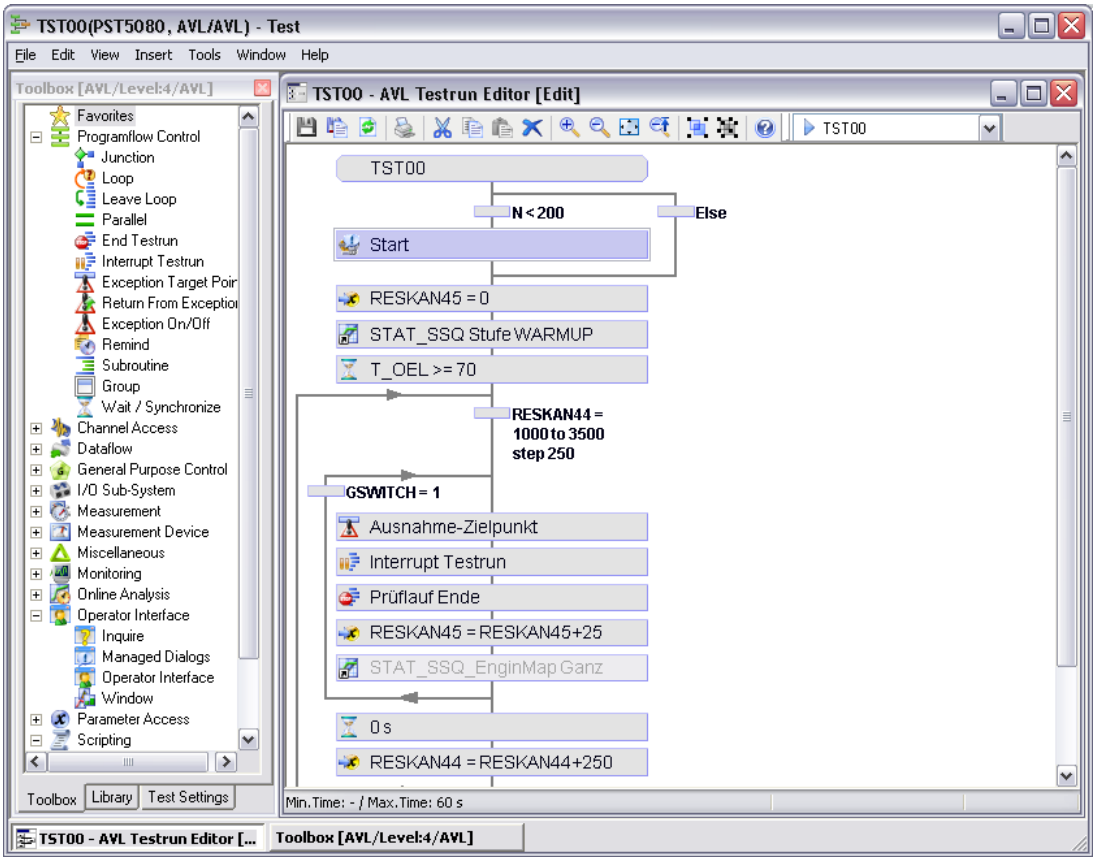

Figura 3.1: BSQ di PUMA.

questi parametri, non resta che analizzare le tecniche di salvataggio adoperate per la chiave `V_IN` nei due diversi sistemi di automazione, PUMA e AdaMo.

3.1.1 AVL PUMA Open

Il salvataggio dei dati preliminari è dunque effettuato come una semplice scrittura di una riga sotto `V_IN` a partire da variabili (dette **Quantity** o **NormName**) di PUMA, che devono essere disponibili e consistentemente valorizzate al momento del salvataggio.
La natura e la modalità di acquisizione dei valori delle quantity richieste variano in base alla tipologia di dati preliminari che caratterizzano la prova. Potrebbero infatti riguardare parametri costanti e noti già prima dell'avvio del test o al contrario potrebbe essere richiesta l'acquisizione contestuale di specifiche grandezze. In ogni caso il sistema d'automazione AVL PUMA Open fornisce un'approccio ben strutturato alla **misura stazionaria**, con tecniche fruibili manualmente o all'interno di un test automatico.
Prima di analizzare i metodi di salvataggio messi a disposizione dal sistema, è bene introdurre sinteticamente alcuni dei principi di funzionamento di PUMA.

Nel POI (PUMA Open Interface, GUI di PUMA), il sistema può presentarsi in tre stati:

1. **MONITOR** Stato base in cui è concesso il controllo generale di tutti i sensori, attuatori e strumenti connessi al banco, in base alla parametrizzazione dei due *loadset* che è necessario caricare:

   - **SYS** (System Parameters), che contiene le impostazioni generali della sala prova ed è suddiviso in **blocchi** di vario tipo, come ad esempio:
     - **SCP**, per la configurazione del freno e dei controllori di Giri (o Coppia) che agiscono su di esso
     - **EMC**, per la parametrizzazione di segnali e controllori di basso livello della sala prova
     - **FFS**, per le caratteristiche dei sensori connessi e ingressi/uscite analogiche/digitali
     - **CAN**, per configurare comunicazioni su CAN Bus
     - **MDV** e **EBH**, per la configurazione degli strumenti di misura
     - **ASC** e **TCC**, per definire macchine a stati in linguaggio VBS
     - **FDV**, per definire formule
     - **SCR**, per implementare VBScript eseguibili nel POI
     - **GWA**, per definire limiti d'allarme su determinate quantity e le relative reazioni del sistema in caso di superamento della soglia
     - **MEI**, per configurare la comunicazione ASAP3 [7] con l'application system (tipicamente ETAS INCA) e i parametri scambiati
     - **PID**, per configurare controllori PID ad esempio per scambiatori termici
     - **LTC**, per definire *lookup tables*
   - **TFP** (Test Facility Parameters), che descrive la parametrizzazione comune di tutto il *testfield* e che quindi teoricamente dovrebbe essere uguale per tutte le sale prova dello stesso sito. Può contenere blocchi di alcuni tipi comuni anche al SYS, ma in particolare presenta diversi blocchi **DST** (Data Storage Table), anche noti come *piani di memorizzazione*. Ciascuno di essi rappresenta un raggruppamento di quantity a cui si può fare riferimento nella definizione di misure stazionarie o di recorder. É buona prassi organizzare questi blocchi in modo tale che possano identificare degli insiemi di variabili accomunate dalla stessa natura. Per esempio è consueto creare un DST per tutte le quantity relative alle concentrazioni dei vari inquinanti, provenienti dai banchi analisi.
     Inoltre il loadset TFP contiene il blocco **KEY**, nel quale sono dichiarate le **chiavi** di memorizzazione che andranno a comporre il file di risultati del test. In questo blocco sono riportate chiavi di due tipi:

- **Internal Key**, che PUMA utilizza per archiviare automaticamente dati e informazioni di log durante le transizioni nei vari stati di funzionamento;
- **Measured Data Key**, a cui l'utente può associare l'area di archiviazione di una misura stazionaria;

2. **MANUAL** Stato completamente operativo di PUMA, in cui è possibile accendere e manovrare il motore, effettuare misure stazionarie e avviare o interrompere recorder. Per raggiungerlo è necessario caricare altri due *loadset* precedentemente configurati:

   - **UUT** (Unit Under Test Parameters), che costituisce la configurazione del motore in prova, per mezzo della dichiarazione di parametri generali che lo descrivono e di strutture ad esso associate. Anche nel loadset UUT è possibile inserire tipi di blocco disponibili nel SYS o nel TFP, ma presenta in particolare i seguenti blocchi che lo caratterizzano:
     - **ECT**, blocco di parametrizzazione del motore, nel quale sono dichiarati valori di riferimento di Velocità e Coppia, oltre ad altre quantity libere;
     - **EMP**, blocco in cui è riportata la **Engine Map** o **Full load Curve**, che riporta la massima Coppia disponibile per ogni valore di Velocità del motore. Questa curva è sovrascrivibile nello stato di MANUAL, con un'apposita procedura automatica *embedded* di PUMA. Di fatto la EMP caratterizza la UUT ed è fondamentale per l'esecuzione degli *statutory test* (cicli omologativi) che consistono in profili di Giri e Coppia calcolati a partire da tale curva.
   - **TST** (Test Parameters), che contiene l'algoritmo di test, il quale fa riferimento all'intera struttura precedentemente descritta. Per questo motivo è necessario che vi sia coerenza e consistenza con la parametrizzazione di banco (SYS), di memorizzazione (TFP) e di unità di prova (UUT) attualmente caricate. Il TST permette di descrivere quindi una procedura di test **automatica**, che può prevedere il controllo del motore in diversi punti operativi, un'interazione con l'utente, l'esecuzione di VBScript, il lancio di misure stazionarie, l'avvio di recorder e molte altre operazioni. L'algoritmo o BSQ (Block Sequence) consiste sostanzialmente in un diagramma di flusso composto da diversi "mattoncini" chiamati **AO** (Activation Object) pre-implementati e disponibili nel *toolbox* (Fig.3.2). Ciascun AO è in realtà riconducibile a determinate *function* o *sub* VB ed è dunque possibile creare VBScript con lo scopo di effettuare molte delle operazioni eseguibili con gli appositi AO. Questa scelta può infatti risultare preferibile per

rendere il BSQ più leggero ed efficiente.
All'interno del TST tuttavia vanno configurati alcuni oggetti utili per il test, alcuni dei quali sono fruibili anche nello stato di MANUAL. In particolare, per quanto riguarda il salvataggio dei dati, è possibile definire Recorder e Measurement Request (o **MRQ**), a ciascuno dei quali sono associati:

- i DST contenenti le quantity da salvare (si noti che tali DST devono essere presenti nel TFP caricato, per coerenza di contesto)
- la chiave sotto cui memorizzare il salvataggio

Una volta inseriti questi oggetti nella *test library*, possono essere utilizzati all'interno del BSQ come argomenti di appositi AO (o di VBScript) ma sono già disponibili nello stato di MANUAL, in cui il TST è ormai caricato. In questo modo è possibile avviare anche manualmente misure stazionarie o recorder, senza che sia necessariamente il test automatico a farlo. Un discorso analogo vale per tutti gli VBScript presenti nella libreria del TST, così come limiti d'allarme.
Altri oggetti che possono essere inseriti nella libreria sono le **SubRoutine**, cioè porzioni di flowchart del tutto analoghe al "main" BSQ, di cui sono quindi sotto-procedure, e gli **SSQ** (StepSequence), che consistono in una successione di step elementari nei quali il motore è portato in un dato punto operativo, con un certo tempo di rampa e un certo periodo di assestamento. Il set di Giri e/o Coppia, i tempi e altre impostazioni che regolano l'SSQ prendono il nome di **Step Buffer Variable** e possono essere parametrizzate con gli ASC (Assignment Script, anche noti come Step Buffer Formula), vale a dire una speciale tipologia di script utili a manipolare tali variabili in un linguaggio derivato da Visual Basic. Al momento dell'esecuzione di uno step di un SSQ, la parte del sistema PUMA che gira su Windows è incaricata di assegnare valori alle Step Buffer Variablea secondo l'ASC caricato, ma è la parte real time di PUMA (INtime) ad eseguire il controllo sulla unit under test.

3. **AUTOMATIC** Stato in cui il sistema prende integralmente il controllo del banco e del motore, al quale si può approdare dallo stato di MANUAL. Una volta richiesto lo stato di AUTOMATIC, è lanciato il BSQ configurato nel loadset TST. Al complemtamento del test automatico, il sistema torna nello stato di MANUAL o direttamente in quello di MONITOR, a seconda dell'implementazione fatta: l'AO `Interrupt Testrun` porta in MANUAL mentre `End Testrun` porta in MONITOR). Diversamente l'interruzione del BSQ potrebbe non essere prevista, ma innescata da condizioni d'allarme e in tal caso il sistema torna solitamente in MANUAL. Failure molto gravi potrebbero tuttavia portare in sicurezza il sistema, ponendolo in MONITOR.

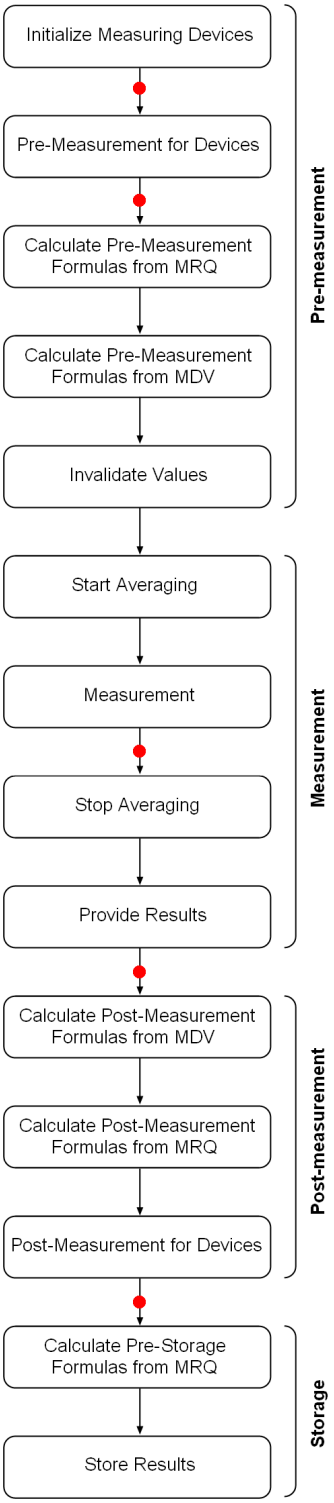

Figura 3.2: Procedura di misura in PUMA.

Definiti questi concetti generali sul funzionamento del sistema AVL PUMA Open, è possibile ora scendere maggiormente nel dettaglio delle misure stazionarie sui cui poniamo l'attenzione per via del loro ruolo cruciale nell'acquisizione dei dati per la chiave `V_IN`.
La figura 3.2 descrive la procedura di **Steady State Measurement**, che può quindi essere eseguita manualmente dall'operatore nello stato MANUAL o automaticamente dal TST nello stato AUTOMATIC. Può essere sintetizzata nelle seguenti 4 fasi:

1. Measurement Preparation o **Pre-Measurement**, in cui sono preparati gli strumenti di misura coinvolti e sono calcolate formule preliminari;

2. **Measurement** Execution, in cui sono posti in stato di misura tutti gli strumenti e sono di conseguenza acquisiti i valori di tutte le variabili coinvolte, da cui è infine calcolata la media nel tempo di misura;

3. **Post-measurement** Procedures, in cui sono effettuate operazioni di pulizia o reset sugli strumenti e sono calcolate formule a partire dai valori raccolti;

4. **Storage** Storage, in cui sono calcolate altre formule propedeutiche al salvataggio e sono infine archiviati i risultati della misura.

La seconda fase rappresenta naturalmente il passaggio più importante di questa procedura ed è riportata nel dettaglio in fig. 3.3, che ne illustra la forte dipendenza dallo stato della unit under test. Perché i dati raccolti in fase di misura siano consistenti, è infatti necessario che il motore sia stabile attorno ai valori impostati di Velocità, Coppia e Carico.
É bene specificare che in generale l'automazione di sala prova controlla il motore secondo una caratteristica o **modalità di controllo** (control mode), rappresentata dalle due grandezze in base alle quali sono pilotati rispettivamente il freno e il motore. Al freno infatti è solitamente trasmesso - per il controllo in retroazione - un *set point* (o *demand value*) di **Velocità**, al quale esso limita il motore (freno stazionario o freno dinamico in modalità passiva) o che garantisce erogando eventualmente una Coppia motrice (freno dinamico in modalità attiva). É tuttavia possibile anche trasmettere al freno un set di **Coppia** (solo per un freno dinamico in modalità attiva) che esso insegue frenando laddove la Coppia misurata sia più bassa del set point e trascinando in caso contrario. Analogamente il motore può essere controllato in più modi, traducendo il controllo sempre nell'attuazione di una richiesta di pedale trasmessa alla ECU. Questo set point può consistere semplicemente in un valore secco di pedale o **Alpha**, espresso in percentuale, su cui non è effettuato nessun controllo. In alternativa la richiesta di acceleratore può essere l'output di un controllo in retroazione su un valore obiettivo di **Coppia**, che agisce aumentando la richiesta al diminuire della *process variable* (la Coppia misurata) e diminuendolo in caso contrario. Allo stesso

modo è possibile calcolare l'output dell'acceleratore in base ad altre grandezze - quali ad esempio la Potenza o la Pressione Media Effettiva (PME) - attuando il cosiddetto **x-control**.
Tale dicotomia di variabili esprime quindi il *control mode* del punto operativo ed è uno dei parametri di ciascuno step dell'SSQ che può essere modificato via ASC. Molto spesso l'esecuzione dell'MRQ è parte di uno di questi step e il suo esito e la sua durata sono fortemente condizionati dalla stabilizzazione del punto operativo così definito. La figura 3.3 fa riferimento in particolare alla stabilizzazione della Velocità e mostra la suddivisione temporale dell'esecuzione di uno step con richiamo di MRQ. Di seguito è fornita una breve legenda:

1. **tr** (ramp time): il tempo richiesto per raggiungere il target di Velocità;
2. **tctrl** (control time): il tempo necessario a stabilizzare la Velocità;
3. **twΔ1** (waiting time 1): il tempo di attesa prima dell'avvio dell'acquisizione, che somma il tempo di rampa (1) e il tempo di controllo (2);
4. **tm** (measuring time): il tempo di acquisizione dei valori delle variabili associate a sensori, strumenti e qualsiasi altro elemento dell'automazione;
5. **ts** (step time): il tempo complessivo richiesto dallo step, che somma il tempo di rampa (1), il tempo di controllo (2), il tempo di misura (4) e il tempo di attesa finale (8);
6. **actual speed value**: il valore attuale della variabile di controllo;
7. **demand speed value**: il set point della variabile di controllo;
8. **twΔ2** (waiting time 2): il tempo di attesa successivo alla misura, dopo il quale giunge alla conclusione lo step.

A questo punto è possibile definire le modalità d'implementazione adoperate per l'acquisizione dei dati archiviati sotto la chiave `V_IN`. Con riferimento alla procedura appena descritta, distinguiamo tre casi in base alla natura dei parametri preliminari che caratterizzano la prova, posto che la definizione dell'MRQ `V_IN_MRQ` contenga i riferimenti alle quantity che identificano i parametri preliminari della prova.

- Nel caso in cui sia sufficiente salvare grandezze già note, come ad esempio l'età (in ore) del motore o dell'ATS, è sufficiente utilizzare l'operazione di **Store Snapshot**, che corrisponde all'esecuzione atomica della quarta fase della procedura di *Steady State Measurement*. Tale comando trascrive semplicemente sotto la chiave indicata i valori correnti delle quantity coinvolte ed è eseguibile tramite l'apposito AO. Inoltre è disponibile anche in linguaggio VB per l'esecuzione da script - sia nel BSQ sia manualmente - tramite la Sub `MrqSnapshot` così definita:

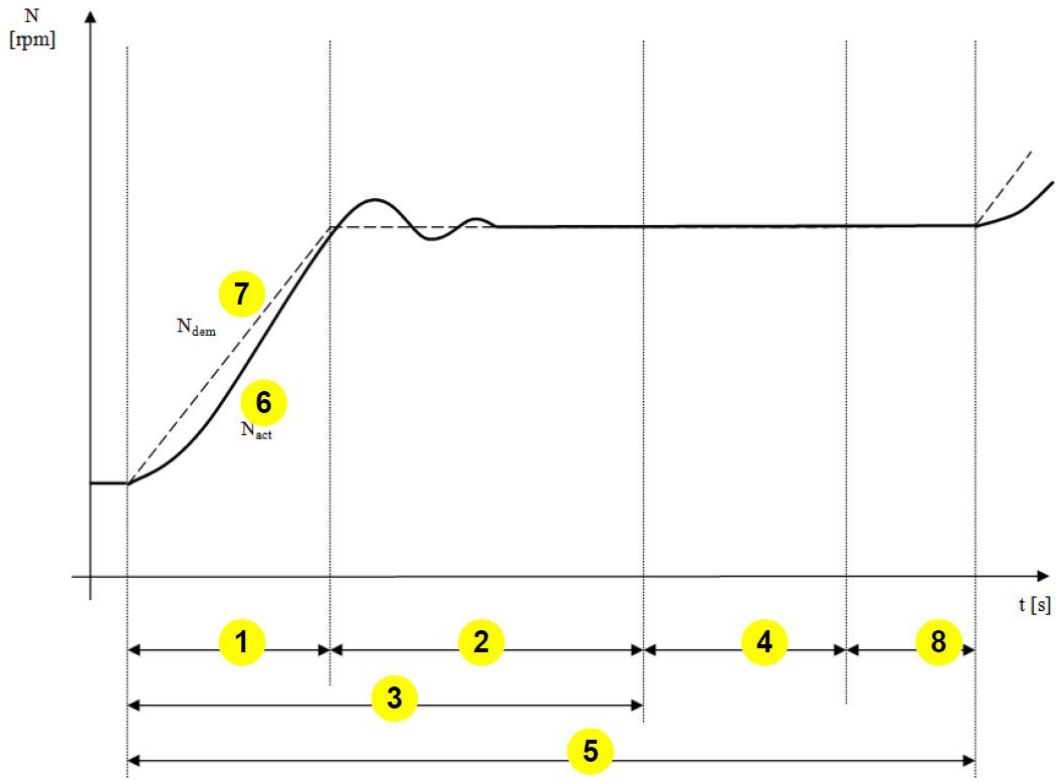

Figura 3.3: Struttura di uno step di un SSQ.

`Sub MrqSnapshot(Name, Key, WaitForCompletion, MonitorTimeout, Timeout)`
dove

  - `Name` è il nome della MRQ invocata
  - `Key` è il nome della chiave sotto cui archiviare i dati
  - `WaitForCompletion` è un flag tramite il quale è possibile eseguire la Sub in modo sincrono (valore 0) o asincrono (1), cioè attendendo o meno il completamento dell'operazione di store
  - `MonitorTimeout` è un flag che attiva (valore 1) o disattiva (valore 0) il monitoraggio tramite timeout dell'operazione
  - `Timeout` è l'intervallo di tempo di timeout, oltre il quale l'operazione è considerata fallita

- Nel caso in cui sia invece necessario effettuare un'acquisizione da uno strumento di misura o estrarre la media di determinati valori in un dato intervallo temporale, è necessario effettuare un'operazione di **Steady State Measurement completa**. L'esecuzione dei task di preparazione e calcolo sui vari strumenti è effettuata automaticamente da PUMA ed è sufficiente invocare nel BSQ l'AO di **Steady State Measurement With Storing** indicante l'MRQ di riferimento `V_IN_MRQ`, tra gli altri parametri (fig 3.4). Inoltre è disponibile anche in questo caso una Sub VB:
`MrqMeasureAndStore(Name, MeasTime, Key, MonitorTimeout, Timeout)`
dove

  - `Name` è il nome della MRQ invocata
  - `MeasTime` è il tempo di misura, nel quale sono raccolti i valori delle quantity coinvolte, da cui è infine estratta la media; questo valore sovrascrive quello indicato nella definizione della MRQ
  - `Key` è il nome della chiave sotto cui archiviare i dati
  - `MonitorTimeout` è un flag che attiva (valore 1) o disattiva (valore 0) il monitoraggio tramite timeout della misura
  - `Timeout` è l'intervallo di tempo di timeout, oltre il quale la misura è considerata fallita

- Nel caso in cui sia necessario controllare il motore in uno specifico punto operativo, è utilizzato il meccanismo di richiamo delle MRQ negli Step Sequence (SSQ) descritti precedentemente. A tale scopo sono disponibili specifiche **Step Buffer Variable** per modificare dinamicamente i parametri della MRQ richiamata per mezzo degli ASC (Assignment Script), descritte nella tabella 3.1.

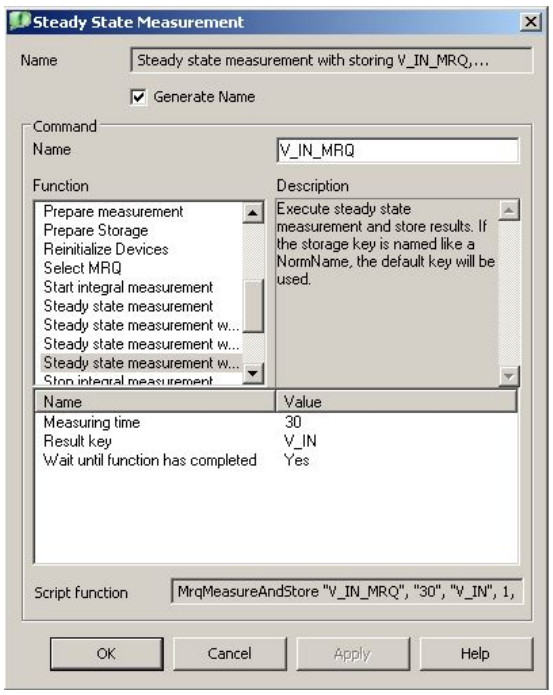

Figura 3.4: Activation Object per eseguire un'operazione di Steady State Measurement in un BSQ.

In ogni caso, infine, sono sempre presenti nella chiave `V_IN` le due quantity `Test_Type` e `TestBench`, che indicano rispettivamente la tipologia di test effettuato (stabilito nel TST) e il nome della sala prova di riferimento (parametrizzato nel SYS), secondo una nomenclatura standard adottata per questo progetto. L'utilità di memorizzare queste informazioni sarà esplicitata più avanti, in merito all'implementazione dei componenti software FileConverter, Parser e AITV.

3.1.2 EURINS AdaMo

Quanto descritto nel paragrafo precedente per l'automation system AVL PUMA Open, fa riferimento a metodologie frutto di concetti teorizzati e resi disponibili da AVL nella loro suite. La natura del sistema EURINS AdaMo è invece votata a uno spettro di applicazioni più ampio, quindi non solamente di tipo automotive. Le sue caratteristiche di flessibilità e granularità, che garantiscono di personalizzare molto le implementazioni, hanno permesso la realizzazione di strutture che costituissero una base comune a quella *embedded* di PUMA.
AdaMo Windows, ovvero il sottosistema *front-end* di AdaMo in esecuzione sul PC desktop, possiede un ambiente di configurazione *offline* ed un solo stato operativo online detto **Run**. In questo stato sono disponibili tutti i **canali logici** (corrispondenti alle quantity di PUMA), che possono essere per il sistema degli **input**, cioè segnali o valori leggibili ma non editabili in un normale *task* di

Parameters	MRQ buffer variable
Measuring time	Mbv_mtim
Result key	Mbv_rkey
Data storage tables	Mbv_dsts
Timeout	Mbv_tout
Pre-measurement (formulas)	Mbv_pref
Post-measurement (formulas)	Mbv_pof
Pre-storage (formulas)	Mbv_stof
Reference to used measurement evaluation definition	Mbv_eval
Reaction on measurement evaluation failure	Mbv_enok
Reaction on measurement failure	Mbv_mnok
Reaction on stabilization failure	Mbv_snok
Reference to used stabilization definition	Mbv_stab

Tabella 3.1: Le Step Buffer Variable che si riferiscono alla gestione di un MRQ in uno step di un SSQ.

AdaMo, o degli **output**, che sono invece leggibili ed editabili. Seguendo una classificazione basata sulla loro natura, i canali possono essere così suddivisi:

- **Costanti**, parametri di input configurabili offline (anche editabili online) suddivisi in parametri di banco (legati alla sala prova) e in parametri della UUT (legati alla unit under test); possono essere di tipo numerico o testuale.
- **Virtuali**, canali liberi e cioè non legati a dispositivi o sorgenti fisiche, ma che sono fondamentali per la realizzazione di architetture SW e per la programmazione dei test automatici; possono essere input o output numerici.
- **Manuali**, canali di input legati a finestre in cui l'utente può inserire appunto manualmente il valore desiderato per il canale; come le costanti, possono essere di tipo numerico o testuale.
- **Formula**, canali di input (poiché non sovrascrivibili nei task) numerici che associano un'espressione di calcolo a canali di input o output.
- **Real Time**, canali di input e output numerici gestiti dal sottosistema real time di AdaMo, in esecuzione sul CompactRIO; qualsiasi canale associato a sensori, strumenti o dispositivi di altra natura connessi al sistema AdaMo Real Time, sarà classificato come canale Real Time.
- **ASAP3**, canali di input associati alla comunicazione via protocollo ASAP [7] con l'application system che controlla la ECU.

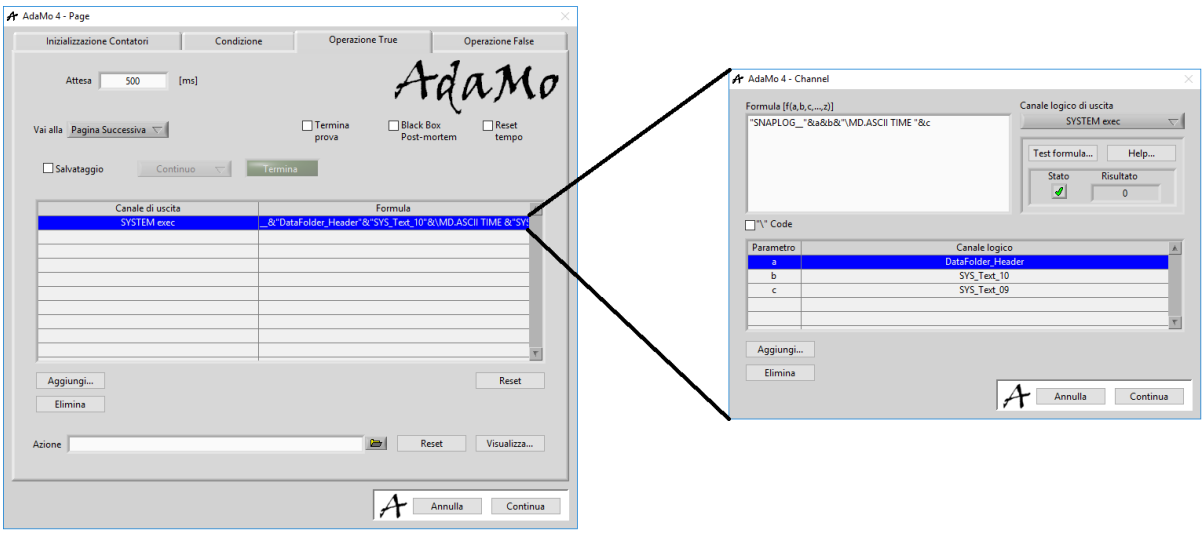

Figura 3.5: Pagina di un'Azione Generica di AdaMo.

- **RS232-TCP/IP**, canali di input e output, numerici e testuali, associati a strumenti di misura connessi ad AdaMo Windows, astratti da comandi di interrogazione implementati nei diversi protocolli, tipicamente AK [13].

A ciascuno dei canali di tipo numerico è possibile associare tre *properties*: un'unità di misura, il numero di decimali richiesti e una scala. Quest'ultima, descritta da un file `scl`, applica stabilmente una funzione (anche di grado maggiore di 1) al canale, i cui valori sono così dinamicamente trasformati. Inoltre può essere utile associare un canale di input ad un canale di output, in modo tale che il valore del primo sia scritto automaticamente sul secondo, con un periodo assicurato di 50 ms.
I valori di queste variabili possono essere salvati con diverse tecniche su un file testuale, dotato di un'intestazione parametrizzabile. Le metodologie di salvataggio sono tre:

- il salvataggio **continuo**, analogo al Recorder di PUMA, permette di acquisire dati in maniera continuativa con una frequenza selezionabile;
- il salvataggio **manuale**, analogo alla Steady State Measurement completa di PUMA, permette di acquisire una media dei valori in un dato intervallo temporale;
- il salvataggio **immediato**, che esegue una media temporale dei valori in un dato intervallo antecedente al comando di salvataggio.

Nello stato operativo di **Run** di AdaMo, è possibile effettuare operazioni e scrivere sui canali di output per mezzo dei *task* citati precedentemente. Essi prendono il nome di **Azioni** (files `act`) e possono avere un'esecuzione atomica o ciclica. Esistono 20 canali numerici speciali detti *contatori* utilizzati all'interno dell'azione

come "variabili d'appoggio". Ogni azione è suddivisa in diverse pagine, ovvero diversi step sintatticamente uguali, composti dalle seguenti porzioni eseguite sequenzialmente:

1. **Inizializzazione dei contatori**, con cui si assegna a ciascuno di essi - all'inizio dell'esecuzione dello step - un valore costante o quello di un canale logico; di default è associato il valore attuale del contatore;

2. **Condizione**, nella quale sono richiamati canali di qualsiasi tipo e che determina con il suo esito (True o False), l'esecuzione della relativa operazione;

3. **Operazione True**, eseguita in caso di Condizione verificata, consiste nelle seguenti operazioni facoltative fondamentali:

   - avvio o interruzione di un salvataggio continuo oppure esecuzione di un salvataggio manuale o immediato;
   - scrittura su canali di output di formule[2] scritte a partire da diversi canali logici; è possibile anche scrivere su canali di output testuali concatenando stringhe corrispondenti ad altri canali di tipo testuale;
   - attesa, prima del completamento dello step, del tempo indicato[3];
   - istruzione di controllo di flusso, per saltare ad uno step antecedente o successivo dell'azione, oppure per ciclare sullo stesso step.

4. **Operazione False**, sintatticamente del tutto uguale all'operazione True, eseguita in caso di Condizione non verificata.

Definita la struttura di un'azione, è opportuno ora fare una distinzione tra due tipologie. Le **Azioni RSTCP** (files `RSTCP.act`) permettono - nelle sezioni Operazione True e Operazione False - la scrittura su canali di output relativi a protocolli RS232-TCP/IP di comunicazione con strumenti di misura. Le **Azioni Generiche** (files `GEN.act`) invece permettono - nelle stesse sezioni - la scrittura su qualsiasi canale di output e permettono anche di invocare azioni RSTCP, di cui non attendono il completamento. Molto spesso infatti le azioni RSTCP hanno una struttura ciclica, grazie alla quale restano in esecuzione in *background* permettendo però all'Azione Generica chiamante di passare allo step successivo. Un canale di output testuale di notevole importanza è `SYSTEM exec`, tramite il quale è possibile lanciare comandi di vario genere al sistema, assegnandoli a questo canale di output. Molto utile per acquisire i dati della chiave `V_IN` è risultato in particolare il comando `SNAPLOG`, che permette di trascrivere istantaneamente il valore di alcuni canali su un file di testo qualsiasi, diverso da quello deputato

[2]Non si tratta di canali di tipo formula, ma calcoli implementati nello stesso *scope* dell'azione.

[3]Questo tempo non può essere inferiore a 50 ms per le Azioni Generiche.

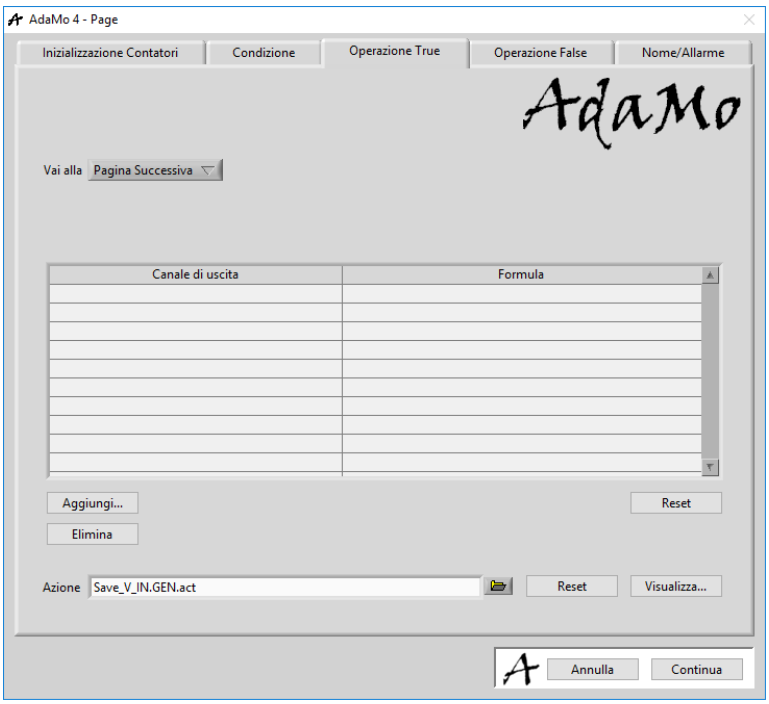

Figura 3.6: Pagina di una Sequenza di AdaMo.

a raccogliere i risultati del test, sul quale scrivono le classiche operazioni di salvataggio. Questo comando è dotato della seguente sintassi:
`SNAPLOG__<path log file>\t<freename00>\t<value00>\t<freename01>\t<value01>...`
dove

- `<path log file>` è il path assoluto del file di testo su cui vengono scritti i dati;
- `<freename**>` è una label d'intestazione a cui associare il valore indicato successivamente; è un testo libero quindi non dev'essere necessariamente uguale al nome di un canale;
- `<value**>` è il valore da associare alla label `<freename**>`; può essere scritto direttamente un valore costante numerico o testuale oppure è possibile utilizzare il valore attuale di un canale logico per mezzo della concatenazione di una stringa.

L'implementazione delle prove automatiche è effettuata per mezzo della scrittura di algoritmi chiamati **Sequenze** (files `prv`), sintatticamente simili alle azioni, composte anch'esse da una successione di pagine o step, ciascuno dei quali è strutturato in questo modo:

1. **Inizializzazione dei contatori**, del tutto analogo all'inizializzazione fatta nelle azioni;

2. **Condizione**, analogamente alle azioni, determina con il suo esito (True o False), l'esecuzione della relativa operazione;

3. **Operazione True**, eseguita in caso di Condizione verificata, consiste nelle seguenti operazioni facoltative:

   - scrittura sui contatori di formule numeriche scritte a partire da diversi canali logici;
   - esecuzione di un'Azione Generica, di cui si attende il completamento;
   - istruzione di controllo di flusso, per saltare ad uno step antecedente o successivo dell'azione, oppure per ciclare sullo stesso step.

4. **Operazione False**, sintatticamente uguale all'operazione True, ma eseguita in caso di Condizione non verificata;

5. **Caricamento Allarmi**, per caratterizzare lo step con delle soglie d'allarme ad hoc.

Va precisato che in generale il valore assunto dai contatori nell'esecuzione di uno step della sequenza sono poi i valori di partenza nel contesto dell'eventuale azione richiamata. In questo modo è possibile **passare dei parametri** dalla sequenza all'azione richiamata, **come se fosse una funzione**.
Per tanti aspetti è possibile trovare un'analogia tra la sequenza di AdaMo e il BSQ di PUMA e parallelamente tra le azioni di AdaMo e le SubRoutine di PUMA. Il concetto chiave della programmazione dei test su un qualsiasi sistema d'automazione di sala prova è dunque proprio l'organizzazione degli algoritmi in procedure **main** (sequenze o BSQ) che richiamano sotto-procedure (azioni o SubRoutine), con cui condividono i canali come variabili globali.
L'architettura appena descritta mette a disposizione tutti gli strumenti necessari a creare strutture per l'acquisizione dei parametri preliminari nella chiave `V_IN`, in modo del tutto analogo rispetto a quanto descritto per AVL PUMA Open. In particolare, è immediato intuire che il test sarà implementato come una sequenza, in cui uno dei primi step richiama un'azione deputata al salvataggio dei dati. Quest'ultima potrà essere implementata diversamente in base alle stesse tre categorie descritte nel paragrafo precedente:

- Nel caso in cui sia sufficiente salvare grandezze già note, l'azione è composta da un solo step nel quale è scritto sul canale `SYSTEM exec` il comando `SNAPLOG`, a cui sono passati come parametri label e valori corrispondenti ai canali coinvolti.

- Nel caso in cui sia invece necessario effettuare un'acquisizione da uno strumento di misura o estrarre la media di determinati valori in un dato intervallo temporale, bisogna anteporre allo step di `SNAPLOG` altri step che

richiamino le azioni RSTCP necessarie a porre in stato di misura gli strumenti coinvolti e ancora altri step funzionali al calcolo della media sui valori dei canali coinvolti.

- Nel caso in cui sia anche necessario controllare il motore in uno specifico punto operativo, bisogna inserire ulteriori step per il controllo del motore tramite i canali logici di output verso freno e pedale.

Anche in questo caso, infine, è memorizzata nella chiave `V_IN` una stringa indicante la tipologia di test effettuato, secondo la nomenclatura adottata per questo progetto. Differentemente da quanto fatto per PUMA, non è necessario "sprecare" una quantity - o, più propriamente per AdaMo, un canale logico - per questo scopo, ma è sufficiente utilizzare nel comando `SNAPLOG` la label `Test_Type` seguita dalla stringa indicante il nome della prova, senza dover concatenare un canale logico di tipo testuale. Al contrario la stringa indicante il nome della sala prova di provenienza, anch'essa registrata sistematicamente in `V_IN`, è memorizzata nella Costante di banco `TestBench`, chiamata esattamente come la relativa quantity di PUMA.

3.2 Archiviazione

I dati raccolti nel corso del test, per mezzo delle acquisizioni continue o stazionarie, sono riportati nel file di risultati che costituisce il prodotto della prova effettuata. La sua struttura è naturalmente definita contestualmente all'implementazione del test in sala prova, in base a modalità e parametrizzazione delle istruzioni di salvataggio inserite nell'algoritmo. Tuttavia è fondamentale che l'organizzazione delle chiavi di memorizzazione, in cui è suddiviso il **test result**, sia stabilita anche in virtù dello sviluppo di **layout** di post-elaborazione da cui ottenere il *report* della prova.
Nel paragrafo 1.2.2 è stata offerta una descrizione delle tipologie di chiave, che a questo punto è opportuno approfondire.

1. Le chiavi **Time Based**, sono frutto di acquisizioni continue (ad esempio il prodotto di un recorder di PUMA) di un determinato insieme di canali, tra cui c'è quello fondamentale della **base tempo**, con unità di misura *secondi* o *millisecondi*, che tendenzialmente prende il nome di `time` o `recorder_time`. Nel caso in cui ci siano più canali indicanti la grandezza tempo, è necessario che almeno uno abbia andamento monotono crescente, perché possa essere preso come riferimento. Su tale base può essere analizzata l'evoluzione temporale delle grandezze associate a tutti i restanti canali.
   Inoltre una chiave Time Based può presentare più mesurement ID, cioè più di una collezione di dati continuativa. Questo è l'effetto che si ottiene

avviando e interrompendo un recorder a più riprese: ogni measurement ID identifica dunque un intervallo di tempo in cui l'acquisizione è stata attiva. Un esempio appropriato è costituito dal ciclo omologativo WHTC [10], che è composto da due fasi in ciascuna delle quali è effettuata una registrazione continua dei canali necessari con frequenza pari a 10 Hz (un campione ogni 100 millisecondi). La chiave Time Based derivante da questa acquisizione è organizzata quindi in due measurement ID, uno per ciascuna fase.

2. Le chiavi **LogPoint Based**, derivano invece da misure stazionarie o acquisizioni puntuali con il metodo dello `Store Snapshot`. Se le chiavi Time Based sono adoperate soprattutto per effettuare acquisizioni quando il motore è sottoposto a transitori di giri e carico, queste chiavi sono invece utilizzate per acquisizioni in punti operativi stazionari e per questo motivo non è necessario individuare una base tempo. Ciascun measurement ID di una chiave logpoint-based può comunque disporre di più righe o - più precisamente - di più *punti*, ovvero acquisizioni successive che si è scelto di legare ad un solo measurement ID. É in realtà questo il caso più frequente per test di tipo *stazionario* come il già citato *Piano Quotato* o *Engine Map*: si adopera una chiave per memorizzare le misure stazionarie e sotto un solo measurement ID sono acquisiti tutti i punti della mappa effettuata.

É doveroso precisare, tuttavia, che la terminologia adottata per descrivere strutture è in realtà riferita alla convenzione adottata da AVL per i file di risultati derivati da sistemi PUMA Open e su cui si basa sostanzialmente l'achitettura di calcolo e presentazione grafica di Concerto. Come anticipato nel capitolo precedente, lo standard di riferimento per questo formato è ASAM ODS [3] [9] e ad esso si attiene l'architettura del database dei *test results* di PUMA, di cui ogni file di risultati costituisce un record, esportabile nel formato ASAM di trasporto ATF [14].
D'altro canto il sistema d'automazione AdaMo utilizza un file di testo per l'output dei dati raccolti, che possiedono un'intestazione seguita da una matrice composta da tante colonne quanti sono i canali posti in acquisizione e tante righe quanti sono i campioni raccolti. Durante un *salvataggio continuo* è aggiunta al file una riga alla volta con una velocità dipendente dalla frequenza impostata. Al contrario, nel caso di un *salvataggio manuale* è aggiunta una sola riga, al termine dell'intervallo di tempo in cui sono accumulati i dati per il calcolo della media. AdaMo consente inoltre di cambiare il file di risultati non solo manualmente, ma anche per mezzo di un'istruzione apposita da eseguire mediante la tecnica di scrittura sul canale `SYSTEM exec` descritta nel paragrafo precedente. Il comando ha la seguente sintassi:
`DATA FILE__[absolute path file *.ASCII]`
nel quale va indicato il path assoluto del nuovo file di risultati, anche mediante concatenazione stringhe utilizzando canali logici testuali contenenti porzioni del path.

In questo modo è possibile avviare acquisizioni continue a più riprese e memorizzare i dati in file separati e, facendo riferimento allo stesso esempio del ciclo omologativo WHTC, questo metodo permette dunque di associare ad ogni fase del test un proprio file.
Un discorso a parte va invece fatto per l'istruzione `SNAPLOG` menzionata a proposito della chiave `V_IN`, che produce un file di testo separato che nulla ha a che vedere con quello canonico - che possiamo chiamare *standard* - su cui AdaMo scrive i dati raccolti dai classici salvataggi continui o puntuali. Questo file si troverà al path indicato tra i parametri del comando ed è inoltre sprovvisto dell'intestazione: si presenta infatti come una matrice composta da tante colonne quanti sono i canali (o più precisamente le label) richiamati nel comando di `SNAPLOG` e tante righe quanti sono i campioni raccolti nelle esecuzioni consecutive dell'istruzione sullo stesso file di testo.
Il principio adottato nell'implementazione delle prove per l'organizzazione dei dati acquisiti segue una semplice analogia tra le chiavi di memorizzazione utilizzate da PUMA e i file di testo creati da AdaMo. In sintesi si può desumere che, data l'implementazione di una specifica prova, per ogni measurement ID di una chiave *time based* di PUMA c'è un file di testo (standard) di AdaMo in cui sono trascritti i dati acquisiti alla frequenza prestabilita; analogamente per ogni measurement ID di una chiave *logpoint* di PUMA c'è un file prodotto da una o più istruzioni `SNAPLOG`.
Si desume da questa descrizione che l'unica via per ottenere una post-elaborazione unificata e neutra rispetto all'automazione di provenienza è la ricerca di una struttura comune ai due sistemi PUMA e AdaMo. Questo rappresenta inoltre l'unico modo per garantire un'archiviazione centralizzata di tutti i test results.
Se da un lato abbiamo a disposizione su PUMA un database strutturato e ben predisposto per l'implementazione di piattaforme di calcolo su Concerto, dall'altro lato AdaMo garantisce una comoda organizzazione dei risultati della prova in vari file di testo separati. Molto spesso infatti è fondamentale poter accedere facilmente ai dati raccolti per mezzo di software per la creazione di fogli di lavoro come Microsoft Excel, che permette di implementare molto rapidamente dei calcoli sulle colonne che compongono il file. Tuttavia nell'ottica di fruire di layout di elaborazione pre-impostati e sviluppati di pari passo con l'algoritmo di test in sala prova, è preferibile compattare i dati in una struttura più maneggevole per la programmazione.
Per questo motivo siamo in definitiva interessati a trasformare la collezione di file di testo prodotti da AdaMo, nel corso dell'esecuzione di una prova, in un **unico file**, il più possibile simile al record ASAM ODS prodotto da PUMA nel corso dell'esecuzione della stessa prova. A fare questo lavoro di raggruppamento e creazione di un file unico per il test result di AdaMo è il **FileConverter**, un componente software realizzato appositamente per questo scopo, sviluppato nel linguaggio di programmazione specifico di Concerto [2], che integra anche uno script ausiliario realizzato in linguaggio Python.

Un elemento di notevole rilevanza per questo progetto, in particolare per i componenti architetturali descritti in questo paragrafo, è il Datasource di Concerto, che consiste in un *data provider* definito dalle seguenti caratteristiche:

- un **nome** univoco a cui fare riferimento nella scrittura di codice relativo a formule e script;
- un **Alias** di default, cioè il nome da associare ai file di questo Datasource una volta aperti, che presenta un suffisso indicante un indice progressivo (per esempio se si sceglie `ASCFILE`, il primo file aperto avrà Alias `ASCFILE1`, il secondo `ASCFILE2` e così via seguendo l'ordine di apertura);
- un **formato** di riferimento, che può essere associato a un tipo di server (è l'esempio dei server ASAM ODS delle sale PUMA) o a un tipo di file accessibile da filesystem;
- le **regole di accesso** al file, cioè indirizzo e parametri di connessione in caso di server (come quelli delle sale PUMA) o path di partenza nel caso di file accessibili da filesystem locale o remoto;
- aree su filesystem da cui attingere le **formule** da associare automaticamente ai file del Datasource (sono naturalmente consentiti riferimenti sia a file di formule Concerto con estensione `frm`, sia file di formule Python con estensione `py`);
- regole di **traduzione** di nomi di chiavi e canali, tramite istruzioni di mapping eventualmente importabili da file chiamati **Dictionary**;
- definizione di **script context** che associano al Datasource una libreria di script utilizzabili a più livelli.

Il **Data Explorer** è l'ambiente di Concerto che permette non solo di creare e gestire i Datasource, ma soprattutto di utilizzarli per accedere ai dati. Inoltre è possibile raggruppare diversi Datasource in un **Data Environment**, descritto da un file `dxv`. Infine nel **Work Environment** in uso, cioè appunto l'ambiente di lavoro attuale di Concerto, è possibile caricare più Data Environment per accedere a tutti i Datasource che ciascuno di essi contiene. Per questo progetto è stato creato appositamente un Work Environment, chiamato **AITV**, e sono stati creati - all'interno di un Data Environment - diversi Datasource, che saranno introdotti nei prossimi paragrafi.
Nel prossimo paragrafo 3.2.1, sarà descritto nel dettaglio il funzionamento del **FileConverter** mentre nel successivo paragrafo 3.2.2 sarà illustrato il metodo di archiviazione centralizzato adoperato per convogliare tutti i dati ottenuti in una struttura unica.

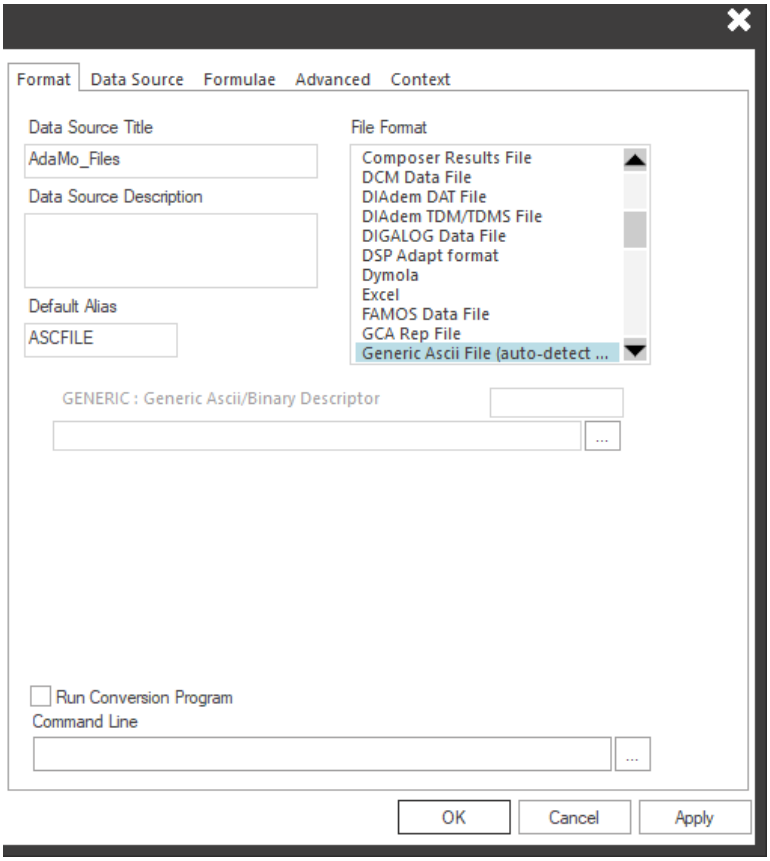

Figura 3.7: Configurazione di un DataSource nel DataExplorer di Concerto.

3.2.1 FileConverter

La collezione di file di testo prodotti da AdaMo costituisce il punto di partenza per la conversione, adoperata dal **FileConverter**, in un file unico e facilmente comparabile con i file di PUMA. Per maggiore chiarezza, è a questo punto opportuno considerare come *test result* di AdaMo l'intera **directory** che ospita i file creati durante l'esecuzione del test. Il numero e i nomi di questi documenti testuali variano in base alla tipologia di prova effettuata e sono sono determinati, come descritto precedentemente, con una duplice prospettiva sull'implementazione sia del test in sala prova sia del layout di post-elaborazione.
Lo scopo di questo componente software è quindi effettuare una conversione della folder di risultati di AdaMo in un file ATF [14], per mezzo dei metodi messi a disposizione dalla suite di scripting di Concerto [2]. L'analogia tra i file testuali e le chiavi di memorizzazione definita nell'implementazione del test sui due sistemi d'automazione, viene dunque tradotta in una corrispondenza univoca peculiare per ogni tipo di test. Da qui l'idea di trascrivere la lista di chiavi necessarie in un **file di mapping**, specifico della prova, da passare come input al FileConverter, in modo tale che possa conservare la flessibilità necessaria ad adattarsi ai diversi scenari.
In generale è frequente che alcune necessità, specifiche dell'attività di ricerca in corso, richiedano l'acquisizione di dati particolari che non rientrano in quelli solitamente salvati dalla prova. Questo non comporta obbligatoriamente la creazione di varianti "ufficializzate" dell'algoritmo di test e dei layout di elaborazione, ma al contrario spesso si riconduce a richieste saltuarie per le quali è sufficiente permettere agli *application engineer* di valutare tali dati acquisiti. Per queste grandezze sono solitamente adoperate chiavi specifiche, da escludere quindi dall'operazione di traduzione da file di testo AdaMo a chiavi di memorizzazione del file ATF. In realtà accade molto più frequentemente che questi canali siano inseriti nei recorder canonici del test, il che li rende disponibili anche in fase di post-elaborazione, sebbene non siano dati adoperati dalle formule predisposte per elaborare il test.
Per poter accedere alle funzionalità di scripting di Concerto è necessario disporre della feature aggiuntiva di licenza **Concerto Application Development Toolbox**. Questa piattaforma di programmazione permette di implementare formule (file `frm`) complesse sui canali presenti nelle chiavi dei file caricati e allo stesso tempo permette di creare script (file `csf`) per gestire l'intera applicazione, dai Datasource alle finestre del layout. Il linguaggio di programmazione Concerto è molto simile a Visual Basic e include alcuni elementi di C, inoltre la versione in uso include un supporto per formule e script realizzabili in linguaggio Python.
Nell'ambiente di programmazione di Concerto, ogni variabile, cioè ogni set di dati manipolabile, è organizzato internamente come un **Dataset**, cioè una *data structure* simile al tipo *Variant* di Visual Basic, ma più potente. Di conseguenza afferisce al tipo **Dataset** ogni variabile in formule e script, così come ogni canale

X	Y	I
1	27,46	0
2	30,72	0
3	32,02	0
4	34,85	0
5	36,48	0
6	38,26	0

Tabella 3.2: Esempio di dataset proveniente da una chiave Logpoint-based.

(o quantity) appartenente a un file accessibile nel Data Explorer, non appena è caricato.
I Dataset possono consistere in uno o più *punti*, potendo quindi essere definiti rispettivamente *scalari* o *vettori*. In particolare il compilatore delle formule processa i dataset senza considerare di quanti punti siano composti e ciascuna operazione applicata al dataset è automaticamente applicata a tutti i suoi punti. Questo approccio garantisce efficienza e fluidità nella computazione, senza incorrere in loop pesanti e dispendiosi per tempo e risorse. Va comunque posta attenzione nell'utilizzo di più dataset con un diverso numero di punti: la somma di dataset di dimensioni diverse non è consentita, ma vettori e scalari possono comunque essere combinati. In particolare se si somma un dataset di dimensione maggiore di 1 (vettore) e un dataset dotato di un solo punto (scalare), il valore dello scalare è sommato a ciascun elemento del vettore.
Un dataset inoltre non consiste semplicemente in un *array*, ma dispone di tre tracce (un esempio in tabella 3.2):

- **x-trace**, che nel caso di canali appartenenti a chiavi logpoint-based è un array di numeri interi progressivi, nel caso di una chiave time-based è un valore di tempo (in secondi o millisecondi); può tuttavia appartenere anche a qualsiasi altro tipo, anche String;

- **y-trace**, che rappresenta il vero valore del canale e consiste in un array di valori numerici o anche di tipo testuali (che chiaramente non possono essere utilizzati per calcoli, ma solo per manipolazioni sul tipo String);

- **marker trace**, che contiene un array di indicatori di vario genere, i quali possono ad esempio indicare se il corrispondente punto è valido o meno (nel caso in cui in un dataset coinvolto in una formula o script ci sia un punto invalido, l'interprete opererà comunque sul punto, ma il corrispondente punto nel dataset risultante sarà marcato anch'esso come invalido).

Il FileConverter consiste essenzialmente nello script `FileConverter.csf` in linguaggio Concerto, che richiama un piccolo script Python ausiliario per la gestione delle intestazioni nei file standard di AdaMo. La sua struttura, schematizzata in fig. 3.8, è presentata di seguito.

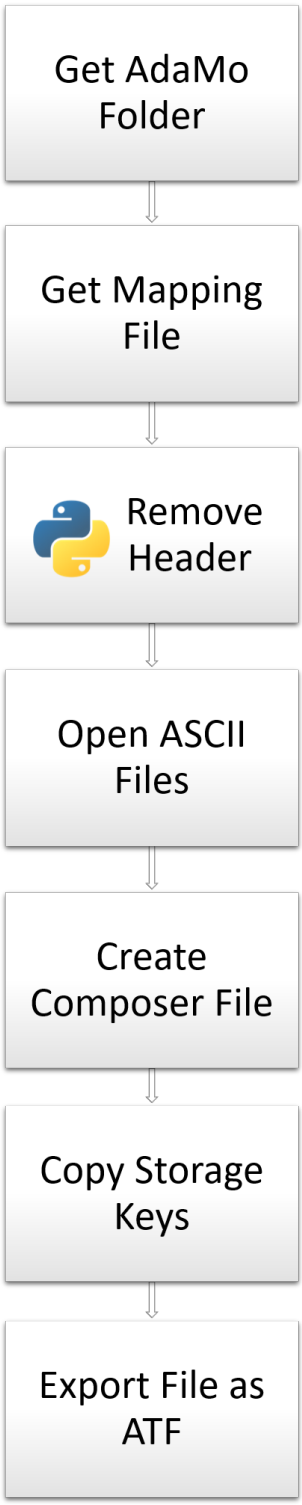

Figura 3.8: Struttura del FileConverter.

1. Innanzitutto è necessario **ottenere il path della folder AdaMo** su filesystem locale o condiviso. I PC desktop che ospitano il client Windows di AdaMo - così come i PC di PUMA - sono dotati di una scheda di rete dedicata alla connessione alla LAN aziendale e questo garantisce di poter accedere a tutti i test results delle sale prova. Inoltre sono disponibili risorse di archiviazione condivise su cui sono solitamente caricati dati e documenti perché siano fruibili da tutti gli utenti.
   Questa porzione di codice, realizzata in linguaggio Concerto, adopera per la selezione del path il metodo `DirDialog` della classe built-in `Application`, che presenta molti metodi e attributi utili per l'interazione con l'utente e la gestione dei vari ambienti di Concerto. In particolare il metodo utilizzato permette la selezione di una folder per mezzo dell'apertura di una finestra di navigazione e ha la seguente sintassi:
   `DirDialog([InititalDir], [Caption])`
   i cui parametri sono così descrivibili:

   - `InititalDir` (opzionale), di tipo `String`, è il path iniziale per la finestra di ricerca nel filsystem;
   - `Caption` (opzionale), di tipo `String`, è il titolo della finestra visibile sul top.

   Il metodo restituisce infine un valore di tipo String, indicante il path selezionato dall'utente dopo la navigazione.
   Il codice implementato per questa fase è il seguente:

```
App = GetApplication()
\\É istanziato l'oggetto "App" della classe Application
SelectedDir = App.DirDialog(startPath, "Select AdaMo result
folder")
\\Nella variabile "SelectedDir" è salvato il path assoluto
\\della directory selezionata dall'utente
```

   In merito alla variabile `startPath`, va fatta una precisazione sulle aree di storage da cui attingere. La fase successiva eseguita dal FileConverter consiste nell'apertura e caricamento in ambiente Concerto di tutti gli ASCII file, presenti nella folder selezionata, che facciano riferimento alla lista di chiavi necessarie per l'elaborazione del test, riportata nel file di mapping. Potrebbe essere sufficiente utilizzare il path ottenuto dal metodo `DirDialog` sopra descritto, ma questo comporterebbe che in fase di apertura l'utente sia costretto a scegliere il formato di questi file, lasciando di fatto una possibilità d'errore. Per ovviare a quest'eventualità è stato creato un apposito Datasource, chiamato `AdaMo_Files`, che associa ad un'area del filesystem di una risorsa di archiviazione condivisa, il formato *ASCII con valori delimitati da tabulazione* tipico dei file prodotti da AdaMo. In

questo modo è possibile effettuare direttamente l'apertura evitando che l'utente selezioni erroneamente un formato sbagliato. Di conseguenza il path di partenza, assegnato come valore alla variabile `startPath` passata come parametro `InitialDir` al metodo `DirDialog`, è un valore di tipo String corrispondente alla locazione dell'area target del Datasource.
Si desume che le folder di test risults vanno posizionate dal filesystem della sala prova di provenienza a quello della risorsa di archiviazione condivisa nella rete aziendale, perché sia disponibile per la conversione.

2. Lo step successivo consiste nella **selezione del file di mapping** relativo alla tipologia di prova eseguita. Tra i parametri salvati nella chiave `V_IN` c'è sempre la quantity `Test_Type` che indica il tipo di prova eseguita, secondo una nomenclatura standard adottata per questo progetto. Per questa ragione il file relativo alla chiave `V_IN` è il primo ad essere aperto, già in questa fase, per poter leggere il valore di `Test_Type` salvato. Tale stringa è fondamentale per la selezione del file corretto, il cui nome contiene per l'appunto il titolo del tipo di test effettuato: ad esempio, per la prova di *Lambda Step* a cui si riferisce il caso di studio approdondito nel capitolo 4, il valore scelto per la variabile `Test_Type` è "LStep" e di conseguenza il file di mapping corrispondente ha come nome *LStep_ Mapping.txt*.
Precedentemente è stata illustrata la necessità di far riferimento al Datasource definito per i file nelle folder di AdaMo (`AdaMo_Files`). Per poter sfruttarne le caratteristiche descritte, è necessario far riferimento al path relativo della cartella selezionata dall'utente nella fase 1, rispetto alla directory target del Datasource. Per questo motivo la stringa restituita dal metodo `DirDialog` è manipolata con la funzione built-in `StrErase` della libreria **String**, che cancella porzioni di stringhe ed è così definita:
`StrErase(str, [pos], [count])`
i cui parametri sono così descrivibili:

   - `str`, di tipo `String`, è la stringa da manipolare;
   - `pos` (opzionale), di tipo `Integer`, è l'indice di partenza per la *sub-string* da cancellare (se è omesso, è usato il valore di default 1);
   - `count` (opzionale), di tipo `Integer`, è il numero di caratteri che compone la *sub-string* da cancellare (se è omesso, sono cancellati tutti i caratteri da `pos` fino alla fine della stringa `str`).

   La funzione restituisce infine la stringa risultante.
   Nel nostro caso è stato necessario estrapolare dalla variabile `SelectedDir` la porzione di path relativo a partire dalla directory di partenza del Datasource, cioè il valore della variabile `startPath`. Il codice per la manipolazione della stringa è il seguente:

```
dataDir=StrErase(SelectedDir,1,StrLen(startPath))
```

```
\\Nella variabile dataDir è salvato il path relativo
```

La funzione `StrLen(str)` restituisce la lunghezza (`Integer`) della stringa `str`.
Per l'apertura del file relativo alla chiave `V_IN` è necessario innanzitutto istanziare un oggetto della classe `File` adoperando il metodo `SelFile` della libreria **Script**, così definito nel caso di selezione per mezzo di un Datasource esistente:
`SelFile("DataSource", "Filename")`
oppure
`SelFile("DataSource\Filename")`
i cui parametri sono così descrivibili:

- `DataSource`, di tipo `String`, è il nome del Datasource di riferimento;
- `Filename`, di tipo `String`, è il path relativo del file.

La funzione restituisce l'oggetto `File` corrispondente al file selezionato.
Successivamente, per l'apertura e il caricamento in Concerto del file, è adoperato il metodo `Open([Alias])` della classe `File`, che ha come unico parametro (opzionale) una stringa che indica l'Alias desiderato da associare al file una volta aperto. Nel caso in cui il parametro è omesso, è utilizzato l'Alias di default per il Datasource, dotato del suffisso indicante l'indice progressivo. Infine il metodo restituice un valore intero (0 o 1) che rappresenta in sostanza un flag indicante la corretta apertura (1) o un errore (0).
Il codice per l'apertura del file relativo alla chiave `V_IN` è il seguente:

```
a1=SelFile("AdaMo_Files"+dataDir+"\V_IN.ASCII")
\\"a1" è un oggetto della classe File
A1=a1.Open("V_IN")
\\L'Alias
selezionato è "V_IN".

\\"A1" è flag che indica l'esito dell'operazione di apertura.
```

Una volta aperto il file *V_IN.ASCII* con Alias `V_IN`, sono automaticamente disponibili tutti i canali in esso archiviati sotto la chiave `D`. Di conseguenza è possibile assegnare alla variabile `T_Type_ds` il dataset corrispondente al canale `Test_Type` in chiave D del file con Alias `V_IN`, con la seguente sintassi:
`T_Type_ds = V_IN:D'Test_Type`
che segue il costrutto tipico `Alias:KEY'DATASET` adoperato in linguaggio Concerto per accedere ai canali.
In base a quanto specificato nei paragrafi precedenti, il dataset risultante - come l'intera chiave `V_IN` - potrebbe non essere dotato di un solo punto ed è perciò necessario estrarre la stringa indicante il tipo di test da un solo

punto (il primo) del dataset `T_Type_ds` con il metodo `y[index]` della classe `Dataset`, che permette di ottenere o modificare il valore corrispondente all'indice `index` sulla *y-trace* del dataset:
`T_Type = T_Type_ds.y[1]`
In questo modo il valore y corrispondente all'indice 1 del dataset `T_Type_ds` è assegnato alla variabile `T_Type`.
Successivamente è aperto il file di mapping corrispondente alla tipologia di prova eseguita, che è posizionato nella sub-folder **lib** della directory target del Datasource `AdaMo_Files`, insieme a tutti quelli relativi agli altri test. Le funzioni per la selezione e l'apertura del file di mapping sono le stesse già adoperate per `V_IN`.
Dal file aperto sono infine estratti i due dataset `Lkeys` e `Rkeys`, che contengono rispettivamente le chiavi Logpoint-based e quelle Time-based indicate nel file di mapping sotto i canali `LogKeys` e `RegKeys`. Dopo aver salvato nelle due variabili le liste di chiavi di interesse, il file di mapping è chiuso con il metodo `Close()` della classe `File`.
Di seguito è riportato il codice implementato per questa fase:

```
a1=SelFile("AdaMo_Files"+dataDir+"\V_IN.ASCII")
A1=a1.Open("V_IN")
\\Apertura del file relativo alla chiave "V_IN"
T_Type_ds = V_IN:D'Test_Type
T_Type = T_Type_ds.y[1]
\\Salvata in "T_Type" la stringa indicante il tipo di prova
a2=SelFile("AdaMo_Files"+"\lib\"+T_Type+"_Mapping.txt")
A2=a2.Open("ASCFILE1")
\\Apertura del file di mapping
LKeys=ASCFILE1:D'LogKeys
RKeys=ASCFILE1:D'RegKeys
\\Salvataggio in "LKeys" e "RKeys" delle due liste di chiavi
a2.Close()
\\Chiusura del file di mapping.
\\L'Alias "ASCFILE1" torna disponibile.
```

In questa fase è inoltre estratto dal dataset `TestBench` del file `V_IN` la stringa indicante la sala prova in cui è stato effettuato il test. Questa informazione è fondamentale per determinare il path di export del file ATF risultante dalla conversione, come sarà descritto nell'ultimo step. Tale stringa è assegnata alla variabile `TBench` in maniera analoga a quanto fatto per `T_Type`:
`TBench_ds = V_IN:D'TestBench`
`TBench = TBench_ds.y[1]`

3. Prima di procedere all'apertura dei file relativi alle chiavi indicate nelle variabili `LKeys` e `RKeys`, è necessario **rimuovere gli header** tipici dei file di tipo time-based creati da ogni tipo di test eseguito sul sistema AdaMo. Nel paragrafo 3.1.2 è stato specificato che i file relativi a chiavi logpoint-based di AdaMo sono prodotti dall'istruzione `SNAPLOG` e che non presentano alcun header. Al contrario i file legati a chiavi time-based sono frutto di acquisizioni continue che trascrivono i dati sul file di testo in codifica ASCII, che costituisce l'output standard di AdaMo e che presenta un'utile intestazione di 10 righe. In questo caso tuttavia i metadati contenuti nell'intestazione possono essere trascurati, poiché non saranno adoperati da questo punto in avanti nel nostro sistema. Inoltre per la corretta apertura di questi file in Concerto e per attingere ai dati sotto forma di chiavi time-based, è opportuno rimuovere le 10 righe iniziali prima di lanciare le istruzioni di apertura già viste in precedenza. Per effettuare questa operazione, che si rende dunque necessaria esclusivamente per i file indicati dalla variabile `RKeys`, è stato adoperato uno script ausiliario realizzato in linguaggio Python chiamato **RegHandler**. Per invocarne l'esecuzione, è stata utilizzata la funzione built-in `ExecutePythonScript(ScriptPath)` della libreria **Script**, che permette di eseguire uno script Python passando come parametro il suo path assoluto. Il file *RegHandler.py* è stato posizionato nella sub-folder **lib** della directory target del Datasource `AdaMo_Files`, come altri script che saranno descritti nei prossimi paragrafi. Di conseguenza è stata adoperata la seguente sintassi per l'esecuzione del `RegHandler`:
`ExecutePythonScript(startPath+"\lib\RegHandler.py")`
adoperando anche in questo caso la variabile `startPath`.
Prima di procedere alla descrizione del codice Python realizzato e dei motivi per i quali è stato ritenuto opportuno avvalersi di questo linguaggio, è bene fare una breve introduzione ad uno dei metodi più efficaci per condividere variabili tra diversi contesti di Concerto.
Le variabili "ordinarie", utilizzate in formule e script Concerto, hanno ambito di visibilità - il cosiddetto *scope* - locale ed esistono solo nel periodo d'esecuzione. Concerto mette a disposizione tuttavia anche delle variabili speciali, le **User Variable**, che al contrario hanno scope globale e mantengono il loro contenuto durante la sessione di Concerto. Si possono definire da un apposito menu e sono riportate all'interno del file *uservariables.ini*, hanno validità assoluta e prescindono quindi dal Work Environment corrente. Possono essere di tipo numerico o testuale e il loro nome generalmente inizia con il carattere `%`. Si richiamano generalmente utilizzando direttamente il loro nome, ma in realtà è possibile creare User Variable il cui nome non inizia con il carattere `%` e in tal caso vanno adoperate le funzioni `GetUserVar` and `SetUserVar` della libreria **Auxiliary**. Infine è possibile vincolare l'esistenza di una User Variable al Work Environment corrente e per farlo è sufficiente utilizzare il prefisso `%CWF_` nel nome della variabi-

le. Sono numerose le utilità di queste peculiari strutture dati ed è stata in particolare sfruttata la loro caratteristica visibilità globale per passare dei parametri tra i diversi script adoperati (soprattutto se realizzati in linguaggi diversi), non solo nel caso del FileConverter che si sta analizzando in questo paragrafo.
Tornando a porre l'attenzione sullo step di rimozione degli header dai file time-based, il richiamo allo script Python è effettuato ciclicamente in un loop che scorre ogni elemento del dataset `RKeys`. Per ciascuno di essi, il `RegHandler` seleziona il file corrispondente ed elimina l'header. Si deduce facilmente che risulta necessario passare due parametri dallo script Concerto allo script Python: la directory selezionata dall'utente e il nome del file attuale, nel corpo del loop. Per fare questo ci si è avvalsi di due User Variable, entrambe legate al Work Environment attualmente in uso (**AITV**):

- `%CWF_AdaMoFilePath`, contenente il path assoluto della folder selezionata dall'utente;
- `%CWF_AdaMoFile`, contenente il nome del file da manipolare.

Si tenga a mente però che in linguaggio Python non è possibile inserire nelle stringhe direttamente il carattere "\" poiché è un carattere speciale[4], ma va preceduto sempre da un altro "\". Per questo motivo la stringa contenuta nella variabile `SelectedDir` dev'essere modificata di conseguenza, per mezzo della function `StrReplaceAll(str, String1, String2)`, che sostituisce `String2` alla sub-string `String1` di `str`. Il codice per il corretto assegnamento del path modificato a `%CWF_AdaMoFilePath` è il seguente:

```
dataDirPy = StrReplaceAll(dataDir,"\","\\")
origPath=%CWF_AdaMoFilePath
%CWF_AdaMoFilePath=%CWF_AdaMoFilePath+dataDirPy
```

Il valore iniziale della variabile `%CWF_AdaMoFilePath` è il path della directory target del Datasource `AdaMo_Files`, già adattato al linguaggio Python raddoppiando ogni carattere \. Perciò è sufficiente concatenare il path relativo contenuto in `dataDirPy`. Ad ogni modo il path originale è stato salvato nella variabile `origPath` in modo tale che al termine dello script `FileConverter.csf` possa essere reimpostata la User Variable `%CWF_AdaMoFilePath` per mezzo dell'istruzione:
`%CWF_AdaMoFilePath=origPath`
Il codice che effettua il loop per la rimozione degli header è dunque il seguente:

[4]Il *backslash* è adoperato nelle stringhe Python per i caratteri di escape, come `\n` (new line) o `\t` (tab), ma anche per altri caratteri speciali.

```
for i=1 to nPoints(RKeys)
     if StrUpper(RKeys.y[i]) <> "NULL" then
          %CWF_AdaMoFile = RKeys.y[i]
          ExecutePythonScript(startPath+"\lib RegHandler.py")
     endif
next i
```

Si noti che per costruire il loop sono stati adoperati il *for statement* di Concerto e la funzione `nPoints(DS)` della libreria **Auxiliary**, che restituisce il numero di punti nel dataset `DS`. Inoltre le istruzioni di assegnamento alla User Variable e di esecuzione dello script Python sono inglobate in un *if statement*, che permette di effettuare tali operazioni solo nel caso in cui il valore `RKeys.y[i]` sia diverso da "NULL" (il confronto è reso *case insensitive* dalla funzione `StrUpper(str)` che rende maiuscolo ogni carattere della stringa argomento `str`). Questo controllo è reso necessario dall'eventualità di avere un numero diverso di chiavi logpoint-based rispetto a quelle time-based. In tal caso è assicurato una dimensione uguale per i due dataset, per mezzo di un'operazione di *padding* che aggiunge la stringa "null" (case insensitive) nei punti sprovvisti di un valore effettivo.
Possiamo ora analizzare nel dettaglio lo script *RegHandler.py*. Uno dei grandi vantaggi che l'utilizzo di Python garantisce è sicuramente la possibilità di importare moduli che implementano strutture e funzioni utili agli scopi del codice che si vuole realizzare. I moduli disponibili sono molto numerosi e facilmente installabili con il sistema di gestione dei pacchetti standard **pip**[5]. In particolare per questo script è stato necessario importare i seguenti moduli:

- **csv**[6], che implementa classi utili per leggere e scrivere dati tabellari in formato CSV;
- **os**[7], che mette a disposizione metodi portabili per utilizzare funzionalità dipendenti dal sistema operativo.
- **concerto**, fornito da AVL come parte integrante del supporto allo scripting Python in Concerto, estende molte delle funzionalità base del linguaggio Concerto.

Le istruzioni di import utilizzate sono le seguenti:

```
import concerto as conc
import csv
import os
```

[5] urlhttps://docs.Python.org/3/installing/index.html
[6] `https://docs.Python.org/3/library/csv.html`
[7] `https://docs.Python.org/3/library/os.html`

Nello specifico in *RegHandler.py* è stata implementata la funzione `Remove-Header(f_path)` che apre il file indicato dal parametro `f_path` e lo copia riga per riga - saltando solo le prime 10 - su un file temporaneo, che infine è sostituito al file originale. Gli step eseguiti sono i seguenti:

(a) il percorso in `f_path` è copiato in `f_in`, mentre ad `f_out` è assegnato il percorso del file temporaneo utilizzando il metodo `replace` della stringa `f_in`;

(b) sono aperti i file indicati da `f_in` e `f_out` (quest'ultimo è contestualmente creato, dato che non esisteva precedentemente) e sono istanziati i due oggetti corrispondenti `inp` e `out`;

(c) sono istanziati gli oggetti `writer` e `reader` del modulo `csv` con i quali è realizzato un loop che scorre le righe del file `inp` e, fatta eccezione per le prime 10, le trascrive sul file `out`;

(d) i file `inp` e `out` sono chiusi con il loro relativo metodo `close()`;

(e) il file indicato dal path `f_in` è rimosso con la funzione `remove()` del modulo `os` e infine il file `f_out` è rinominato con il nome originale contenuto in `f_in`, per mezzo della funzione `rename(f_out,f_in)` del modulo `os`.

Il codice della funzione è di seguito riportato:

```
def RemoveHeader(f_path):
     f_in=f_path
     f_out=f_in.replace(".ASCII","_temp.ASCII")
     with open(f_in, 'rt') as inp, open(f_out, 'wt', newline='')
as out:
          writer = csv.writer(out)
          for i,row in enumerate(csv.reader(inp)):
               if i>=10:
                    writer.writerow(row)
          inp.close()
          out.close()
          os.remove(f_in)
          os.rename(f_out,f_in)
          return 1
```

Infine la funzione appena definita è invocata passando come parametro il path del file da manipolare. A tale scopo sono adoperate le due User Variable `%CWF_AdaMoFilePath` e `%CWF_AdaMoFile`, richiamate con la struttura `variables` del modulo `concerto`. Quest'ultima consiste in un *dictionary* contenente le coppie nome-valore di tutte le User Variable attualmente presenti nella sessione di Concerto, che sono così leggibili e modificabili anche

in ambiente Python.
Il codice per l'ottenimento del path del file da modificare e il seguente richiamo della funzione `RemoveHeader(f_path)` è il seguente:

```
base_path = conc.variables["%CWF_AdaMoFilePath"]
file_name = conc.variables["%CWF_AdaMoFile"]
file_path = base_path + "\\" + file_name + ".ASCII"
RemoveHeader(file_path)
```

Anche in questo caso è stato adoperato il doppio carattere \ nella costruzione della stringa indicante il path del file, dal momento che nel linguaggio Python \ è un carattere speciale delle stringhe.

4. Analogamente a quanto fatto per il file `V_IN`, sono ora **aperti tutti i file** i cui nomi sono riportati nelle variabili `LKeys` e `RKeys`, attingendo dalla directory selezionata dall'utente (`SelectedDir`). Anche in questo caso è dunque stata utilizzata la funzione `SelFile` per individuare ciascun file attraverso il Datasource `AdaMo_Files` e ottenerne il corrispondente oggetto della classe `File`; successivamente è stato adoperato il metodo `Open` per l'apertura del file. Queste operazioni sono state eseguite all'interno di due *cicli for*, che scorrono i componenti dei dataset `LKeys` e `RKeys`:

```
for i=1 to nPoints(LKeys)
     if StrUpper(LKeys.y[i]) <> "NULL" then
          a2=SelFile("AdaMo_Files"+dataDir+"\"+LKeys.y[i]+
".ASCII")
          A2=a2.Open(LKeys.y[i])
     endif
next i
for i=1 to nPoints(RKeys)
     if StrUpper(RKeys.y[i]) <> "NULL" then
          a2=SelFile("AdaMo_Files"+dataDir+"\"+RKeys.y[i]+
".ASCII")
          A2=a2.Open(RKeys.y[i])
     endif
next i
```

Analizzando il corpo del primo loop, si nota che è stato utilizzato anche in questo caso il metodo `y[index]` sul dataset `LKeys` per ottenere il valore corrispondente all'indice attuale e posizionarlo nella stringa indicante il path relativo del file attuale, a partire dalla directory target del Datasource `AdaMo_Files`. L'Alias associato è il nome stesso del file e della relativa chiave. Il secondo loop replica pedissequamente la struttura del primo, ma sul dataset `RKeys`.

Il risultato è che adesso tutti i file indicati nel file di mapping sono stati aperti ed è possibile accedervi senza possibilità d'errore, dal momento che sono stati adoperati Alias inequivocabilmente uguali al nome del file di provenienza e della chiave sotto cui i dati contenuti saranno copiati.

5. A questo punto è necessario **creare un data file virtuale** a cui fare afferire le chiavi di memorizzazione tratte dai file aperti allo step precedente. A tale scopo, Concerto mette a disposizione i **ComposerFile**, aree di memoria gestite dal software, accessibili esattamente come un file di dati caricato da un Datasource. Come normali file di dati, possiedono un Alias e sono anch'essi composti da canali raggruppati in chiavi, che però è possibile modificare: possono essere create o rimosse anche intere chiavi di un Composer File. Generalmente sono adoperate le *Composer window* per creare canali e popolarli con valori inseriti manualmente o per trascinare dataset e chiavi da altri test. Per creare e gestire questi file di dati virtuali in ambiente di scripting, è invece disponibile la classe `Composer`.
   Inoltre, un'altra classe indispensabile per la procedura di creazione del file Composer e per altre operazioni illustrate nei prossimi step, è **Addressing**, che permette di istanziare riferimenti a unità di dati come test, measurement ID o dataset. Il metodo di questa classe utilizzato per definire un riferimento è `set` e ha i seguenti parametri facoltativi, da inserire o omettere a seconda di come si vuole utilizzare l'oggetto:

   - `Alias` (String): l'Alias di un test generalmente già aperto;
   - `GroupName` (String): nome di un Datasource;
   - `TestName` (String): nome di un test, utilizzabile con `GroupName` per associare il riferimento ad un test non aperto;
   - `ConcertoFileRef` (oggetto della classe File): utilizzato nel caso in cui si disponga di un *file object* ottenuto per esempio dalla funzione `SelFile` già introdotta;
   - `DataKeyName` (String): nome di una chiave di memorizzazione;
   - `MeasId` (Integer): measurement ID di una chiave;
   - `ChannelName` (String): nome di un canale, appartentente ad una chiave di un file;
   - `DataSetRef` (String): nome di un dataset;
   - `ChannelString` (String): stringa che indica un canale o dataset e segue il classico costrutto `Alias:KEY'DATASET`.

   Per creare dunque il file Composer è innanzitutto necessario istanziare un oggetto della classe `Composer` con la seguente sintassi:
   `Dim cmp As Composer`
   Dopodiché è stato utilizzato il suo metodo `CreateConcertoFile(Addr,...)`

che necessita del parametro obbligatorio `Addr` - un oggetto della classe `Addressing` - e accetta il parametro facoltativo `LayoutEmbedded`, un flag che permette di legare l'esistenza del Composer file al layout di Concerto attualmente aperto (il valore di default è 0). L'oggetto `Addr` passato a questo metodo deve presentare necessariamente i due parametri `TestName` e `Alias` precedentemente definiti.
Alla luce di quanto descritto, è stato istanziato un oggetto della classe `Addressing` e gli sono state associati, tramite il metodo `set`, i due parametri necessari alla creazione del composer. Infine è stato utilizzato il metodo `CreateConcertoFile` dell'oggetto `cmp`, che restituisce l'oggetto della classe `File` relativo al nuovo Composer file. Il codice è di seguito riportato:

```
Dim cmp As Composer
Dim newFileAddr As Addressing
newFileAddr.Set("TestName:", StrReplaceAll(dataDir,"\",""),
"Alias:", "PUMA1")
fo = cmp.CreateConcertoFile(newFileAddr, "LayoutEmbedded:",0)
```

Come si nota nella seconda riga, come parametro `TestName` è stato utilizzato il nome della directory selezionata dall'utente, ottenuto rimuovendo il carattere \ dalla stringa salvata nella variabile `dataDir`. Infine l'Alias adoperato per il Composer file è `PUMA1` per analogia con la struttura del test risult tipico di PUMA, che si intende ricreare.

6. Una volta che tutti i file sono stati aperti, il loro contenuto diviene accessibile ed è dunque possibile **copiarlo nel Composer file**. In base al modo in cui sono stati costruiti e manipolati i file prima dell'apertura, i dati che essi contengono saranno disponibili automaticamente sotto una chiave precisa, a seconda del tipo:

   - i file relativi a misure stazionarie (chiavi logpoint-based), provenienti cioè da istruzioni `SNAPLOG`, presentano i loro dati sotto la chiave `D`, canonicamente adottata per acquisizioni puntuali;
   - i file prodotti da misure continue (chiavi time-based), manipolati con lo script `RegHandler.py` nello step 3 per eliminare l'intestazione, presentano i dati sotto la chiave `TM`, canonicamente adottata per acquisizioni in base tempo.

   Inoltre nei nostri test le chiavi di tipo time-based, a differenza di quelle logpoint-based, sono composte spesso di più measurement ID, ciascuno associato ad uno dei file indicati nella variabile `RKeys`. Per distinguerli si aggiunge al loro nome un indice progressivo: ad esempio la chiave `REG` prodotta da un test PUMA, coposta da due measurement ID, sarà analogamente ricreata a partire da due file ASCII prodotti da AdaMo, chiamati

REG1 e REG2.
La classe Addressing, introdotta nell'analisi dello step precedente, torna utile anche in questo caso, per il processo di migrazione delle chiavi D e TM dei vari file verso il Composer cmp. In particolare sono necessari due oggetti di questa classe, uno (src) per la chiave *sorgente* da copiare e uno (trgt) per la chiave *target* in cui incollare i dati:

```
Dim src As Addressing
Dim trgt As Addressing
```

Più nello specifico, analizzando il caso della chiave V_IN, è stato adoperato su questi due oggetti il metodo set con i seguenti parametri:

```
src.Set("Alias:", "V_IN", "DataKeyName:", "D")
trgt.Set("Alias:", "PUMA1", "DataKeyName:", "V_IN", "MeasId:",
1)
```

Si noti che per la chiave sorgente è stato adoperato l'Alias V_IN, che identifica il file V_IN, e il nome della chiave D, visto che contiene dati di tipo logpoint-based; mentre per la chiave target è indicato l'Alias PUMA1 che identifica il Composer cmp, il nome della chiave desiderata V_IN e il measurement ID 1, che è l'unico sempre presente nelle nostre chiavi logpoint-based.
Prima di copiare la chiave identificata dall'oggetto src su quella identificata da trgt, è necessario che quest'ultima sia creata. A tale scopo è stato utilizzato il metodo AddNewDataKey(Addr,...) dell'oggetto cmp, appartenente alla classe Composer, con questa sintassi:
`cmp.AddNewDataKey(trgt,"BaseType:","LOGPOINT","Homogeneous:",1,"Resolution:",1)`
in cui sono stati adoperati i seguenti argomenti, tralasciandone altri opzionali:

- trgt è l'oggetto che identifica la chiave da creare nel Composer cmp, che nel nostro caso è il target della migrazione;
- BaseType (opzionale) è una stringa che può avere come valori accettabili LOGPOINT o TIMEBASED e indica il tipo di chiave da creare;
- Homogeneous (opzionale) è un flag che determina se la chiave da creare possa accettare solo dati omogenei; questo significa che tutti i suoi canali dovranno avere stessa risoluzione e numero di punti;
- Resolution (opzionale) è un flag che indica la risoluzione dei dati per la chiave da creare;

Infine il metodo `CopyDataKey(Addr, ...)` dell'oggetto `cmp` permette di copiare la chiave identificata da `src` in quella identificata da `trgt`, seguendo la seguente sintassi:

```
cmp.CopyDataKey(trgt,"SrcAddrRef:",src,"BaseType:",
"LOGPOINT","Homogeneous:",1,"TakeOverBaseValues:",1,
"Resolution:",1)
```

Il primo oggetto della classe `Addressing` passato è `trgt`, che identifica la chiave target in cui incollare i dati, mentre il riferimento alla chiave sorgente da copiare è effettuato per mezzo del parametro obbligatorio `SrcAddrRef`, a cui è associato l'oggetto `src`. I successivi parametri, opzionali, sono tutti analoghi a quelli utilizzati per il metodo `AddNewDataKey`, fatta eccezione per `TakeOverBaseValues`, cioè un flag che impedisce di adattare la *x-trace* dei canali copiati a quella degli eventuali canali già esistenti nella chiave.
In sintesi il codice per creare la nuova chiave nel Composer e per copiarci dentro i dati provenienti dal file ASCII è il seguente:

```
src.Set("Alias:", "V_IN", "DataKeyName:", "D")
trgt.Set("Alias:", "PUMA1", "DataKeyName:", "V_IN", "MeasId:",
1)
cmp.AddNewDataKey(trgt,"BaseType:","LOGPOINT","Homogeneous:",
1,"Resolution:", 1)
cmp.CopyDataKey(trgt,"SrcAddrRef:",src,"BaseType:",
"LOGPOINT","Homogeneous:",1,"TakeOverBaseValues:",1,
"Resolution:",1)
a1.Close()
```

Si noti che nell'ultima riga il file relativo a `V_IN` è chiuso con il metodo `Close()` dell'oggetto `a1` della classe `File`, che lo identifica.
La stessa procedura è ripetuta per tutte le chiavi indicate nella variabile `LKeys` - anch'esse afferenti alla tipologia logpoint-based - in un ciclo for:

```
for i=1 to nPoints(LKeys)
     if StrUpper(LKeys.y[i]) <> "NULL" then
          src.Set("Alias:",LKeys.y[i],"DataKeyName:","D")
          trgt.Set("Alias:","PUMA1","DataKeyName:",LKeys.y[i],
"MeasId:", 1)
          cmp.AddNewDataKey(trgt,"BaseType:","LOGPOINT",
"Homogeneous:", 1, "Resolution:", 1)
          cmp.CopyDataKey(trgt,"SrcAddrRef:",src,
"BaseType:","LOGPOINT","Homogeneous:",1,"TakeOverBaseValues:",
1,"Resolution:",1)
          af=SelFile(LKeys.y[i])
          af.Close()
```

```
     endif
next i
```

L'Alias adoperato per `src`, così come il nome della chiave per `trgt`, è in questo caso la stringa indicata da `LKeys.y[i]`. Si noti inoltre che per istanziare l'oggetto della classe `File`, corrispondente al file appena copiato, è utilizzata la funzione `SelFile` a cui è passato come parametro il relativo Alias indicato da `LKeys.y[i]`. Sull'oggetto `af` così ottenuto, è infine invocato il metodo `Close()` per la chiusura del file.
Leggermente diverso invece è il corpo dell'altro ciclo for necessario a copiare tutte le chiavi riportate in `RKeys`:

```
for i=1 to nPoints(RKeys)
     if StrUpper(RKeys.y[i]) <> "NULL" then
          last_char=StrErase(RKeys.y[i], 1,
StrLen(RKeys.y[i])-1)
          KName=RKeys.y[i]
          index=1
          if Not(isAlpha(last_char)) then
               index=CReal(last_char)
               KName=StrErase(RKeys.y[i],
StrLen(RKeys.y[i]), 1)
          endif
          src.Set("Alias:", RKeys.y[i], "DataKeyName:", "TM")
          trgt.Set("Alias:", "PUMA1", "DataKeyName:", KName,
"MeasId:", index)
          cmp.AddNewDataKey(trgt,"BaseType:","TIMEBASED",
"Homogeneous:",1,
"Resolution:",0.1)
          cmp.CopyDataKey(trgt, "SrcAddrRef:", src,
"BaseType:", "TIMEBASED","Homogeneous:",1,
"TakeOverBaseValues:",0,"Resolution:",0.1)
          af=SelFile(RKeys.y[i])
          af.Close()
     endif
next i
```

Per ogni iterazione è innanzitutto necessario discriminare se l'attuale file corrisponde ad un measurement ID o se combacia con l'intera chiave e per determinarlo è analizzato l'ultimo carattere del nome indicato da `RKeys.y[i]`, estratto con la funzione `StrErase`. Tale sub-string (di un solo elemento), salvata in `last_char`, è passata come parametro alla funzione `isAlpha` della libreria **String** per determinare se corrisponde ad un

carattere dell'alfabeto, per poi negare la condizione così ottenuta con la funzione `Not` della stessa libreria. Nel caso dunque in cui la condizione sia `True`, si deduce che si tratta del suffisso numerico progressivo tipico dei file corrispondenti a un measurement ID di una chiave time-based e di conseguenza si salva in `index` l'indice del measurement ID corrispondente, mentre a `KName` è assegnato il vero nome della chiave, sottraendo l'ultimo carattere alla Stringa `RKeys.y[i]`, con la funzione `StrErase`. Nel caso in cui la condizione sia `False`, si deduce che il file indicato da `RKeys.y[i]` corrisponde ad una intera chiave time-based, perciò sono mantenuti i valori di inizializzazione per le variabili `index` (1) e `KName` (`RKeys.y[i]`).
I valori di queste due variabili sono utilizzati infine nella procedura di creazione e copia della chiave, che risulta del resto quasi identica a quella realizzata per le chiavi logpoint-based indicate da `RKeys`. Le uniche differenze sono le seguenti:

- nel richiamo del metodo `set` sull'oggetto `src`, è stato indicato `"TM"` come nome della chiave, in quanto riferita ad un file di natura time-based;
- nel richiamo del metodo `set` sull'oggetto `trgt`, è stato indicato `KName` come nome della chiave e `index` come numero del measurement ID;
- nel richiamo dei metodi `AddNewDataKey` e `CopyDataKey` sull'oggetto `cmp`, è stato indicato `TIMEBASED` come valore del parametro `BaseType`.

7. Infine **il file Composer**, costruito negli step precedenti, **può essere esportato in formato ATF** per mezzo del metodo `Export` della classe `File`. Naturalmente è necessario prima istanziare un *file object* corrispondente al Composer identificato da `cmp`, che invece è un oggetto della classe `Composer`. Per fare questo è nuovamente adoperata la funzione `SelFile` a cui è passato l'Alias `PUMA1` relativo al Composer:
   `f_cmp=SelFile("PUMA1")`
   Su questo oggetto è infine richiamato il metodo `Export`, che richiede come unico parametro una stringa indicante il percorso assoluto del file da generare, inclusa l'estensione. I vari formati disponibili sono i seguenti:

   - ATF (ASAM Transport File)
   - ATFX (ASAM Transport File XML)
   - MF4 (MDF4 File Format)
   - XLSX (MS Excel File)
   - MCF (PUMA Recorder File)
   - CTF (CONCERTO Transport File)

   Tuttavia, come già anticipato precedentemente e come sarà meglio approfondito nel prossimo paragrafo, il formato a cui siamo interessati è **ATF**. Il

path assoluto da passare come parametro al metodo `Export` consiste nella stessa posizione nel filesystem della risorsa di rete, indicata dall'utente allo step 1, da cui sono stati aperti i file ASCII. Questo path è costruito con le variabili `SelectedDir` e `dataDir`:

```
f_cmp.Export(SelectedDir+StrReplaceAll(dataDir,"\", "")
+".atf", 0)
```

Si noti che il metodo `Export` richiede un secondo parametro, `withFormulaResults`, che permette di includere nell'export eventuali formule attualmente associate al file, il che non rispecchia il nostro caso. La stessa istruzione è eseguita nuovamente su un path diverso, che corrisponde alla subfolder dedicata alla sala prova in cui è stato eseguito il test, nella directory target del Datasource `AdaMo_Files`. La stringa indicante il nome della sala è stato assegnato alla variabile `TBench` nello step 2 ed è concatenata a `startPath` per realizzare il costrutto che segue:

```
f_cmp.Export(startPath+"\"+TBench+"\"
+StrReplaceAll(dataDir, "\", "")+".atf", 0)
```

Il motivo di questa ridondanza è chiarito nel prossimo paragrafo. A questo punto è possibile chiudere il file Composer con lo statement:

```
f_cmp.Close()
```

Infine, prima di concludere l'esecuzione dello script *FileConverter.csf*, è necessario ripristinare la User Variable `%CWF_AdaMoFilePath` con il valore inizialmente salvato in `origPath`, come anticipato nella descrizione dello step 3:

```
%CWF_AdaMoFilePath=origPath
```

Per rendere l'applicazione `FileConverter.csf` eseguibile anche da utenti meno esperti o che non dispongono del path effettivo dello script, è opportuno registrarlo tra i **job** che Concerto mette a disposizione. Si tratta di estensioni del software base, che i programmatori possono definire come script `csf`, integrabili per mezzo di una registrazione nel file di configurazione *concerto.ini*. Successivamente all'inserimento dello script nella corrispondente entry del file *concerto.ini*, all'avvio di Concerto sarà visibile nel tab **Calculations\Jobs** e dunque eseguibile. Il titolo del job corrispondente sarà uguale al nome del file `csf`, a meno che non sia adoperata nel corrispondente script la *keyword* `Description` per indicare la stringa che dovrà apparire come nome del job. Per questo motivo è stato inserito come prima riga del `FileConverter.csf` il seguente statement, secondo il corretto costrutto:

```
Description:  AITV File Conversion
```

Di conseguenza lo script realizzato sarà accessibile dal tab **Jobs**, come illustrato in fig 3.9.

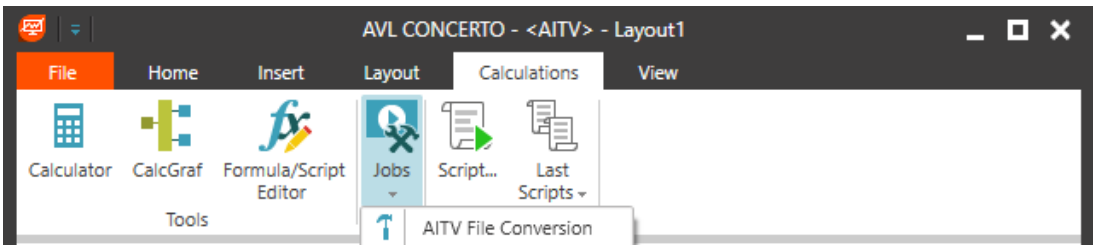

Figura 3.9: Esecuzione del job FileConverter.

3.2.2 Database centralizzato

Il processo di conversione appena descritto è completato dall'esportazione del file *Composer* che è stato creato. Sono dunque generati due file ATF, uno posizionato nella stessa cartella che costituisce il test result di AdaMo selezionato dall'utente per la conversione e che contiene i file ASCII d'origine; l'altra istruzione di `Export` è effettuata su un path fisso, corrispondente alla **sub-folder dedicata alla sala prova** in cui è stato eseguito il test, nella directory target del Datasource `AdaMo_Files`, identificata dalla variabile `startPath` nel codice del `FileConverter`. Tale directory, posizionata nel filesystem della risorsa di archiviazione condivisa nella LAN aziendale, costituisce il punto di partenza per la selezione delle cartelle di risultati da convertire, navigando in `AdaMo_Files` dal Data Explorer di Concerto. Nella stessa directory sono inoltre posizionate tante **sub-folder quante sono le sale prova AdaMo**, nelle quali sono automaticamente caricati gli ATF generati, al termine dell'esecuzione del `FileConverter`, come è stato descritto.
Va così delineandosi la struttura di un'area di archiviazione centralizzata, che convoglia nella risorsa di rete condivisa tutti file di risultati delle sale prova AdaMo. D'altro canto è comunque garantita una portabilità dell'ATF prodotto, grazie all'export supplementare effettuato nella cartella di risultati selezionata dall'utente. Il fatto che la directory contenuta in `startPath` sia il punto di partenza per la navigazione in `AdaMo_Files` deriva naturalmente dalla volontà di raccogliere gli stessi test result testuali - originali delle sale AdaMo - nell'area condivisa, ma non implica necessariamente che gli operatori di sala prova o gli application engineer siano costretti a caricarli nelle sub-folder predisposte per ogni sala prova. Di conseguenza è plausibile che le directory selezionate dall'utente siano posizionate in percorsi del tutto scorrelati (ma comunque vincolati al punto di partenza di `startPath`) e che l'utente abbia interesse a disporre dell'ATF generato anche in quel path.
Una parte del **Database centralizzato** consiste dunque di quattro percorsi di rete - tanti quante sono le sale AdaMo - ciascuno dei quali è accessibile da un Datasource che associa a tale path il formato ATF, assegnando come Alias di default `RESULT` (fig. 3.10). Le quattro sale prova in questione si chiamano **CM2**, **CM3**, **CM4** e **CM5**. A completamento di questa struttura, è necessario definire

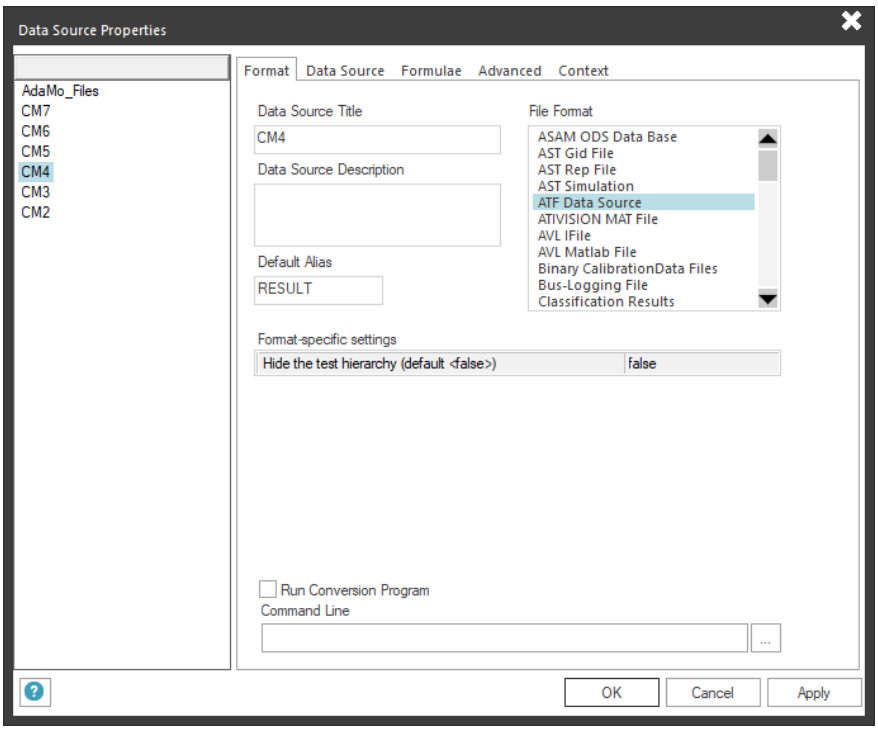

Figura 3.10: Definizione dei DataSource corrispondenti alle quattro sale prova AdaMo.

due Datasource per le rimanenti sale prova PUMA (**CM6** e **CM7**), i cui test archiviati sono tuttavia già accessibili attraverso la LAN aziendale. La piattaforma **Santorin** di AVL è dotata infatti di un server che permette di accedere al database ASAM ODS della sala prova PUMA, per mezzo di Datasource definiti su Concerto (fig 3.11), per i quali vanno definiti i seguenti parametri:

- **Server Name**, il nome o l'indirizzo del computer che ospita il database e il server ASAM-compliant;
- **Santorin Registry/RPC Number** (RPC Interface), a cui va assegnato uno dei Server Name proposti nel menu *drop-down* o, in alternativa, un *RPC number* fornito dal server (solitamente un valore a nove cifre decimali);
- **Version**, la versione dell'ASAM server a cui collegarsi;
- **Additional Parameters**, per direttive di accesso al database (ad esempio l'inserimento di credenziali).

Si noti inoltre che anche per questi due Datasource è stato impostato di default l'Alias `RESULT`, per rendere l'elaborazione del test del tutto avulsa da formato e sala prova d'origine. Si può asserire dunque che il **Database centralizza-**

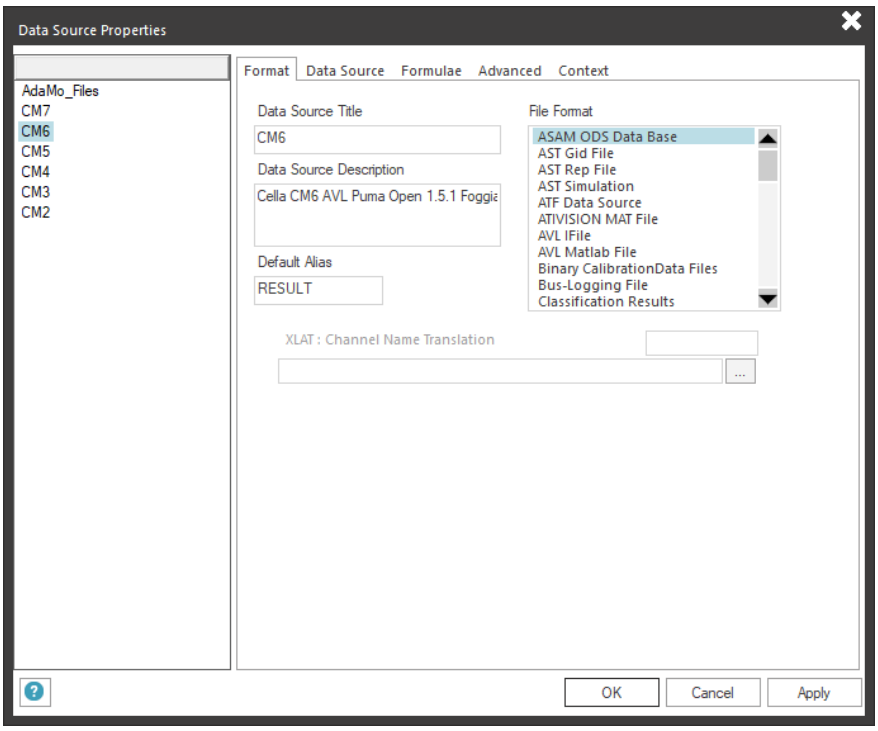

Figura 3.11: Definizione dei DataSource corrispondenti alle due sale prova PUMA.

to sia una collezione di Datasource opportunamente definiti, piuttosto che una reale struttura che raccoglie i test result di tutte le sale prova. Infatti, sebbene non esista realmente un database unico - fatta eccezione per l'area dedicata alle quattro sale prova AdaMo - è stato tuttavia creato su Concerto un **Data Environment** chiamato **AITV**, che contiene i 6 Datasource definiti precedentemente (fig. 3.12). Grazie a questo stratagemma è comunque garantito l'accesso da tutti gli ambienti di Concerto ai dati delle sei sale prova, come porzioni di un **Database centralizzato**. Si noti che nel Data Environment **AITV** è presente anche il Datasource `AdaMo_Files` adoperato dal FileConverter per accedere alle directory di risultati da convertire.
In fig. 3.13 è infine proposto un semplice schema che sintetizza la struttura appena descritta del database centralizzato.

3.3 Post-Elaborazione

Grazie al FileConverter e alle tecniche di archiviazione impiegate, il formato d'origine dei dati diviene a questo punto del tutto ininfluente ed è possibile accedere ai dati raccolti da ogni sala prova per mezzo dei sei Datasource definiti nel Data Environment `AITV`. Inoltre per ciascuno di essi è stato impostato di default

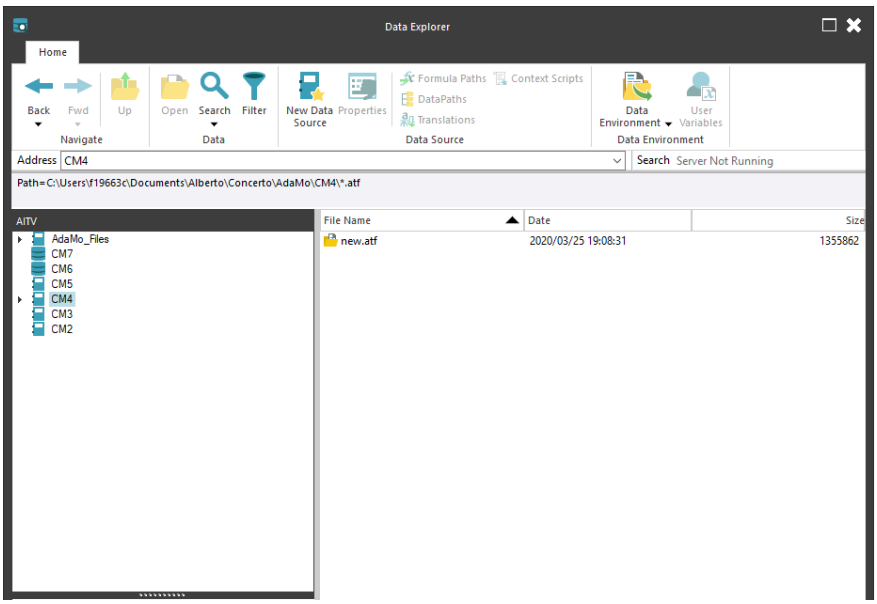

Figura 3.12: Data Environment AITV per l'accesso al database centralizzato delle sei sale prova.

l'Alias `RESULT`, come descritto nel paragrafo 3.2.2, in modo tale da rendere i layout di post-elaborazione totalmente neutrali rispetto al formato e alla sala prova d'origine. Tutti i Datasource divengono così *data provider* equivalenti sotto ogni aspetto.
In questo nodo dell'architettura, dedicato alla post-elaborazione, sono adoperati dei **layout** Concerto (file `cly`), che permettono di raccogliere le diverse finestre realizzate per la presentazione grafica dei dati. Sono diverse le tipologie di **window** inseribili nel layout, caratterizzate da vari oggetti, di cui si propongono alcuni esempi:

- **Diagram**, utilizzato per la visualizzazione grafica dei dati per mezzo di **sub-objects** di diversa natura, di cui sono proposti alcuni esempi:
  - **line curve**, che consistono basicamente di linee spezzate su un piano cartesiano le cui ordinate sono i valori di un *dataset* - cioè la sua *y-trace* - mentre l'asse delle ascisse può essere l'*x-trace* dello stesso dataset, o anche la *y-trace* di un altro dataset;
  - **bar curve**, cioè istogrammi con riferimenti analoghi alle line curve, ma con valori rappresentati da colonne (o *bars*);
  - **contour map**, cioè grafici tridimensionali in formato `z=f(x,y)`, che consistono in isolinee sul piano `x/y`, le quali raggruppano punti[8] a quota uguale;

[8]Si tratta di valori non misurati, ma interpolati tridimensionalmente da funzioni spline cubiche, comparabili a mappe o linee contour [2].

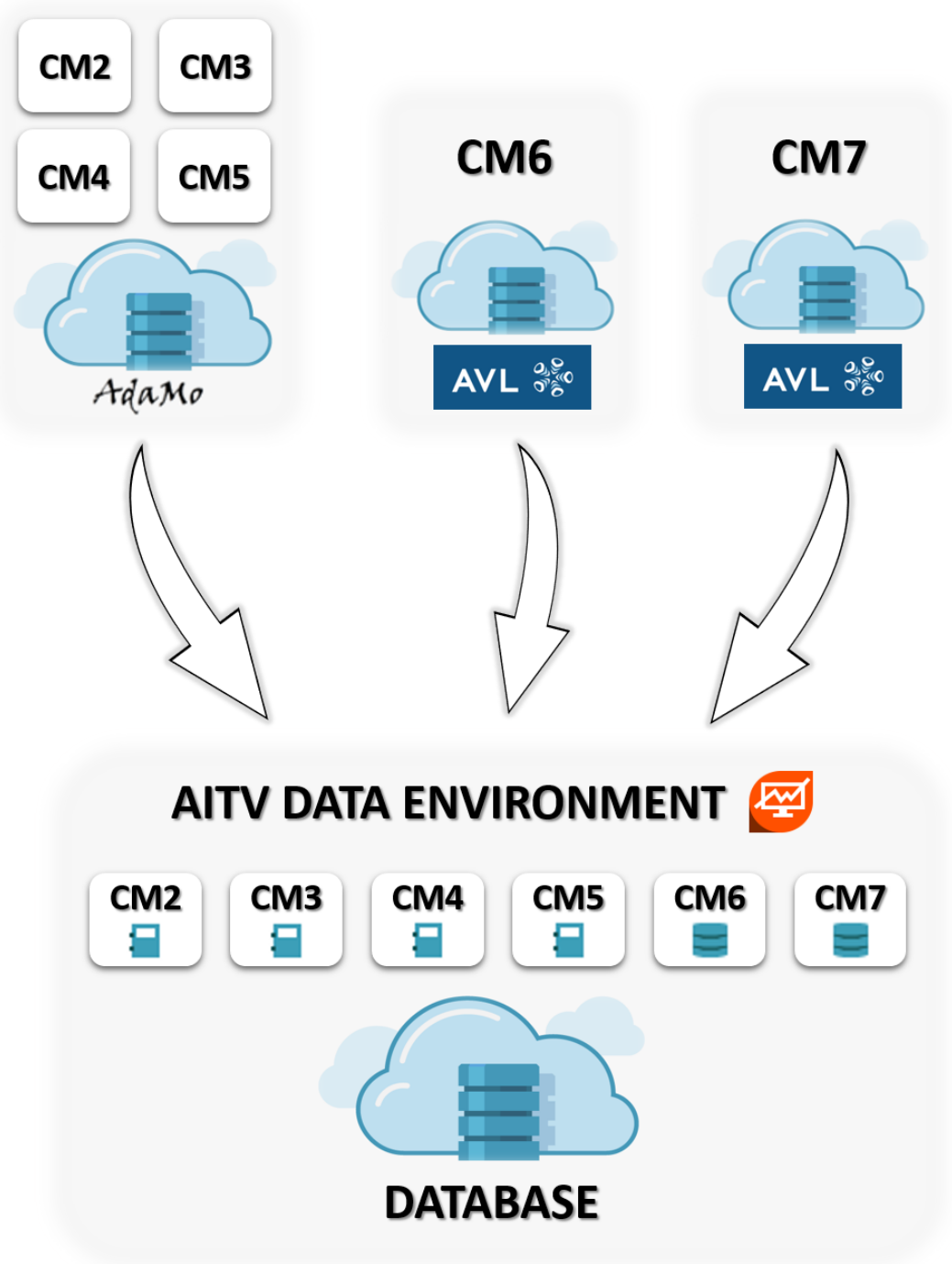

Figura 3.13: Struttura del Database centralizzato.

  - **table**, orizzontali o verticali, usate per visualizzare dati in formato alfanumerico.

- **Report**, che può consistere di liste di dati, form, tabelle e semplice testo;
- **Dialog**, cioè una finestra d'interazione con l'utente, che può presentare diversi oggetti - inseribili anche nelle finestre di tipo `Diagram` -, come *action button*, *list box*, *check box*, e altri ancora.
- **Composer**, già introdotta nel paragrafo 3.2.1, è una finestra per la creazione e manipolazione manuale di chiavi e dataset di un file Composer.

Naturalmente i dati associati alle finestre possono essere dataset *misurati*, cioè provenienti da un file di dati caricato, oppure **formule** (file `frm`) che applicano ai dataset misurati calcoli implementati nel linguaggio Concerto - introdotto nei paragrafi precedenti - con il quale si realizzano anche gli script `csf`, dai quali differiscono per la necessaria presenza della keyword `return` che indica la variabile da restituire. Si tenga presente che nella logica di funzionamento del linguaggio di Concerto ogni variabile è un dataset e di conseguenza ogni formula è a tutti gli effetti un dataset, che si va ad aggiungere a quelli *misurati*. L'utilizzo di formule è quindi un modo per aggiungere **canali virtuali** ad un file di dati e ciascuna di esse può accedere ad un numero illimitato di dataset - naturalmente anche altre formule - per il calcolo, ma restituisce un solo dataset come risultato. In linguaggio Concerto, il riferimento ad altri dataset è effettuato per mezzo del costrutto `Alias:KEY'DATASET`.
La metodologia più robusta per la definizione e l'utilizzo di formule consiste nell'assegnarle ad un Datasource, in modo tale che siano associate automaticamente ad ogni file da esso raggiungibile. L'associazione è effettuata per mezzo della dichiarazione di percorsi relativi a directory, nelle quali sono raccolti i file `frm` che implementano le formule. Ad ogni percorso è assegnata una chiave che sarà automaticamente aggiunta ai file raggiungibili dal Datasource e che contiene tutte i dataset-formule contenuti nella relativa directory. Per i sei Datasource `CM2`, `CM3`, `CM4`, `CM5`, `CM6` e `CM7`, sono state configurate tutte le chiavi - e i relativi path - necessarie per avere a disposizione le formule utili a realizzare i layout di post-elaborazione delle prove.
Analogamente agli script, possono essere implementate anche formule in linguaggio **Python**, che permettono di mettere tutte le potenzialità di questo linguaggio a servizio degli utilizzi previsti per le classiche formule Concerto. A tal proposito è opportuno specificare che Concerto integra di base un *Python Environment* basato su una versione 3.6.4 e aggiunge alle feature built-in i seguenti due moduli:

- **NumPy**[9], molto utile per il calcolo scientifico con Python, ma può anche essere usato per strutture dati multidimensionali defininendo all'occorrenza tipi di dato arbitrari;

[9] `numpy.org`

- **Pandas**[10], molto utile per la manipolazione e l'analisi dei dati, per mezzo di strutture dati quali `Series` (monodimensionale) e `DataFrame` (bidimensionale).

Com'è stato anticipato nel paragrafo 3.2.1, a proposito dello script ausiliario `RegHandler.py` a supporto del `FileConverter`, è comunque possibile estendere l'ambiente Python di base e integrare altri package. Inoltre AVL mette a disposizione il modulo `concerto` per la gestione degli elementi base della programmazione Concerto, anche in ambiente Python. In particolare per la gestione dei dataset è possibile utilizzare due metodi:

- `concerto.ds`, che restituisce un dataset come un *array* di NumPy, accettando come parametri le coordinate del canale;
- `concerto.ds`, che restituisce uno o più dataset come un *DataFrame* di Pandas, accettando dunque più coordinate di canali come parametri.

Infine vale la pena includere in questa descrizione le **Macro** (file `mac`), un utile strumento di calcolo ausiliario a quelli già citati. Sebbene condividano la stessa sintassi delle formule (fatta eccezione per la keyword `arg()` con cui accettano parametri), le Macro non generano canali virtuali, ma sono più propriamente funzioni o sub-routine definibili dall'utente e integrabili nel codice di formule e script o utilizzabili per le trasformazioni degli assi x e y negli oggetti Diagram. Dal momento che, diversamente dalle formule, non generano dataset virtuali che vanno ad aggiungersi a quelli *misurati*, le Macro non possono essere associate ai Datasource. Resta tuttavia un importante strumento per il riutilizzo di codice in più contesti.
Per ogni test implementato sui sistemi d'automazione di sala prova, è dunque realizzato il relativo layout Concerto per mezzo di tutti questi strumenti. É possibile in questo modo estrarre tutti i risultati finali delle prove e definire una serie di finestre del layout che andranno a comporre il report del test effettuato.
Di particolare interesse, per l'analisi che si sta conducendo sul progetto realizzato, è soprattutto la **gestione dei risultati finali calcolati dalle maschere di post-elaborazione**, che a questo punto possono essere oggetto di **validazione mediante AITV** - che sarà analizzato nel paragrafo 3.4 - o possono essere raccolti e **aggiunti al training set** corrispondente alla prova effettuata, contribuendo alla precisione di future validazioni. Entrambe le operazioni corrispondono all'esecuzione di un job associato, a seguito del caricamento del file di risultati d'interesse e dell'applicazione del layout, il che costituisce l'unico requisito in funzione del quale i due task sono parimenti richiamabili dall'utente. Infatti, come illustrato nel capitolo 2, post-elaborazione e validazione costituiscono due elementi dell'architettura paralleli e connessi, posizionati sullo stesso livello: sebbene la logica di utilizzo di questi strumenti preveda che i risultati siano validati

[10] `https://pandas.pydata.org`

prima di utilizzarli per aggiornare il training set, l'utente gode della libertà di scegliere se e quale job eseguire, a seconda dello step che sta effettuando nell'attività di ricerca. Si ricorda infatti che il test è dapprima eseguito in sala prova, successivamente il file di risultati - eventualmente convertito dal FileConverter - è caricato in Concerto attraverso il corretto Datasource perché sia applicato il layout di elaborazione. I risultati così ottenuti sono validati mediante AITV e infine, se ritenuti affidabili, sono aggiunti al training set insieme ai parametri di input contenuti in `V_IN`.
Nel prossimo paragrafo è analizzato il **Parser**, il componente software - registrato come job Concerto - che è in grado di estrarre i valori rilevanti definibili come risultati finali della prova, per caricarli nel file corrispondente al training set corretto inseme ai corrispondenti parametri di `V_IN`.

3.3.1 Parser

Molti degli elementi base della programmazione Concerto/Python sono stati introdotti nei paragrafi precedenti, in particolar modo in merito al FileConverter. Buona parte di essi saranno dunque dati per scontati in questa sezione, nell'analisi del **Parser**. Nelle prossime pagine ci sono comunque molteplici riferimenti al paragrafo 3.2.1, per meglio spiegare analogie e differenze con il codice già illustrato in quella sezione.
Per effettuare l'operazione di raccolta dei valori di input e output della prova e di scrittura degli stessi nel file di training set, è stato realizzato uno script interamente in linguaggio Python, naturalmente eseguito in ambiente Concerto. Nel caso del `FileConverter.csf` è stato ritenuto opportuno integrare i due differenti linguaggi disponibli, in virtù delle utili caratteristiche di entrambi. Nello specifico è stato necessario realizzare in linguaggio Concerto la vera e propria struttura del FileConverter, soprattutto per via della semplice gestione dei file Composer, impossibile da ottenere lavorando esclusivamente in linguaggio Python. É stata tuttavia sfruttata la possibilità di integrare il modulo `csv` per la procedura di rimozione delle intestazioni dei file time-based di AdaMo, effettuata dallo script `RegHandler.py`. In questo caso, invece, è stato possibile realizzare il Parser interamente in linguaggio Python, mantenendo una compattezza maggiore del codice e della struttura senza il vincolo di sincronizzare l'esecuzione di due script paralleli e di gestire lo scambio di parametri.
Per registrare il Parser come job Concerto, è stato tuttavia necessario realizzare il piccolo script `Parser.csf`, che semplicemente lancia il `Parser.py` per mezzo della funzione `ExecutePythonScript`, già adoperata nel caso del FileConverter. L'intero codice Concerto del `Parser.csf` è riportato di seguito:

```
Description:  AITV Parser
startPath = %CWF_AdaMoFilePath
startPath = StrReplaceAll(startPath,"\\","\")
```

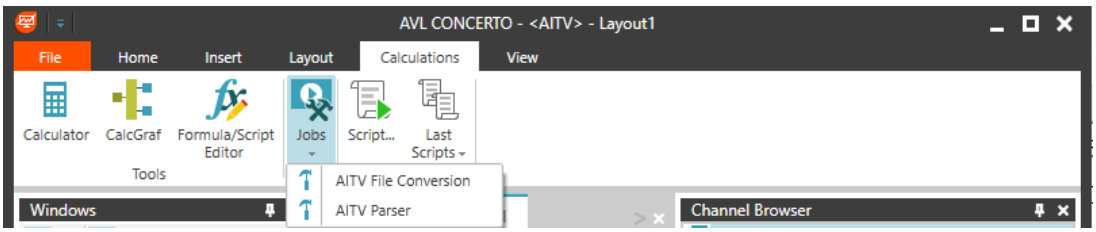

Figura 3.14: Esecuzione del job Parser.

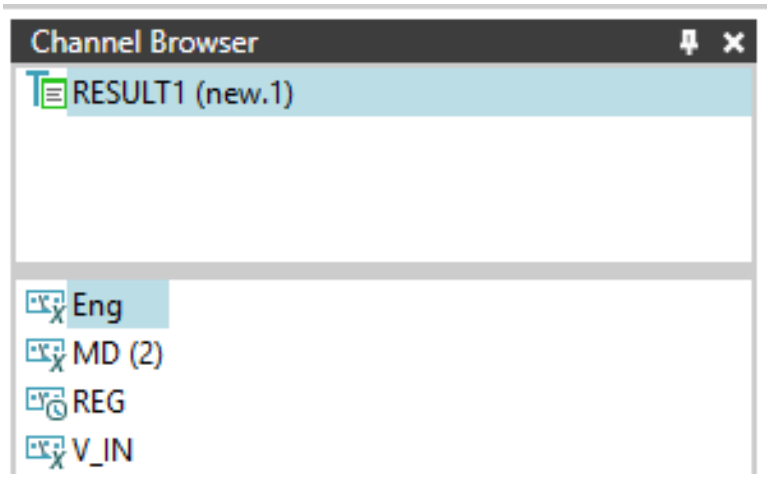

Figura 3.15: Visualizzazione del file di risultati aperto.

```
ExecutePythonScript(startPath + "lib\Parser.py")
```

Analizzando brevemente le precedenti righe, la prima istruzione determina il nome del job Concerto (com'è riportato in figura 3.14), mentre le due righe successive hanno lo scopo di determinare il path della directory - anche target del Datasource `AdaMo_Files` - nella quale è presente la sub-folder `lib` contenente script e file di testo accessori come i file di mapping delle chiavi. Il percorso di questa directory è memorizzato come valore di default della User Variable `%CWF_AdaMoFilePath`, adoperata per il passaggio di tale informazione dall'ambito del `FileConverter.csf` al `RegHandler.py`. Si noti tuttavia che in questa stringa ogni carattere \ è raddoppiato, dal momento che si tratta di un carattere speciale delle stringhe in Python. Di conseguenza è necessario "riconvertire" il path perché sia utilizzabile in linguaggio Concerto, per mezzo della funzione `StrReplaceAll` con cui è stato rimpiazzato ogni \\ con un solo \. Infine è eseguito lo script `Parser.py`, posizionato nella sub-folder `lib`, per mezzo della funzione `ExecutePythonScript`. Si pone ora l'attenzione sul `Parser.py`, di cui verrà di seguito analizzata la struttura. Si tenga a mente che l'esecuzione del job Parser avviene appena dopo la conclusione della post-elaborazione per mezzo del layout e delle formule implementate, ma contestualmente ad essa, senza cioè

chiudere la sessione di Concerto o effettuare altre operazioni. Più precisamente, è fondamentale che al momento dell'esecuzione del Parser sia aperto il file di risultati della prova elaborata (fig. 3.15), rigorosamente per mezzo del Datasource corrispondente alla sala prova di provenienza. In questo modo è assicurato che siano correttamente assegnate tutte le formule necessarie a determinare i risultati finali del test, che il Parser deve inserire nel Training Set. Inoltre questo garantisce che sia correttamente assegnato l'Alias `RESULT1`, fondamentale per l'esecuzione del codice analizzato di seguito.
Innanzitutto è stato necessario importare i quattro moduli con i seguenti statement:

```
import concerto as conc
import numpy as np
import pandas as pd
import ctypes as ct
```

In particolare:

- il modulo `concerto` è necessario per la gestione dei dataset del file di risultati con Alias `RESULT1` e di altri file ausiliari;
- il modulo `numpy` è mandatorio per la gestione dei dataset ottenuti per mezzo del metodo `concerto.ds`, che restituisce infatti il dataset argomento come un array di NumPy;
- il modulo `pandas` è mandatorio per la gestione dei dataset ottenuti per mezzo del metodo `concerto.dsframe`, che restituisce infatti i dataset argomento come un DataFrame di Pandas; inoltre sono state adoperate altre strutture di Pandas per la gestione e l'aggiornamento del training set;
- il modulo `ctypes` è stato integrato semplicemente per la visualizzazione di un popup a fine script, che ne indica il corretto completamento.

Con riferimento alla fig. 3.16, si può suddividere la struttura del `Parser.py` nei seguenti cinque punti.

1. All'inizio dello script, dopo aver importato i moduli necessari, la prima operazione consiste nell'**aprire il file di training set** corrispondente alla tipologia di prova a cui afferisce il file di risultati con Alias `RESULT1`. Tutti i training set, così come tutti i file di mapping e gli script, sono contenuti nella già citata sub-folder `lib`. Non è necessario caricare il training set come un file Concerto, poiché non siamo interessati a leggerne il contenuto, ma è più opportuno aprirlo come `DataFrame` del modulo Pandas. Pertanto non è adoperato il Datasource `AdaMo_Files`, che di base punta alla directory del filesystem della risorsa d'archiviazione condivisa, nella quale

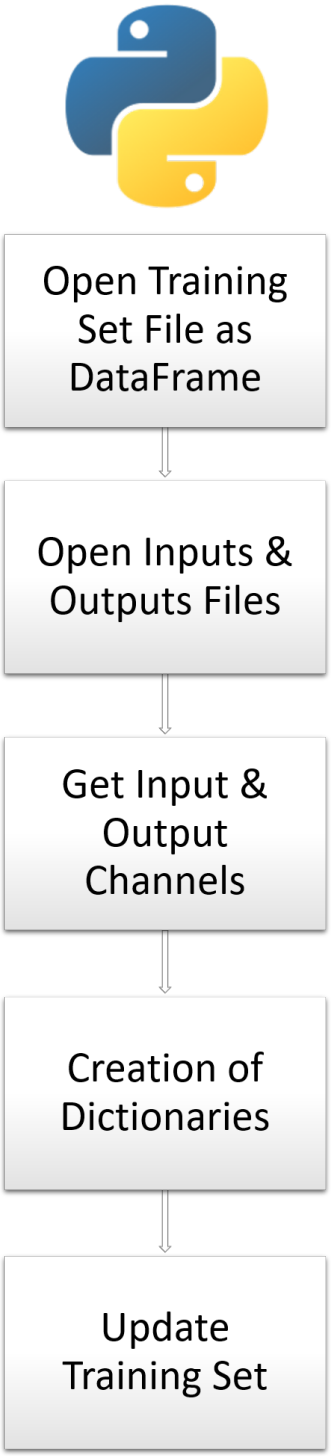

Figura 3.16: Struttura del Parser.

è posizionata la sub-folder `lib`, ma è necessario ricorrere nuovamente alla User Variable `%CWF_AdaMoFilePath`, questa volta sfruttandone il valore originale che compone il path con il doppio backslash \\. É dunque prima di tutto istanziata la variabile `startPath` in questo modo:

```
startPath = conc.variables["%CWF_AdaMoFilePath"]
```

Per ottenere il valore di `%CWF_AdaMoFilePath` è adoperata la struttura `variables` del modulo `concerto`.
Successivamente è necessario ottenere il tipo di prova eseguita, riportato nel dataset `Test_Type` appartenente alla chiave `V_IN` del test `RESULT1`, che viene assegnato alla variabile `test_type`. In questo caso è utilizzato il metodo `ds` che restituisce come array NumPy il dataset indicato dal classico costrutto `Alias:KEY'DATASET`. Di questo array è estratto il primo elemento, corrispondente all'indice `0`, a differenza di quanto avviene in linguaggio Concerto nel quale i dataset - così come altre strutture - hanno come indice di partenza `1`. Al netto di questa differenza di convenzione, il codice di seguito riportato è analogo a quello adoperato nel `FileConverter.csf` per lo stesso scopo, che era stato realizzato in linguaggio Concerto mediante il metodo `y[index]`.

```
test_type = conc.ds("RESULT1:V_IN'Test_Type")[0]
```

Le due stringhe `startPath` e `test_type` sono dunque combinate per determinare il percorso del file di training set in questo modo:

```
training_set_path=startPath+"lib\\"+test_type+
"_TrainingSet.txt"
```

Come si può evincere, anche il file di training set ha nel nome la stringa indicante il tipo di prova, analogamente al file di mapping delle chiavi.
Questo path è infine utilizzato per caricare il training set come `DataFrame` per mezzo del metodo `read_csv` del modulo `pandas`, al quale in questo caso sono passati come argomenti solo il percorso del file, assegnato alla variabile `training_set_path`, e il separatore adoperato nella divisione delle colonne, cioè `"\t"`.

```
training_set = pd.read_csv(training_set_path, sep="\t")
```

In realtà sono molto più ampie le potenzialità di questo metodo, che permette di filtrare i dati acquisiti dal file csv letto, ad esempio eliminando righe e colonne o effettuando automaticamente parsing di alcuni formati nelle stringhe dei singoli elementi. Anche le utilità della struttura `DataFrame` sono svariate, ma nel nostro caso specifico è particolarmente funzionale la sua suddivisione in righe e colonne accessibili per nome. In questo modo è possibile, come sarà illustrato nel punto 5, aggiungere nuove entry del DataFrame - per mezzo del metodo `append` - creandole come `dictionary`, una struttura dati tipica di Python che consiste in una collezione di dati non ordinata, modificabile e indicizzata.
In conclusione, il codice che rappresenta la prima fase d'esecuzione del Parser è il seguente:

```
startPath = conc.variables["%CWF_AdaMoFilePath"]
test_type = conc.ds("RESULT1:V_IN'Test_Type")[0]
training_set_path=startPath+"lib\\"+test_type+
"_TrainingSet.txt"
training_set = pd.read_csv(training_set_path, sep="\t")
```

2. In merito al FileConverter, per ottenere la lista delle chiavi da mappare nel nuovo file composer è stato utilizzato un file testuale di mapping, nel quale erano riportati i nomi delle chiavi logpoint-based e di quelle time-based. Analogamente, il Parser si avvale di due altri **file che contengono la lista di canali rispettivamente di input e output della prova**, ovvero l'insieme di parametri iniziali del test (salvati in `V_IN`) e quello dei risultati finali (calcolati dalla maschera di post-elaborazione). Anche questi file, distintivi di ogni prova, sono posizionati nella cartella `lib` e anche nel loro nome è presente la stringa indicante il tipo di prova, precedentemente assegnato alla variabile `test_type`.
   Diversamente da quanto detto a proposito del training set, è opportuno aprire questi due file con la metodologia classica di Concerto e avvalendosi del Datasource `AdaMo_Files`. In questo modo sono entrambi caricati e rendono disponibili i due importanti dataset contenenti i nomi dei canali di input e output.
   Per aprire dunque i due file è adoperato il metodo `select_file` del *submodule* `data`, appartenente al modulo `concerto`, il quale restituisce un *file object* in maniera del tutto analoga alla funzione `SelFile` usata nel codice Concerto del FileConverter. Anche in questo caso, l'argomento da passare a `select_file` è il percorso relativo del file da selezionare, definito a partire dal Datasource di riferimento:

   ```
   inp_file=conc.data.select_file("AdaMo_Files" + "\\lib\\"
   + test_type + "_Inputs.txt")
   out_file=conc.data.select_file("AdaMo_Files" + "\\lib\\"
   + test_type + "_Outputs.txt")
   ```

   Gli oggetti `FileObject` restituiti da `select_file`, in questo caso chiamati `inp_file` e `out_file`, hanno metodi e attributi molto simili alla classe `File` del linguaggio Concerto. Per aprire i due file è infatti invocato il metodo `open`, a cui è passato come argomento l'Alias da utilizzare nel caricamento:

   ```
   inp_file.open("INPUTS")
   out_file.open("OUTPUTS")
   ```

Si chiarisce tuttavia che ci dev'essere una certa coerenza tra il contenuto di questi due file e la struttura del training set: quest'ultimo deve contenere tante colonne quanti sono i canali di input e output indicati nei file, riportando esattamente gli stessi nomi, naturalmente a meno della chiave di cui fanno parte. Non è comunque fondamentale che sia osservato un ordine preciso nella successione delle colonne, dato che l'accesso a ciascuna di esse è sempre effettuato mediante il nome del dataset corrispondente.

3. Il contenuto dei file caricati nello step precedente, si riduce dunque semplicemente a un solo dataset per ciascuno, `Inputs` e `Outputs`. In questa fase si procede ad **assegnare il valore di questi due canali alle variabili** `inp` e `out`. Analogamente a quanto visto per il FileConverter, per cui si è evidenziato che i file di tipo logpoint-based aperti mediante il Datasource `AdaMo_Files` presentano la sola chiave `D` una volta caricati, anche il caricamento di questi due file produce lo stesso risultato, in quanto strutturati in maniera analoga. Per questo motivo le istruzioni per ottenere i dataset di nostro interesse dai due file, per mezzo del metodo `ds` del modulo `concerto`, sono le seguenti:

   ```
   inp=conc.ds("INPUTS:D'Inputs")
   out=conc.ds("OUTPUTS:D'Outputs")
   ```

   Si noti che sono stati adoperati i due Alias dichiarati precedentemente, che permettono di riferirsi correttamente ai due file.
   Una volta assegnato il contenuto dei due dataset alle variabili `inp` e `out`, che sono dunque strutturate come array del modulo NumPy, è possibile chiudere i due file per mezzo del metodo `close` dei due file object istanziati:

   ```
   inp_file.close()
   out_file.close()
   ```

   Si chiarisce infine che il significato dei termini "input" e "output", usati per indicare rispettivamente i parametri di input memorizzati nella chiave `V_IN` durante il test e i risultati finali ottenuti dalla post-elaborazione, si riferisce al ruolo che questi parametri rivestono nella predizione performata da AITV, che sarà introdotta nel prossimo paragrafo.

4. Come anticipato al punto 1, a rendere particolarmente funzionale la struttura dati `DataFrame` di Pandas per il processo di aggiornamento del training set, è soprattutto la possibilità di effettuare riferimenti per nome alle colonne, il che garantisce in questo caso di poter sfruttare nel metodo `append` - che sarà utilizzato nel prossimo step - la struttura dati `dictionary` per la costruzione di una nuova entry. In questa fase **è creato un dictionary per gli input e uno per gli output**, che sono successivamente **com-**

binati in un solo dictionary corrispondente alla nuova entry del training set.
Come precedentemente accennato, la peculiarità di questi costrutti di Python consiste nell'associazione chiave-valore che caratterizza ogni elemento. Possono essere definiti esplicitando direttamente alcuni elementi, mediante le parentesi graffe: ad esempio lo statement `dict0 = {"el1": 1, "el2": 2}` crea il dictionary `dict0` dotato di due elementi, di cui il primo è identificato dalla stringa `"el1"` e ha valore `1`, mentre il secondo è `"el2"` di valore `2`. Nel nostro caso tuttavia ciascun dictionary è popolato mediante un for loop, che scorre gli elementi del dataset contenente la lista di canali di input o output, perciò è più opportuno l'utilizzo del costruttore `dict()`, con il quale è realizzata un'inizializzazione del dictionary senza dichiarazione di elementi:

```
in_dict=dict()
out_dict=dict()
```

L'inserimento di nuovi elementi in un dictionary invece può essere implementato con un semplice costrutto che esemplificato dallo statement `dict0["el3"]=3`, che aggiunge al dictionary `dict0` un nuovo elemento la cui chiave è la stringa `"el3"`, a cui è assegnato il valore `3`.
Prima di analizzare la struttura dei due loop che popolano i dizionari `in_dict` e `out_dict`, è importante ricordare che all'interno della stessa prova possono essere raccolti più campioni della chiave `V_IN`, per altrettanti punti nei dataset dei valori finali calcolati dalla post-elaborazione. Si consideri ad esempio il caso di studio del Lambda Step analizzato nel capitolo 4, si tratta di un tipo di prova che ripete la stessa procedura su più punti operativi (determinati da un valore di Giri e uno di Coppia), per ciascuno dei quali è raccolto un nuovo campione per ogni canale della chiave `V_IN` ed è calcolato un valore risultate di Oxygen Storage. Per questo motivo la procedura di creazione di una nuova entry per il training set dev'essere effettuata per ogni punto della chiave `V_IN`. Si deduce quindi che a tale scopo è realizzato un for loop, il cui numero di iterazioni è determinato dalla dimensione di un qualsiasi dataset appartenente a `V_IN`, che sarà identica alla dimensione dei dataset relativi ai valori finali, in virtù di una corretta definizione delle formule e in generale della maschera di post-elaborazione. É dunque assegnata alla variabile `chsize` la dimensione del dataset `RESULT1:V_IN'Test_Type`, già adoperato in precedenza per ottenere la stringa indicante il tipo di prova, che sicuramente sarà presente nella chiave `V_IN`. Tra i differenti metodi volti ad ottenere la dimensione di un dataset, si è preferito l'utilizzo del metodo `dsinfo` del modulo concerto, che restituisce un dictionary contenente tutte le proprietà (nome, descrizione, unità di misura, dimensione, tipo, etc.) del dataset passato

come parametro per mezzo del costrutto tipico `Alias:KEY'DATASET`. In particolare siamo interessati all'elemento che indica la dimensione, perciò l'istruzione implementata è la seguente:
`chsize=conc.dsinfo("RESULT1:V_IN'Test_Type")["size"]`
A questo punto è possibile definire il for loop che itera su tutti gli elementi memorizzati in `V_IN`, mediante la funzione built-in `range`, che restituisce una sequenza di numeri che di default parte da 0 e incrementa di 1 (argomenti opzionali), ma termina al numero specificato come come unico parametro obbligatorio. Il for statement utilizzato è dunque il seguente:
`for i in range(chsize):`
Il corpo di questo loop è di seguito analizzato:

- Innanzitutto è realizzato un ciclo for con il quale è popolato `in_dict`, assegnando alla variabile `x` il valore dell'elemento corrente dell'array `inp`:

  ```
  for x in inp:
      in_dict[x]=conc.ds("RESULT1:V_IN'"+x)[0]
  ```

  Per ogni iterazione è dunque aggiunto a `in_dict` un nuovo elemento la cui chiave è la stringa attualmente contenuta in `x` e il cui valore è dato dall'elemento attuale (determinato dall'indice i del for più esterno) dell'array NumPy restituito dal metodo `ds`, a cui è passata la stringa che segue il solito costrutto `Alias:KEY'DATASET`, dove il `DATASET` è in questo caso la stringa `x`.
  Si specifica che gli elementi dell'array `inp`, che riporta il contenuto del relativo file, sono stringhe corrispondenti ai **nomi** dei dataset riportati nella chiave `V_IN`.

- Successivamente è implementato il ciclo che popola `out_dict`. Contrariamente a quanto visto per `in`, gli elementi dell'array `out` sono stringhe che indicano **anche la chiave di provenienza** del dataset rappresentante un risultato della prova (tipicamente una formula. Per questa ragione, per derminare l'identificativo del nuovo elemento è utilizzato il metodo `split(')` per dividere la stringa `y`, che si presenta nella forma `KEY'DATASET`, in modo da ottenere una `list`[11], di cui il secondo elemento (indice 1) è il nome del dataset. Questa differenza nella nomenclatura dei canali in `out`, comporta anche una variazione dell'argomento del metodo `ds`, dal momento che la variabile `y` contiene anche il nome della chiave. Il codice realizzato è di seguito riportato:

[11] Si tratta di un altro *data collection type* di Python, che rappresenta una successione di elementi ordinata e modificabile e permette membri duplicati. Altri *data collection type* sono il già citato `dictionary`, il `set` (ordinato e non modificabile, permette duplicati) e le `tuple` (non ordinato e non indicizzato, non permette duplicati).

```
for y in out:
    out_dict[y.split("'")[1]]=conc.ds("RESULT1:"+y)[i]
```

- Ottenuti i due dictionary `in_dict` e `out_dict`, vanno ora uniti per formare il nuovo `entry_dict`. A tale scopo in realtà si rivelano efficaci più tecniche. Una prima ipotesi potrebbe consistere nell'utilizzo del metodo `update` di uno dei due dictionary - ad esempio `in_dict` -, a cui è passato come argomento l'altro dictionary - `out_dict`. Quest'operazione modificherebbe però la struttura dell'oggetto `in_dict`, compromettendo l'iterazione successiva del loop su tutti i campioni della chiave `V_IN` e complicando l'implementazione. Per questo motivo si è preferito adoperare l'opertore `**kwargs`, che generalmente permette di passare ad una funzione un numero variabile di argomenti definiti come coppie chiave-valore. L'applicazione di quest'operatore ad un dictionary ha dunque l'effetto di espanderne il contenuto deserializzandolo in una collezione di coppie chiave-valore. Di conseguenza la creazione del nuovo dictionary `entry_dict`, consiste in una semplice definizione mediante il contenuto deserializzato di `in_dict` e `out_dict`:
  `entry_dict={**in_dict,**out_dict}`
- La nuova entry, creata nello step precedente, è dunque aggiunta al DataFrame `training_set` mediante il metodo `append`, al quale è passato come argomento il dictionary `entry_dict` e il flag `ignore_index = True`. Va precisato che potrebbe essere passato a questo metodo anche un intero DataFrame e per questo motivo il flag `ignore_index` permette di ignorare gli indici (in un DataFrame gli indici sono i valori che identificano le righe) corrispondenti alle nuove righe da aggiungere. L'istruzione adoperata è la seguente:
  `training_set=training_set.append(entry_dict, ignore_index = True)`

Infine è di seguito riportato il codice complessivo per questa fase:

```
in_dict=dict()
out_dict=dict()
chsize=conc.dsinfo("RESULT1:V_IN'Test_Type")["size"]
for i in range(chsize):
    for x in inp:
        n_dict[x]=conc.ds("RESULT1:V_IN'"+x)[i]
    for y in out:
        out_dict[y.split("'")[1]]=conc.ds("RESULT1:"+y)[i]
    entry_dict={**in_dict,**out_dict}
```

```
training_set=training_set.append(entry_dict, ignore_index
= True)
```

5. **Il DataFrame** `training_set`, ottenuto nella fase 1 e aggiornato nella precedente fase 4, **è infine trascritto nel file di riferimento**, da cui è stato letto. Per fare questo è utilizzato il metodo `to_csv` dell'oggetto `training_set`, a cui sono passati come argomenti il path del file su cui scrivere - nel nostro caso sovrascrivere - che è salvato nella variabile `training_set_path`, il separatore delle colonne `"\t"` - cioè la tabulazione come per il metodo di lettura - e il flag `index=False` che indica di non inserire il nome (l'indice) delle righe. Il file di training set è infatti predisposto in modo tale da avere nomi di colonne, ma non di righe. L'istruzione per l'aggiornamento del training set è la seguente:
   `training_set.to_csv(training_set_path, sep="\t", index=False)`
   Infine è mostrato all'utente un popup che segnala il completamento dell'esecuzione del parser, sfruttando il modulo `ctypes` importato come `ct`:
   ```
   ct.windll.user32.MessageBoxW(0, "Training Set updated
   successfully!",
   "AITV Parser", 0)
   ```

3.4 Validazione dei risultati

Gli elementi architetturali realizzati che sono stati descritti finora rappresentano una solida struttura per le attività di ricerca in sala prova, poiché si basano su un'implementazione del test realizzata in modo confrontabile nei due sistemi d'automazione, su un sistema d'archiviazione dei risultati centralizzato e caratterizzato da una comune struttura dei dati, infine su una post-elaborazione coerente con l'algoritmo di test e del tutto *automation-neutral*. A completamento di questa architettura è posto **AITV** (Artificial Intelligence Test Validator), uno strumento di validazione dei risultati ottenuti dal post-processing, mediante tecniche di Intelligenza Artificiale.
L'utente, giunto al termine dell'elaborazione dei risultati, può dunque validarli mediante l'esecuzione del job Concerto relativo ad AITV e successivamente inserirli nel training set, insieme ai parametri preliminari, per mezzo del job Concerto relativo al Parser. Mentre il secondo è stato descritto nel paragrafo precedente, si procederà ora ad analizzare struttura e caratteristiche della validazione con AITV.
Generalmente sono adoperate metodologie di validazione dei risultati basate su calcoli che derivano da studi sui fenomeni fisici e chimici associati alle grandezze risultanti da validare e con il supporto di AITV non si pretende assolutamente di fare a meno di tali tecniche. La raccolta sistematica dei risultati dei test - già validati - permette di avere dati sufficienti per effettuare una predizione di un risultato, sulla base dei parametri di input memorizzati nella chiave `V_IN`. La

scelta delle variabili in base alle quali effettuare la predizione è affidata all'utente, ma una volta presentato l'esito di tale processo - il valore stimato in sé e la differenza rispetto al valore realmente calcolato - è possibile utilizzare nuovamente il tool modificando le variabili di input e i parametri della predizione. L'obiettivo è dunque fornire uno strumento flessibile e personalizzabile, a vantaggio di utenti esperti e dotati delle competenze necessarie a determinare i migliori candidati per costituire l'input della predizione e che, di conseguenza, saranno allo stesso modo in grado di fare un uso efficace dello strumento. Si può giungere quindi a stabilire la validità di un risultato se si ottiene almeno una predizione vicina al valore realmente restituito dai calcoli della post-elaborazione, ma se invece i valori stimati da AITV si discostano molto da questi ultimi, si potrebbe sollevare l'ipotesi di un errore nella catena di misura, attuazione e calcolo. Naturalmente è necessario comprendere a pieno le modalità di parametrizzazione del modello predittivo, che influenzano le prestazioni di AITV tanto quanto la selezione delle feature di input più opportune. A tale scopo è dunque offerta nel prossimo paragrafo 3.4.1 una panoramica della **Regressione Lineare con il metodo del Gradient Descent**, cioè l'algoritmo di **Machine Learning** su cui si basa il modello predittivo di AITV. L'implementazione vera e propria in un job Concerto che lo metta in pratica è invece descritta nel successivo paragrafo 3.4.2.
É bene sottolineare, infine, che uno dei risultati che ci si auspica per questo progetto, è la scoperta in futuro dell'esistenza di correlazioni sottovalutate tra alcune grandezze acquisite e i risultati delle prove, mediante il loro impiego per la predizione.

3.4.1 Regressione Lineare con Gradient Descent

La disciplina del **Machine Learning** definisce numerose metodologie per la realizzazione di sistemi in grado di apprendere da un insieme di dati, che differiscono tra loro principalmente per scopo e approccio. In particolare siamo interessati alla creazione di un algoritmo in grado di trovare e stimare una relazione tra due variabili, data una collezione di coppie di loro valori. Immaginando di avere un sistema che produca una grandezza d'uscita y, data una grandezza d'ingresso x, un insieme di combinazioni input-output ottenute dal sistema costituisce il **training set** per il modello predittivo e ogni suo elemento è chiamato **training case**. L'apprendimento su tale insieme ha lo scopo di determinare una **funzione ipotesi** $h(x)$, che possa rappresentare la relazione tra x e y e che sia dunque in grado di stimare il valore di y, dato un nuovo valore di x. Questo meccanismo descrive per sommi capi il funzionamento dei sistemi di Machine Learning appartenenti alla categoria del **Supervised Learning**, a cui afferiscono le tecniche di **Regressione Lineare** e **Regressione Logistica** (o **Classificazione**). Come suggeriscono i nomi, la differenza tra i due tipi di sistema consiste nella differente natura della variabile di output da stimare, che nel caso della Regressione Lineare è appunto una grandezza reale e dunque "lineare", mentre per la

Regressione Logistica si tratta di una grandezza discreta e dunque una "classe". Secondo un'altra classificazione accademica dei sistemi di Machine Learning, si può dire che la Regressione Lineare e la Classificazione appartengono ai **Modelli Discriminativi**, che cercano la probabilità condizionata $p(y|x)$, con x ingresso e y uscita.
Nel merito degli obiettivi attuali di questo progetto, siamo in particolare interessati all'utilizzo della tecnica della **Regressione Lineare**, dal momento che la quasi totalità dei classici risultati dei test è riconducibile a grandezze reali. Ad ogni modo è comunque molto ampio l'insieme di elementi comuni anche alla Classificazione e pertanto l'analisi che segue è in gran parte utile anche alla sua comprensione. Peraltro tra le idee alla base di sviluppi futuri di questo sistema, elencate nel capitolo 5, figura anche la proposta di impiego delle tecniche di Classificazione, tra le quali si contemplano anche Reti Neurali e Macchine a Supporto Vettoriale.
Si specifica inoltre che l'integrazione di questo algoritmo di Intelligenza Artificiale non presuppone la creazione di un *Machine Learning System* completo, ma sono state comunque adoperate diverse tecniche che rendono l'approccio il più possibile approfondito e robusto. Per esempio non è attualmente opportuno ricorrere alla tecnica di validazione del modello su un **test set** separato o mediante la cross validazione sullo stesso training set, dal momento che i dati collezionati per ciascuna prova non sempre sono in numero sufficiente per una suddivisione. Anche questo aspetto sarà discusso nel capitolo 5.
Si concentra dunque il focus sulla Regressione Lineare, che sarà adoperata per ottenere una predizione di un valore risultante del test da confrontare con il risultato realmente calcolato. Per cominciare si consideri il caso di un sistema caratterizzato da una sola variabile di input x e una d'uscita y. L'obiettivo consiste nella definizione di una funzione ipotesi $h_\theta(x)$ che rappresenti la relazione tra x e y come una retta così definita:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

dove θ_0, detto **bias** rappresenta l'intercetta all'origine della retta e θ_1 è il coefficiente angolare; possiamo considerare i due parametri come elementi del vettore colonna Θ. Un'espressione di $h_\theta(x)$ più generale, che prevede l'utilizzo di n variabili di input è invece la seguente:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$$

e in questo caso il vettore colonna Θ comprenderà tutti i relativi parametri θ_0, θ_1, θ_2, fino a θ_n.
Per determinare i valori ottimali dei θ_k, bisogna definire una **funzione di costo** $J(\Theta)$ che fornisce un indice dell'errore relativo all'ipotesi (h_θ) in funzione dei θ_k attuali e dunque in un certo senso misura quanto la definizione di h_θ si discosta

da quella ottimale. Tra le diverse possibili espressioni di $J(\Theta)$, si utilizza la seguente:

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

dove:

- m è il numero di istanze del training set;
- $\frac{1}{2m}$ è il fattore di normalizzazione (il 2 serve a semplificare l'espressione della derivata di $J(\Theta)$, che vedremo più avanti);
- $(h_\theta(x^{(i)}) - y^{(i)})$ è l'errore tra l'i-esimo valore stimato - a partire dall'i-esimo valore di input $x^{(i)}$ - e l'i-esimo valore vero $y^{(i)}$ del training set.

Determinare la funzione h_θ ottima significa dunque trovare i θ_k che minimizzano la funzione di costo $J(\Theta)$:

$$\Theta_{min} = argmin_\Theta(J(\Theta))$$

Naturalmente l'ideale sarebbe che la funzione $J(\Theta)$ sia convessa, in modo tale che abbia un solo minimo globale in corrispondenza del quale si possa considerare ottima la funzione h_θ, ma in realtà questa circostanza è piuttosto rara. Per questa ragione è necessario utilizzare l'algoritmo chiamato **Gradient Descent**, che consiste nel muovere iterativamente i θ_k nello spazio di ricerca, incrementandoli di una quantità che dipende dal gradiente ∇ della funzione costo. L'aggiornamento dei θ_k termina idealmente quando si è raggiunto il minimo globale di $J(\Theta)$, ma dato che potrebbe essere una condizione temporalmente dispendiosa e talvolta impossibile da ottenere, si poassono imporre quattro condizioni d'arresto dell'algoritmo:

- **Relative Tolerance**: la differenza tra due gradienti consecutivi è sotto una certa soglia;
- **Absolute Tolerance**: il valore di $J(\Theta)$ è sotto una certa soglia e assume dunque un valore ritenuto sufficiente;
- **Max Iterations**: si raggiunge un numero massimo di iterazioni consentite;
- **Gradient Norm Tolerance**: la norma del gradiente è inferiore ad una certa soglia.

Nel caso di AITV, si è scelto di utilizzare l'opzione di **Max Iterations** per imporre l'interruzione dell'algoritmo, poiché le scelte da parte dell'utente per la predizione, in termini di variabili di input e parametri, potrebbero fortemente condizionare le prestazioni del Gradient Descent e la pur robusta piattaforma di

calcolo di Concerto potrebbe non sostenere un numero eccessivo di iterazioni.
Come anticipato, l'incremento dei parametri θ_k dipende dal gradiente ∇ della funzione costo e in particolare per ogni θ_k:

$$\theta_k^{new} = \theta_k^{old} - \alpha \frac{\partial J(\Theta)}{\partial \theta_k}$$

dove:

- α è il **Learning Rate**, che può assumere un valore compreso tra 0 e 1;
- $\frac{\partial J(\Theta)}{\partial \theta_k}$ è la derivata parziale di $J(\Theta)$ rispetto a θ_k.

Il valore di α determina il "passo" del Gradient Descent, ovvero l'entità del movimento dei θ_k ad ogni step. Un α troppo elevato porterebbe il sistema a divergere, muovendo i θ_k via via di una quantità sempre più grande a causa del crescente valore di $\frac{\partial J(\Theta)}{\partial \theta_k}$. D'altro canto però un α troppo piccolo renderebbe i movimenti estremamente ridotti e di conseguenza prolungherebbe eccessivamente i tempi di ricerca. La scelta del Learning Rate opportuno è dunque un requisito fondamentale, insieme alla selezione del parametro di input opportuno, perché AITV effettui una previsione ottimale.
Tralasciando tutti i calcoli necessari ad ottenerla, si riporta di seguito la generica espressione della derivata di $J(\Theta)$, per il caso di Regressione Lineare **Multivariate**, cioè basata su più di una variabile di input:

$$\frac{\partial J(\Theta)}{\partial \theta_k} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_k^{(i)}$$

dove $x_k^{(i)}$ è il valore i-esimo della k-esima variabile di input.
Si specifica inoltre che l'aggiornamento dei θ_k può seguire due metodologie, corrispondenti ad altrettante versioni del Gradient Descent:

- **Batch**, che aggiorna ogni θ_k sulla base di tutti i campioni del training set, computazionalmente più lento ma in grado di assicurare la convergenza;
- **Stocastico**, che aggiorna ogni θ_k sulla base di un solo campione del training set alla volta, computazionalmente più rapido ma senza garanzie di convergenza (dato che i movimenti risultano più piccoli e non necessariamente orientati verso il minimo della funzione $J(\Theta)$.

La versione adoperata per AITV è quella **Batch**, che utilizza ad ogni iterazione tutti i campioni del training set per aggiornare ciascun θ_k. Ad ogni modo entrambe le tipologie di Gradient Descent si basano su un'inizializzazione randomica dei valori dei θ_k, dai quali ha inizio l'algoritmo di ricerca dell'ipotesi ottimale.
Le equazioni descritte possono essere trasformate in *forma matriciale*, ottenendo

una rappresentazione più compatta. L'espressione in tale forma della funzione costo per esempio è la seguente:

$$J(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m}(X\Theta - Y)^T(X\Theta - Y)$$

Come vedremo nel prossimo paragrafo, il modulo Python **NumPy** mette a disposizione diversi metodi grazie ai quali è possibile implementare molti calcoli in questa forma.
Un passaggio fondamentale preliminare all'apprendimento del modello mediante l'algoritmo Gradient Descent, è la normalizzazione delle variabili d'ingresso, cioè la procedura detta **Feature Scaling**. Le diverse feature di input potrebbero presentare range di valori molto diversi tra loro e di conseguenza i Learning Rate ideali potrebbero variare di molto da variabile a variabile. Un metodo comune per effettuare quest'operazione è la **Min-Max Normalization**, che forza i valori in un intervallo $[a, b]$ scelto, esponendo però il modello al rischio di incontrare al momento della predizione dei valori di input al di fuori di tale intervallo. In questo progetto invece è stata adoperata la **Z-Score Normalization**, molto più robusta anche in merito alla presenza di *outlier*, cioè i rari valori molto distanti dalla media. Essa produce una distribuzione a media nulla per mezzo della sottrazione del valor medio e dividendo per la deviazione standard, senza quindi aver bisogno di definire un intervallo obiettivo. Ciascun valore x di ogni variabile di input va quindi trasformato in un nuovo valore z così definito:

$$z = \frac{(x - \mu)}{\sigma}$$

dove:

- μ è la media dei valori della variabile x;
- σ è la deviazione standard dei valori della variabile x.

Infine è possibile misurare le prestazioni del modello di Regressione Lineare per mezzo di 3 metriche, che permettono di valutare la risposta del sistema su un set di m campioni:

- **Mean Absolute Error** (**MAE**), che è tendenzialmente più robusto dell'indicatore RMSE riguardo alla presenza di eventuali *outlier*:

  $$MAE = \frac{1}{m}\sum_{i=1}^{m}|h(x^{(i)}) - y^{(i)}|$$

  Questo valore indica dunque quanto è mediamente distante il valore predetto su ogni training case da quello reale della variabile di output, in termini assoluti.

- **Mean Squared Error** (**MSE**), che ha il vantaggio di penalizzare errori molto grandi ma è più sensibile rispetto al MAE agli *outlier*:

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2$$

  Questo indice eleva al quadrato l'errore calcolato per ogni training set e per questo motivo conferisce un peso maggiore a differenze molto grandi in valore assoluto.

- **Root Mean Squared Error** (**RMSE**), che ha un significato analogo al MSE, ma ha un andamento diverso per modelli basati su gradiente:

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2} = \sqrt{MSE}$$

  Questo indice è dunque ottenuto semplicemente come radice quadrata del valore di MSE.

Per AITV sono stati implementati tutti i tre parametri di valutazione presentati, ma nel leggere i valori di questi indici va tenuto presente il loro significato, come sarà descritto in merito al caso di studio nel capitolo 4.

3.4.2 AITV: Artificial Intelligence Test Validator

La realizzazione dello strumento di validazione AITV, secondo i dettagli teorici espressi nel paragrafo precedente, consiste nell'implementazione di uno script in linguaggio Python, con integrazioni di script ausiliari in linguaggio Concerto e prevede inoltre l'impiego di altre tecniche di presentazione dei dati tipiche della piattaforma di post-elaborazione AVL. Anche in questo caso, come per il paragrafo relativo al Parser, saranno ritenuti assodati molti concetti relativi alla programmazione in Concerto e a funzioni e metodi già incontrati nelle precedenti descrizioni.
Sebbene sia interamente realizzato in linguaggio Python, che garantisce l'accesso a metodi indispensabili per l'implementazione dei calcoli introdotti nel paragrafo precedente, **AITV** sfrutta il richiamo di due script `csf` per la gestione dell'interfaccia utente. Inoltre perché potesse essere registrato come **job** di Concerto, è stato necessario realizzare - analogamente al Parser - lo script `AITV.csf` che semplicemente lancia l'esecuzione di `AITV.py`, che costituisce la vera e propria implementazione del validatore. É `AITV.csf` infine ad essere registrato come job. Il codice in esso contenuto è di seguito riportato:

```
Description:  AITV Validation
```

```
startPath = %CWF_AdaMoFilePath
startPath = StrReplaceAll(startPath,"\\","\")
ExecutePythonScript(startPath + "lib\AITV.py")
```

Dove la direttiva `Description` indica il titolo del Job (com'è visibile in fig. 3.17), mentre la funzione `ExecutePythonScript` lancia lo script `AITV.py`, posizionato nella sub-folder `lib` della directory target del datasource `AdaMo_Files`, il cui path è memorizzato nella User Variable `%CWF_AdaMoFilePath`. Quest'ultima, dal momento che è stata concepita per essere usata nello script ausiliario Python `RegHandler.py` del FileConverter, è una stringa strutturata con doppi \\ e per questa ragione è necessaria una conversione perché sia fruibile anche in linguaggio Concerto, per il quale il \ non è un carattere speciale, a differenza di Python.
Si analizza ora il vero e proprio algoritmo di validazione, implementato in `AITV.py`, il quale effettua una predizione basata su una **Regressione Lineare con Gradient Descent**, che utilizza come input **una o due variabili** selezionate dall'utente tra quelle disponibili nella chiave `V_IN` per stimare il valore del risultato scelto. La sua logica di funzionamento si basa sul fatto che al momento dell'esecuzione del job sia aperto il file di risultati da validare e applicato il layout di post-elaborazione. Si ribadisce che è fondamentale che il test result sia aperto mediante l'opportuno Datasource relativo alla sala prova di provenienza, perché siano correttamente associate le formule responsabili del calcolo dei risultati da validare.
Per questo codice Python è stato necessario importare i seguenti moduli:

- il modulo `concerto` è necessario per la gestione dei dataset del file di risultati con Alias `RESULT1` e di altri file ausiliari come quello del training set;
- il modulo `numpy` è mandatorio per la gestione dei dataset per mezzo del metodo `concerto.ds`, che restituisce il dataset argomento come un array di NumPy, ma è al contempo fondamentale per implementare i calcoli necessari al Gradient Descent;
- il modulo `pandas` è mandatorio per la gestione dei dataset per mezzo del metodo `concerto.dsframe`, che restituisce i dataset argomento come un DataFrame di Pandas, ma è al contempo fondamentale per alcune strutture utili alla gestione del training set;
- il modulo `time`, utilizzato per il metodo `sleep` che permette di sospendere l'esecuzione dello script per alcuni secondi.

Le istruzioni utilizzate per l'importazione e la creazione delle relative variabili `conc`, `np`, `pd` e `time`, sono di seguito riportate.

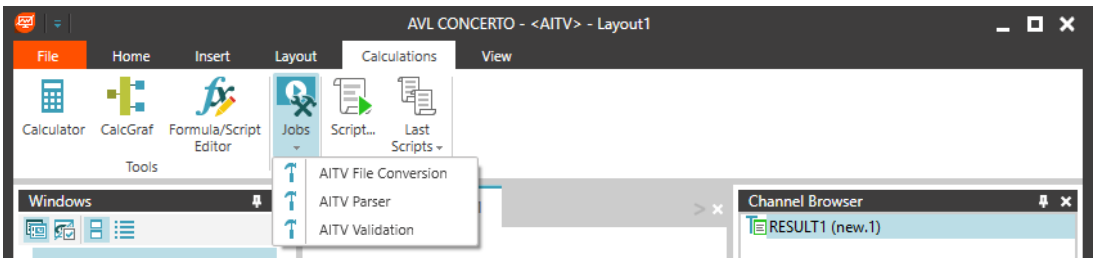

Figura 3.17: Esecuzione del job di Validazione AITV.

```
import concerto as conc
import numpy as np
import pandas as pd
import time
```

Innanzitutto è necessario definire alcune funzioni che si rendono necessarie per l'implementazione dell'algoritmo di validazione. La loro struttura e il loro scopo è di seguito riportato.

- `featureNormalize` effettua la procedura di normalizzazione di una variabile d'ingresso con il metodo della **Z-Score Normalization**, che produce una distribuzione a media nulla per mezzo della sottrazione del valor medio e dividendo per la deviazione standard:

$$z = \frac{(x - \mu)}{\sigma}$$

  Per fare questo è necessario che la funzione riceva come argomento l'array `x` dei valori della variabile di input, contenuti nel file di training set. Da esso è calcolata la media `m` dei valori con il metodo `mean` di NumPy e analogalmente la deviazione standard `s` con il metodo `std`, anch'essa di NumPy. Infine, per ogni elemento dell'array `x`, sono effettuate le operazioni di sottrazione e divisione per produrre l'array risultate `x_norm`, che infine è restituito dalla funzione con la keyword `return`.

```
def featureNormalize(x):
    m = np.mean(x)
    s = np.std(x)
    x_norm = (x-m)/s
    return x_norm, m, s
```

  Si noti che inoltre sono restituiti anche il valore della media `m` e della deviazione standard `s` in modo tale che sia possibile - come vedremo - normalizzare anche il valore del parametro di ingresso attuale, tramite il

quale è effettuata la predizione del risultato richiesto.

- `computeCost` implementa il calcolo della funzione di costo $J(\Theta)$, secondo la forma matriciale

$$J(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m}(X\Theta - Y)^T(X\Theta - Y)$$

  Si ricorda che questo valore stabilisce quanto la funzione ipotesi attuale si discosti da quella ottimale ed è alla base del funzionamento del Gradient Descent. Per il calcolo di $J(\Theta)$ è necessario passare alla funzione tre parametri:

  1. l'array bidimensionale `X` corrispondente alla matrice di input X di dimensione m`x`$(n+1)$ (con m uguale al numero dei training case e n uguale al numero di variabili di input scelte dall'utente, che può valere dunque 1 o 2), composta da una prima colonna di `1` - corrispondenti al coefficiente 1 associto al **bias** θ_0 - seguita dalle colonne dei valori delle n variabili di input estratte dal file del training set (una sola colonna nel caso di una sola variabile);
  2. l'array monodimensionale `y` corrispondente al vettore **colonna** contenente gli m valori di output reali del training set;
  3. l'array monodimensionale `theta` corrispondente al vettore dei θ_k attuali, che caratterizzano la funzione ipotesi attuale e sulla base dei quali va calcolata la funzione di costo; è importante tenere a mente che si tratta di un vettore **riga**, che sarà infatti trasposto in un vettore colonna per il prodotto scalare.

  Per l'implementazione del calcolo matriciale, è stato sfruttato il modulo NumPy per le due funzioni `dot`, che effettua il prodotto righe per colonne (*inner product*) di due array che naturalmente devono essere opportunamente dimensionati, e `transpose` che restituisce un array corrispondente alla matrice trasposta di quella corrispondente all'array passato come argomento. In particolare il calcolo del prodotto $X\Theta$ è necessario trasporre l'array `theta` con `np.transpose` prima di passarlo come parametro alla funzione `np.dot` che effettua il prodotto righe per colonne, dal momento che la variabile `theta` è un array riga di dimensione 1`x`$(n+1)$. Si ottiene in questo modo un array NumPy corrispondente ad una matrice di dimensione m`x`1. Ne consegue dunque che il termine $(X\Theta - Y)$ si ottiene semplicemente effettuando una sottrazione elemento per elemento delle due matrici di dimensione m`x`1 ed è implementato in Python in questo modo:
  `term = np.dot(X,np.transpose(theta))-y`
  La matrice ottenuta, corrispondente alla variabile `term`, va successivamente trasposta per ottenere $(X\Theta - Y)^T$, che sarà infine moltiplicato per la

matrice $(X\Theta - Y)$ con il prodotto righe per colonne:
`np.dot(np.transpose(term),term)-y)`
che moltiplicato il fattore $\frac{1}{2m}$ restituisce infine il valore di $J(\Theta)$:
`J = np.dot(np.transpose(term),term)-y)/(2*m)`
In sintesi la funzione `computeCost` è così definita:

```
def computeCost(X, y, theta):
    m = len(y)
    term = np.dot(X,np.transpose(theta))-y
    J = np.dot(np.transpose(term),term)-y)/(2*m)
    print(J)
    return J
```

Si noti che la variabile `m` rappresenta il numero di campioni del training set ed è infatti ottenuta dalla funzione built-in `len` che restituisce la dimensione dell'array `y` passato come parametro. Infine l'istruzione `print(J)` stampa sulla finestra dei messaggi di Concerto il valore di `J`.

- `gradientDescent` implementa l'algoritmo di ricerca dei valori θ_k ottimali mediante il Gradient Descent, come è stato descritto nel paragrafo precedente. Le due impostazioni fondamentali per effettuare questa procedura, entrambe stabilite dall'utente, sono costituite dal valore del Learning Rate α e dal numero di iterazioni richieste per l'algoritmo di Gradient Descent. Questi parametri sono passati come argomenti `alpha` e `num_iter` alla funzione, insieme agli stessi tre argomenti `X`, `y` e `theta` utilizzati da `computeCost`. Per realizzare l'algoritmo ci si è serviti di tre for loop annidati:

  1. il loop più esterno serve a iterare il processo per il numero di volte richiesto dall'utente, memorizzato nella variabile `num_iters`;
  2. il loop intermedio itera sui diversi θ_k per l'aggiornamento di ciascuno di loro;
  3. il loop più interno effettua la sommatoria per l'incremento del θ_k attuale.

  Si ricorda infatti che l'aggiornamento dei valori dei θ_k, effettuata secondo la definizione

$$\theta_k^{new} = \theta_k^{old} - \alpha \frac{\partial J(\Theta)}{\partial \theta_k}$$

  si basa su un metodo **Batch**, che dunque sfrutta tutte le istanze del training set per il calcolo del gradiente ∇:

$$\frac{\partial J(\Theta)}{\partial \theta_k} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_k^{(i)}$$

Il codice della funzione è dunque il seguente:

```
def gradientDescent(X, y, theta, alpha, num_iters):
     m = len(y)
     for iter in range(num_iters):
          for k in range(len(theta)):
               incr=0
               for i in range(m):
                    incr = incr + (np.dot(X[i,],np.transpose(theta))
-y[i])*X[i,k]
               theta[k]=theta[k]-(alpha*incr/m)
          print("Actual value of Cost Function J:")
          J_act = computeCost(X,y,theta)
     return theta
```

Si noti che anche in questo caso è stato salvato in `m` il numero di training case, mediante la funzione `len` applicata sull'array degli output `y`. Inoltre per l'implementazione dei loop è stata adoperata anche in questo la funzione built-in `range` (già descritta nel merito del Parser), alla quale è passato il numero di iterazioni `num_iters` per il loop più esterno, la dimensione del vettore riga corrispondente all'array `theta` per iterare sui diversi θ_k nel loop intermedio, il numero di istanze del training set `m` nel loop più interno. In particolare il calcolo del gradiente ∇ per ciascun θ_k si basa sul calcolo della sommatoria $\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})x_k^{(i)}$ che è effettuato aggiungendo alla variabile `incr` (inizializzata a 0 fuori dall'ultimo loop) un contributo calcolato in ogni iterazione dell'ultimo ciclo for come $(h_\theta(x^{(i)}) - y^{(i)})x_k^{(i)}$, che in Python si può implementare con l'istruzione:
`incr = incr + (np.dot(X[i,],np.transpose(theta))-y[i])*X[i,k]`
Nella quale il vettore `X[i,]` corrispondente all'`i`-esima riga della matrice `X` e contenente il coefficiente 1 per il bias seguito dai valori `i`-esimi delle variabili di input, è moltiplicato (con prodotto righe per colonne) per la trasposta di `theta` ottenuta con la funzione `np.transpose`. Questo prodotto tra una matrice 1x$(n+1)$ (`X[i,]`) e una $(n+1)$x1 (`np.transpose(theta)` (si tenga a mente che n è il numero di variabili di input selezionate dall'utente e può valere 1 o 2) ha come risultato uno scalare a cui è dunque sottratto il valore `i`-esimo della variabile di output. L'intero termine $(h_\theta(x^{(i)}) - y^{(i)})$ così ottenuto, è moltiplicato per $x_k^{(i)}$, cioè l'i-esimo valore della k-esima variabile di input, che corrisponde all'elemento `X[i,k]`. naturalmente nel caso di `i` uguale a 0, per qualsiasi `k`, il valore di `X[i,k]` sarà 1, in quanto coefficiente di bias.
Infine è calcolato - e mostrato nella finestra di messaggi di Concerto dall'apposita istruzione contenuta nel corpo di `ComputeCost` - il valore della

funzione di costo $J(\Theta)$ al termine di ognuna delle `num_iters` iterazioni, per tenere traccia dell'evoluzione della funzione ipotesi $h_\theta(x)$ nel corso del Gradient Descent. Tale valore, assegnato alla variabile `J_act`, è calcolato mediante la funzione `computeCost` precedentemente definita, alla quale sono passati l'array corrispondente alla matrice degli ingressi `X`, l'array corrispondente alla colonna delle uscite `y` e l'array corrispondente alla riga degli attuali θ_k `theta`.

- `computeErrors` effettua il calcolo degli indici che permettono di valutare la funzione ipotesi $h_\theta(x)$ in base all'errore di predizione commesso su tutti gli elementi del training case. In particolare i tre parametri sono il **Mean Absolute Error** (**MAE**), il **Mean Squared Error** (**MSE**) e il **Root Mean Squared Error** (**RMSE**), descritti nel capitolo precedente e così definiti:

$$MAE = \frac{1}{m}\sum_{i=1}^{m}|h(x^{(i)}) - y^{(i)}|$$

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2$$

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2} = \sqrt{MSE}$$

  Questa funzione necessita degli stessi argomenti di `computeCost`, cioè l'array corrispondente alla matrice degli ingressi `X`, l'array corrispondente alla colonna delle uscite `y` e l'array corrispondente alla riga degli attuali θ_k `theta`. É innanzitutto assegnato anche in questo caso ad `m` il numero di campioni, che è pari alla dimensione di `y`, ed è implementato un ciclo for su tutte le istanze del training set per mezzo della funzione `range` a cui è dunque passato il parametro `m`. All'interno di questo loop è assegnato alla variabile `error` l'errore $h(x^{(i)}) - y^{(i)}$ relativo al training case attuale ed è utilizzato per realizzare le sommatorie presenti nelle espressioni di MAE e MSE, incrementando le relative variabili `MAE` e `MSE` (inizializzate a 0 fuori dal loop) ad ogni iterazione rispettivamente con il valore assoluto di `error` - per mezzo della funzione `np.absolute` - e con il quadrato di `error` - per mezzo della funzione `np.power`. Analogamente a quanto visto per `gradientDescent`, anche in questo caso il valore dell'espressione $h(x^{(i)}) - y^{(i)}$, assegnato a `error`, è calcolato per mezzo dell'istruzione
  `error = np.dot(X[i,],np.transpose(theta))-y[i]`
  Infine i valori così ottenuti al termine del loop sono divisi per `m` per completare il calcolo di MAE e MSE, mentre l'indice RMSE è calcolato semplicemente come radice del valore di MSE, utilizzando la funzione `np.sqrt`. Il codice della funzione `computeErrors` è il seguente:

```
def computeErrors(X, y, theta):
     m = len(y)
     MAE = 0
     MSE = 0
     for i in range(0,m):
          error = np.dot(X[i,],np.transpose(theta))-y[i]
          MAE = MAE + np.absolute(error)
          MSE = MSE + np.power(error)
     MAE = MAE/m
     MSE = MSE/m
     RMSE = np.sqrt(MSE)
     print("The Mean Absolute Error (MAE) calculated over the
training set is:", MAE)
     print("The Mean Squared Error (MSE) calculated over the
training set is:", MSE)
     print("The Root Mean Squared Error (RMSE) calculated over
the
training set is:", RMSE)
     return MAE, MSE, RMSE
```

Anche in questo caso i valori ottenuti, prima di essere restituiti con la keyword `return`, sono anche stampati sulla finestra dei messaggi di Concerto.

Le quattro funzioni precedentemente definite sono adoperate per realizzare l'algoritmo di validazione di AITV, la cui struttura è riportata in fig. 3.18. Prima di procedere all'analisi dei vari step, si specifica che in questo contesto sono state create e adoperate altre cinque User Variable di Concerto, tutte associate al Work Environment **AITV** in uso, necessarie per lo scambio di parametri tra gli ambienti di scripting Concerto e Python:

- `%CWF_AITV_in`, che indica il primo parametro di input selezionato dall'utente, in base al quale è effettuata la predizione;
- `%CWF_AITV_in2`, che indica l'eventuale secondo parametro di input selezionato dall'utente, ma è una stringa vuota nel caso in cui sia richiesto l'uso di un solo ingresso;
- `%CWF_AITV_out`, che indica il risultato della post-elaborazione da validare;
- `%CWF_AITV_SelPoint`, che rappresenta l'indice del dataset `%CWF_AITV_out` selezionato dall'utente per la validazione;
- `%CWF_AITV_alpha`, che indica il Learning Rate selezionato dall'utente per la predizione;

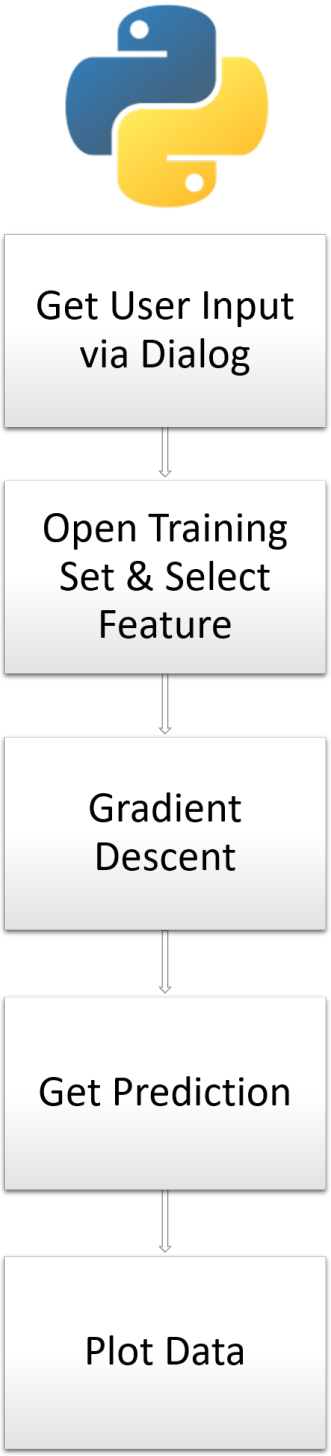

Figura 3.18: Struttura dell'algoritmo di validazione AITV.

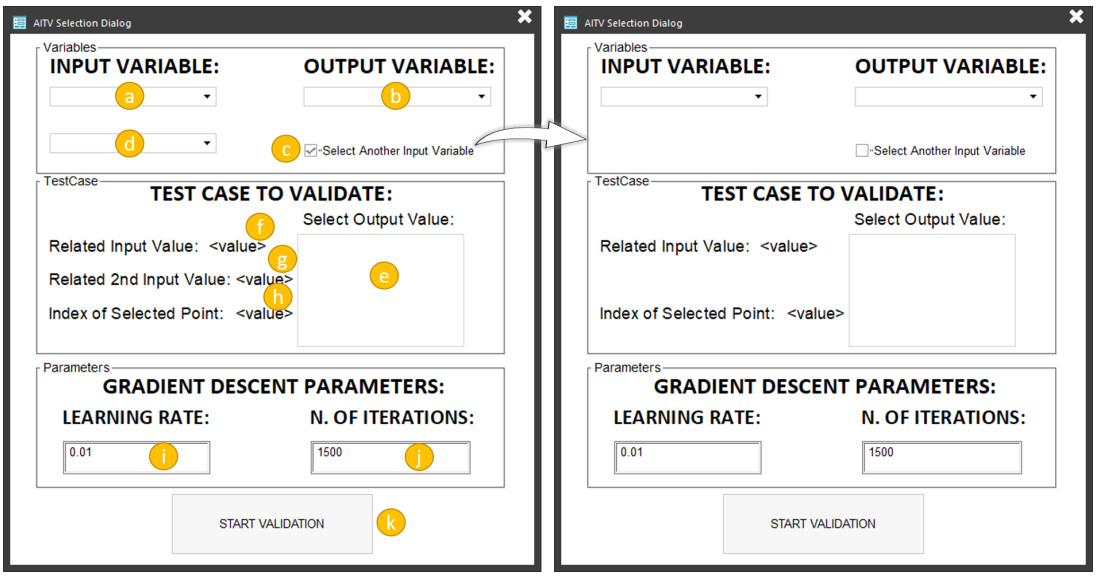

Figura 3.19: Dialog di AITV per la richiesta di input all'utente.

- `%CWF_AITV_nIters`, che indica il numero di iterazioni del Gradient Descent, richiesto dall'utente;
- `%CWF_AITV_syn`, che serve a sincronizzare l'esecuzione di `AITV.py` con lo script Concerto che effettua il caricamento della finestra di dialogo con l'utente.

In riferimento alla fig. 3.18, sono di seguito descritte le cinque fasi che compongono la procedura di validazione.

1. Innanzitutto è necessario **ricevere dall'utente gli input** necessari a parametrizzare l'algoritmo, mediante una semplice **interfaccia grafica**, rappresentata da un **Dialog**. Nel paragrafo 3.3 sono stati introdotti gli elementi base che AVL Concerto mette a disposizione per la realizzazione dei layout di post-elaborazione, molti dei quali sono orientati alla costituzione di efficaci interfacce grafiche per l'interazione con l'utente, volte alla realizzazione di applicazioni complesse come quella rappresentata da questo progetto. In particolare in questo step ci si avvale di un **Dialog**, cioè un oggetto associabile ad una **Window**, che rappresenta una finestra di dialogo tramite la quale è possibile interagire con l'utente, attraverso diversi oggetti quali action button, listbox, check box e così via. A ciascuno di questi oggetti sono associati specifici *eventi* legati a transizioni di stato come ad esempio il cambio dell'elemento selezionato per un listbox o per un action button l'attimo in cui è premuto. Al verificarsi di un evento è possibile eseguire uno script associato, che può essere un file `csf` o `py`, oppure può essere uno script *embedded*, cioè associato direttamente all'oggetto e in particolare al suo evento. La finestra creata è stata salvata nella solita

cartella `lib` con il nome **AITV_Selection.cdg** ed è riportata in fig. 3.19, a cui si fa riferimento nell'analisi che segue dei vari oggetti che la compongono. Si specifica che tutti gli script associati agli eventi di questi oggetti, sono di tipo *embedded* e realizzati in linguaggio Concerto, indispensabile per accedere a librerie e classi necessarie per manipolare le finestre. Inoltre al Dialog è associato il nome `"AITV Selection Dialog"`, tramite il quale è possibile istanziare diversi oggetti utili.

(a) Il ComboBox **InputBox** è un menu a tendina a cui è stato associato il dataset `INPUTS:D'Inputs`, che fa riferimento alla lista dei canali di input tipici della prova eseguita, estratti dal relativo file nella cartella `lib`, e permette dunque all'utente di selezionare la prima variabile di input da adoperare per la predizione. Naturalmente è necessario che il file contenente la lista dei parametri di input sia aperto al momento del caricamento della finestra, ma questo dettaglio sarà meglio chiarito più avanti. L'oggetto grafico **InputBox** in ascolto dell'evento `OnSelChanged`, innescato al cambiamento della selezione tra i vari elementi, a cui è associato il seguente script:

```
dlg=SelWin("AITV Selection Dialog")
cb=dlg.SelObj("InputBox","COMBOBOX")
%CWF_AITV_in=cb.SelectedText
cb2=dlg.SelObj("InputBox2","COMBOBOX")
inDS=INPUTS:D'Inputs
newTxt=""
for i=1 to npoints(inDS)
     if not(inDS.y[i] = %CWF_AITV_in) then
          newTxt=newTxt + inDS.y[i] + StrChr(10)
     endif
next i
newTxt=StrErase(newTxt,StrLen(newTxt),1)
cb2.Text=newTxt
dlg.Paint()
```

Per mezzo della funzione `SelWin` della libreria **Script**, è possibile ottenere un oggetto della classe `Window` corrispondente alla finestra Dialog identificata dal nome `"AITV Selection Dialog"`, dopodiché tale oggetto è assegnato alla variabile `dlg`. In questo modo è possibile accedere a tutti i metodi della clase `Window` che permettono di parametrizzare, manipolare, aprire o chiudere le finestre. In questo caso `dlg` è utilizzato per selezionare l'oggetto corrispondente al ComboBox `InputBox`, per mezzo del metodo `SelObj`, al quale è passato il nome dell'oggetto target e la tipologia a cui afferisce (cioè `"COMBOBOX"`).

Questo metodo restituisce un oggetto della classe **Object**[12], che permette di parametrizzare e manipolare gli oggetti grafici. L'oggetto `cb` così ottenuto è usato per accedere al nuovo elemento selezionato dall'utente sulla ComboBox `InputBox`, disponibile grazie all'attributo `cb.SelectedText`, il cui valore (String) è assegnato alla User Variable `%CWF_AITV_in`, grazie alla quale è possibile accedere a questa stringa anche nello script Python `AITV.py`.
Successivamente è generata la lista delle rimanenti variabili di input, tra i quali l'utente potrebbe eventualmente selezionare il secondo dataset per la predizione. A tale scopo è istanziato l'oggetto `cb2` relativo al ComboBox `InputBox2` **(d)** utilizzato per la selezione dell'eventuale seconda variabile di input, utilizzando il metodo visto precedentemente. Con un for loop che scorre su tutti gli elementi del dataset `INPUTS:D'Inputs`, assegnato a `inDS`, è generata la stringa `newTxt` contenente tutti i nomi di canali diversi da quello appena selezionato in `InputBox`, separati dal carattere di *new line* che si può ottenere con la funzione `StrChr(ASCIIcode)` che restituisce il carattere ASCII corrispondente al codice passato come argomento (in questo caso 10). Questo loop genera tutti gli elementi del ComboBox `InputBox2`, aggiungendo come ultimo carattere della stringa un `StrChr(10)` di troppo, che è eliminato mediante la funzione `StrErase` della libreria **String**. Infine `newTxt` è assegnato all'attributo `Text` dell'oggetto `cb2` e, a seguito di un *refresh* della finestra con il metodo `dlg.Paint()`, il ComboBox `InputBox2` è parametrizzato correttamente.

(b) Il ComboBox **OutputBox**, in maniera del tutto analoga ad `InputBox`, è utilizzato per far selezionare all'utente l'output da validare, tra quelli calcolati dalla post-elaborazione ed elencati nel relativo file, che dev'essere anch'esso aperto al momento del caricamento della finestra Dialog. A questo ComboBox è dunque associato il dataset `OUTPUTS:D'Outputs` ed è in ascolto dell'evento `OnSelChanged`, al verificarsi del quale esegue questo script embedded:

```
dlg=SelWin("AITV Selection Dialog")
cb=dlg.SelObj("OutputBox","COMBOBOX")
%CWF_AITV_out=cb.SelectedText
lb=dlg.SelObj("CaseList", "LISTBOX")
lb.AddDS("RESULT1:"+%CWF_AITV_out)
```

Le prime tre righe, analogamente a `InbutBox`, selezionano la finestra Dialog, ottengono l'oggetto relativo al ComboBox `OutputBox` e assegnano il nuovo valore selezionato dall'utente, accessibile da `cb.Selec-`

[12]N.B. In questo caso il termine "object", usato come nome della classe, sta ad indicare gli "oggetti grafici".

tedText, alla User Variable %CWF_AITV_out. Successivamente è selezionato l'oggetto (e) corrispondente al ListBox CaseList, che permette all'utente di selezionare quale valore vuole validare, tra quelli appartenenti al dataset appena selezionato in OutputBox dall'elenco dei risultati disponibili. Dopo aver istanziato quest'oggetto lb, gli viene assegnato il dataset appena selezionato e salvato in %CWF_AITV_out, per mezzo del metodo lb.AddDS, a cui è passato il nome del dataset con il classico costrutto Alias:KEY'DATASET. Si ricorda che la lista dei parametri di Output, salvata nel relativo file, presenta per ogni elemento la stringa KEY'DATASET.

(c) Il CheckBox **SecondInput** permette all'utente di accedere alla selezione di una seconda variabile di input per la predizione e ha dunque la funzione di attivare o nascondere tutti gli oggetti relativi a essa. Questo semplice oggetto è in ascolto dell'evento OnClick, scatenato al variare dello stato del CheckBox, ad opera dell'utente. In corrispondenza di questa transizione è dunque eseguito il seguente script *embedded*:

```
dlg=SelWin("AITV Selection Dialog")
chb=dlg.SelObj("SecondInput","CHECKBOX")
cb=dlg.SelObj("InputBox2","COMBOBOX")
cb.Visible=chb.Checked
t_label=dlg.SelObj("xCaseLabel2","TEXT")
t_label.Visible=chb.Checked
t_value=dlg.SelObj("xCase2","TEXT")
t_value.Visible=chb.Checked
dlg.Paint()
if chb.Checked = 0 then
    %CWF_AITV_in2=""
endif
```

Anche in questo caso, tramite le prime due righe, è selezionata la finestra di Dialog e istanziato di conseguenza dlg, tramite il quale è selezionato l'oggetto relativo al CheckBox SecondInput assegnato alla variabile chb. É poi istanziato l'oggetto cb relativo alla ComboBox InputBox2, che si vuole nascondere nel caso in cui il CheckBox non sia spuntato. Per questo motivo è assegnato all'attributo cb.Visible il valore di chb.Checked, che restituisce 1 nel caso in cui il CheckBox relativo a chb sia spuntato e 0 altrimenti. Allo stesso modo sono mostrati o nascosti i due Text xCaseLabel2 ("Related 2nd Input Value:") e xCase2 (g), rispettivamente mediante gli oggetti t_label e t_value e infine, per rendere attive queste modifiche dinamiche, è necessario effettuare un refresh del Dialog mediante il metodo dlg.Paint(). La

parte destra della fig. 3.19 mostra l'aspetto della finestra nel caso in cui il CheckBox sia deselezionato.
Le ultime tre righe di questo script contengono un *if statement* necessario a svuotare la User Variable `%CWF_AITV_in2`, associata al ComboBox `InputBox2`, in modo tale che in `AITV.py` sia possibile determinare se l'utente abbia selezionato solo una variabile di input o due.

(d) Il ComboBox **InputBox2**, più volte citato nel merito degli oggetti precedenti, è un menu a tendina per la selezione della seconda variabile di input, nel caso in cui sia spuntato il CheckBox `SecondInput`. Come si è potuto evincere, non ha direttamente associato il dataset `INPUTS:D'Inputs`, diversamente da `InputBox`: il suo contenuto è creato dallo script associato all'evento `OnSelChanged` di `InputBox` eliminando dalla lista contenuta in `INPUTS:D'Inputs` il canale già selezionato come primo input. Anche questo oggetto è in ascolto dell'evento `OnSelChanged`, in corrispondenza del quale esegue questa procedura:

```
dlg=SelWin("AITV Selection Dialog")
cb=dlg.SelObj("InputBox2","COMBOBOX")
%CWF_AITV_in2=cb.SelectedText
```

In modo del tutto analogo a `InputBox`, è selezionato l'oggetto della finestra Dialog e successivamente quello del ComboBox `InputBox2`, tramite il quale si accede all'attributo `SelectedText`, il cui valore è infine assegnato alla User Variable `%CWF_AITV_in2`.

(e) Il ListBox **CaseList**, come anticipato, presenta la lista di tutti i punti del dataset risultato, selezionato dall'utente tramite `OutputBox`, e permette dunque all'utente di selezionare il valore da validare. Non ha un'associazione statica ad un dataset ma, come è stato descritto, essa varia in base alla selezione su `OutputBox`, per opera dello script associato a tale ComboBox. Anche **CaseList** è in ascolto dell'evento `OnSelChanged` e ha il seguente script associato:

```
dlg=SelWin("AITV Selection Dialog")
lb=dlg.SelObj("CaseList", "LISTBOX")
%CWF_AITV_SelPoint=lb.GetSelectedPoints()
t_index=dlg.SelObj("Index","TEXT")
t_index.Text=lb.GetSelectedPoints()
t_value=dlg.SelObj("xCase","TEXT")
x_case=DS("RESULT1:"+%CWF_AITV_in).y[%CWF_AITV_SelPoint]
t_value.Text=CStr(x_case)
t_value2=dlg.SelObj("xCase2","TEXT")
x_case2=DS("RESULT1:"+%CWF_AITV_in2).y[%CWF_AITV_SelPoint]
```

```
t_value2.Text=CStr(x_case2)
dlg.Paint
```

Le prime tre righe, ancora una volta, estraggono semplicemente l'input dell'utente e lo salvano nella corrispondente User Variable `%CWF_AITV_SelPoint`, ma in questo caso siamo interessati ad ottenere l'**indice** (di tipo Integer) e non del valore selezionato dall'utente e per questo motivo è adoperato il metodo `lb.GetSelectedPoints()`. Il codice successivo è invece finalizzato ad aggiornare tre oggetti Text: `Index` **(h)** con l'indice selezionato; `xCase` **(f)** con il valore - corrispondente all'indice `Index` - del dataset di input selezionato su `InputBox`; `xCase2` **(g)** con il corrispondente valore del secondo dataset di input selezionato su `InputBox`. Per fare quest'ultima operazione è adoperata la funzione `DS` della libreria **Standard**, che restituisce il dataset corrispondente alla stringa `Alias:KEY'DATASET` passata come parametro ed è infine utilizzato il metodo `y.[index]` dell'oggetto relativo a tale dataset, per estrarre il valore associato al punto scelto usando come indice `%CWF_AITV_SelPoint`. I valori di `x_case` e `x_case2`, ottenuti in questo modo, sono assegnati come stringhe alle property `Text` dei due oggetti `t_value` e `t_value2` relativi alle caselle testuali **(f)** e **(g)**, effettuando il casting dei valori verso il tipo String con la funzione `CStr` della libreria **String**. Infine è effettuato il refresh della finestra, tramite il metodo `dlg.Paint`, in modo tale da mostrare sui Text i valori aggiornati.

(f) Il Text **xCase**, manipolato dallo script dell'oggetto `CaseList`, non ha handler di eventi associati.

(g) Il Text **xCase2** è mostrato solo nel caso in cui il CheckBox `SecondInput` sia spuntato e manipolato dallo script dell'oggetto `CaseList`. Non ha handler di eventi associati.

(h) Il Text **Index**, manipolato dallo script dell'oggetto `CaseList`, non ha handler di eventi associati.

(i) Il TextBox **LearnRate** è utilizzato per permettere all'utente di inserire il valore di **Learning Rate** per l'algoritmo Gradient Descent. Questo tipo di oggetto consente di associare direttamente una User Variable, a cui è automaticamente assegnato un nuovo valore non appena l'utente aggiorna il TextBox. Per questo motivo è stato sufficiente associare a `LearnRate` la variabile `%CWF_AITV_alpha`, senza aver bisogno di implementare un ulteriore script.

(j) Il TextBox **nIters**, analogamente a `LearnRate`, permette all'utente di inserire il numero di iterazioni per l'algoritmo Gradient Descent e gli è stata associata la User Variable `%CWF_AITV_nIters`,

che viene automaticamente aggiornata con il meccanismo descritto precedentemente.

(k) Il Button **CONTINUE**, che presenta sulla superficie il testo `"START VALIDATION"`, consente di confermare i dati immessi dall'utente e chiudere il Dialog per proseguire con l'esecuzione dello script che l'ha aperto. Per associare a `CONTINUE` l'azione di chiusura della finestra, è sufficiente spuntare il checkbox `Exit Dialog Execution in Window`, tipico di questo tipo di oggetto e predisposto per questo meccanismo.

Questa doverosa premessa fornisce molti dettagli necessari a comprendere la struttura di questo primo step della validazione. Come è stato specificato, prima di aprire il Dialog è necessario caricare in Concerto i due file contenenti le liste dei parametri di input e dei risultati della prova effettuata, senza dimenticare che dev'essere già aperto con Alias `RESULT1` il file di risultati oggetto della validazione. Il codice di questa prima sezione è il seguente:

```
test_type = conc.ds("RESULT1:V_IN'Test_Type")[0]
startPath = conc.variables["%CWF_AdaMoFilePath"]
inp_file=conc.data.select_file("AdaMo_Files" +"\\lib\\"+
test_type+"_Inputs.txt")
out_file=conc.data.select_file("AdaMo_Files" +"\\lib\\"+
test_type+"_Outputs.txt")
inp_file.open("INPUTS")
out_file.open("OUTPUTS")
```

Dal primo punto[13] del dataset `Test_Type`, appartenente alla chiave `V_IN` del file di risultati `RESULT1`, è ottenuto il nome identificativo della tipologia di prova svolta ed è salvato nella variabile `test_type` come sempre con il metodo `conc.ds`. Successivamente è salvato in `startPath` il contenuto della User Variable `%CWF_AdaMoFilePath` che costituisce il punto di partenza per accedere alla cartella `lib` contenente script e file testuali di nostro interesse. Sono poi selezionati - con il metodo `conc.data.select_file` - i due file utilizzando il Datasource `AdaMo_Files`, che garantisce di trovare sotto la chiave `D` i dataset contenenti le liste di canali, istanziando così i due oggetti `inp_file` e `out_file` della classe `FileObject`. Infine è eseguito il metodo `open` di questi due oggetti specificando gli Alias `INPUTS` e `OUTPUTS` completando una procedura sostanzialmente analoga a quella vista per il Parser.

A questo punto è possibile caricare e mostrare all'utente il Dialog, ma per

[13] Si sceglie il primo punto per sicurezza, dato che sarà sempre presente sebbene non sappiamo *a priori* di quanti punti sia composta la chiave `V_IN`. I valori di `Test_Type` saranno comunque sempre tutti uguali per ogni punto.

fare questo è necessario ricorrere allo script ausiliario `LoadDialog.csf` in linguaggio Concerto, la cui esecuzione tuttavia procede parallelamente a quella di `AITV.py`. Per questo motivo è necessario **sincronizzare** i due thread, mediante la User Variable `%CWF_AITV_syn`, il cui valore è inizialmente `0` e viene impostato a `1` da `LoadDialog.csf` al termine della sua esecuzione. Per questo motivo è necessario un *while loop* di attesa e dunque questa sezione è così implementata:

```
conc.variables["%CWF_AITV_syn"]="0"
conc.execute_concerto_script(startPath+"lib\\LoadDialog.csf")
while conc.variables["%CWF_AITV_syn"] == 0:
    pass
conc.variables["%CWF_AITV_syn"]="0"
```

Il valore di `%CWF_AITV_syn` è inizialmente impostato `0` e al termine del loop di attesa è resettato a `0`, con l'accortezza di assegnare comunque un dato di tipo String a `conc.variables["var"]` come prescritto da AVL nella documentazione relativa al modulo `concerto`.
Passiamo ora ad analizzare la struttura di **LoadDialog.csf**, il cui codice è di seguito riportato:

```
startPath = %CWF_AdaMoFilePath
startPath = StrReplaceAll(startPath,"\\","\")
LoadWindow(startPath+"lib\AITV_Selection.cdg")
dlg=SelWin("AITV Selection Dialog")
dlg.Activate()
dlg.WaitForDialog()
dlg.Close()
%CWF_AITV_syn=1
```

Innanzitutto è determinato il path iniziale per la selezione della finestra, con la solita tecnica di sostituzione del doppio \\ con un solo \, dopodiché la finestra è caricata nel layout di post-elaborazione correntemente aperto, con la funzione `LoadWindow` della libreria **Script**. É poi assegnato alla variabile `dlg` l'oggetto corrispondente al Dialog, con la funzione `SelWin`, a cui è passato il nome identificativo `"AITV Selection Dialog"` come già descritto per gli script embedded degli oggetti grafici. Successivamente sono eseguiti tre metodi dell'oggetto `dlg`: `dlg.Activate()` rende la finestra attiva e la porta in primo piano per far partire l'interazione con l'utente; `dlg.WaitForDialog()` lascia in attesa lo script `LoadDialog.csf` finchè l'utente con conferma i parametri scelti cliccando sul Button `CONTINUE`[14]; `dlg.Close()` chiude la finestra rimuovendola dal layout corrente senza

[14] É stato configurato in modo tale da chiudere il Dialog una volta premuto.

alterarlo. Come ultima operazione è infine impostato a `1` il valore di `%CWF_AITV_syn`, per comunicare a `AITV.py` la conclusione dell'esecuzione di `LoadDialog.csf`.
Tornando a spostare il focus su **AITV.py**, i valori selezionati dall'utente, come è stato descritto, sono salvati nelle relative User Variable e sono a questo punto assegnati per maggiore semplicità computazionale alle variabili locali `feature_x`, `feature_x2`, `feature_y`, `point`, `iterations`, `alpha`. Si tenga a mente che le User Variable sono rese disponibili in ambiente Python da `conc.variables` come **stringhe**, perciò è necessario effettuare il casting con le funzioni `int` e `float` per le variabili `point`, `iterations` e `alpha`. Sono poi estratti il singolo valore risultate scelto dall'utente e il corrispondente valore della variabile di input selezionata, i quali sono assegnati rispettivamente a `y_case` e `x_case`. Il contenuto di `feature_x2` è invece verificato in un *if statement* in modo tale che sia utilizzato per assegnare il valore corretto a `x_case2` solo nel caso in cui sia diverso da una stringa vuota: nella presentazione del CheckBox `SecondInput` è stato riportato infatti che la variabile `%CWF_AITV_in2` è resettata nel caso in cui l'utente rinunci alla selezione di una seconda veriabile di input. Infine sono chiusi i file contenenti le liste dei canali di input e output. Il codice di quest'ultima sezione dello step 1 è di seguito riportato:

```
feature_x = conc.variables["%CWF_AITV_in"]
feature_x2 = conc.variables["%CWF_AITV_in2"]
feature_y = conc.variables["%CWF_AITV_out"]
point=int(conc.variables["%CWF_AITV_SelPoint"])-1
iterations = int(conc.variables["%CWF_AITV_nIters"])
alpha = float(conc.variables["%CWF_AITV_alpha"])
x_case=conc.ds("RESULT1:V_IN'"+feature_x)[point]
if feature_x2 != "":
    x_case2=conc.ds("RESULT1:V_IN'"+feature_x2)[point]
y_case=conc.ds("RESULT1:"+feature_y)[point]
inp_file.close()
out_file.close()
feature_y = feature_y.split("'")[1]
```

Si noti l'ultima istruzione: la stringa `feature_y` è inizialmente nella forma `KEY'DATASET`, ma nel training case i nomi delle variabili di output sono riportati senza l'indicazione della chiave di provenienza, come è stato illustrato nel merito del Parser. Perciò è stato necessario adoperare il metodo `split` delle stringhe di Python, per dividere la stringa nei due elementi a sinistra e a destra dell'apice `'` e scegliere solo il secondo.
Complessivamente questo primo step di `AITV.py` è così implementato:

```
test_type = conc.ds("RESULT1:V_IN'Test_Type")[0]
startPath = conc.variables["%CWF_AdaMoFilePath"]
inp_file=conc.data.select_file("AdaMo_Files" +"\\lib\\"+
test_type+"_Inputs.txt")
out_file=conc.data.select_file("AdaMo_Files" +"\\lib\\"+
test_type+"_Outputs.txt")
inp_file.open("INPUTS")
out_file.open("OUTPUTS")

conc.execute_concerto_script(startPath+"lib\\LoadDialog.csf")
while conc.variables["%CWF_AITV_syn"] == 0:
    pass
conc.variables["%CWF_AITV_syn"]="0"

feature_x = conc.variables["%CWF_AITV_in"]
feature_x2 = conc.variables["%CWF_AITV_in2"]
feature_y = conc.variables["%CWF_AITV_out"]
point=int(conc.variables["%CWF_AITV_SelPoint"])-1
iterations = int(conc.variables["%CWF_AITV_nIters"])
alpha = float(conc.variables["%CWF_AITV_alpha"])
x_case=conc.ds("RESULT1:V_IN'"+feature_x)[point]
if feature_x2 != "":
    x_case2=conc.ds("RESULT1:V_IN'"+feature_x2)[point]
y_case=conc.ds("RESULT1:"+feature_y)[point]
inp_file.close()
out_file.close()
feature_y = feature_y.split("'")[1]
```

2. Lo step successivo consiste nel **caricamento del training set** con la conseguente **selezione delle opportune colonne** in base alle **variabili di input selezionate** dall'utente. Innanzitutto è determinato, in base al tipo di prova, il path del file di training set che viene dunque aperto e caricato come DataFrame di Pandas, in maniera analoga a quanto visto per il Parser:

   ```
   training_set_path=startPath+"lib\\"+test_type+
   "_TrainingSet.txt"
   training_set = pd.read_csv(training_set_path, sep="\t")
   ```

   Dopodiché da `training_set` è estratto un altro DataFrame `data` con il metodo `loc`, che permette di specificare quali righe e colonne prendere dall'oggetto da cui è invocata. Nel nostro caso è necessario mantenere tutte le righe per conservare i dati di tutti i training case, ma sono selezionate solo le colonne corrispondenti alle due variabili di input e quella di output

scelta, che sono successivamente rinominate per maggiore comodità in `'X'`, `'X2'` e `'y'` con il metodo `rename`, a cui è passato come argomento un dictionary contenente le istruzioni per assegnare le nuove **label**:

```
data = training_set.loc[:, [feature_x,feature_x2,feature_y]]
data = data.rename(columns=feature_x:  'X', feature_x2:  'X2',
feature_y:  'y')
```

Questa operazione è eseguita all'interno di un blocco condizionale che verifica l'eventuale presenza di una seconda variabile di input selezionata dall'utente: nel caso in cui sia richiesta una regressione lineare monovariabile, sono replicate le due istruzioni precedenti, ma con il solo input relativo a `feature_x`. L'*if statement* è dunque il seguente:

```
if feature_x2 != "":
    data=training_set.loc[:,[feature_x,feature_x2,feature_y]]
    data=data.rename(columns=feature_x:'X',feature_x2:'X2',
feature_y:  'y')
else:
    data=training_set.loc[:,[feature_x,feature_y]]
    data=data.rename(columns=feature_x:'X',feature_y:'y')
```

I DataFrame di Pandas permettono di accedere alle colonne che lo compongono per mezzo delle relative label e in questo modo le colonne di valori delle due variabili di input sono estratte e assegnate come array NumPy alle variabili `x_orig` e `x_orig2`, che sono successivamente passate come argomenti della **funzione featureNormalize** per la Z-Score Normalization. Analogamente alle operazioni precedenti riguardo la seconda feature di input, `x_orig2` è istanziata solamente nel caso in cui `feature_x2` non sia vuota:

```
x_orig = np.array(data.X)[:,None]
x, mean, std_dev = featureNormalize(x_orig)
if feature_x2 != "":
    x_orig2 = np.array(data.X2)[:,None]
    x2, mean2, std_dev2 = featureNormalize(x_orig2)
```

Per ciascuna feature sono restituite rispettivamente la colonna dei valori di input normalizzati, la media dei valori originali e la deviazione standard (questi ultimi due parametri torneranno utili al passo 4) e sono assegnati alle variabili `x`, `mean` e `std_dev` per la prima feature e a `x2`, `mean2` e `std_dev2` per l'eventuale seconda feature.
Infine è estratta anche la colonna dei valori della variabile di output ed

assegnata come a `y` array NumPy ed è calcolato il numero di campioni nel training set come dimensione di `y`:

```
y = np.array(data.y)
m = len(y)
```

Complessivamente questo secondo step è così implementato:

```
training_set_path=startPath+"lib\\"+test_type+
"_TrainingSet.txt"
training_set = pd.read_csv(training_set_path, sep="\t")
if feature_x2 != "":
    data=training_set.loc[:,[feature_x,feature_x2,feature_y]]
    data=data.rename(columns=feature_x:'X',feature_x2:'X2',
feature_y:'y')
else:
    data=training_set.loc[:,[feature_x,feature_y]]
    data=data.rename(columns=feature_x:'X',feature_y:'y')
x_orig = np.array(data.X)[:,None]
x, mean, std_dev = featureNormalize(x_orig)
if feature_x2 != "":
    x_orig2 = np.array(data.X2)[:,None]
    x2, mean2, std_dev2 = featureNormalize(x_orig2)
y = np.array(data.y)
m = len(y)
```

3. Questo step, cruciale per il processo di validazione, **esegue l'algoritmo di Gradient Descent** che cerca la funzione ipotesi ottimale per la predizione sul training set. Per poter fruire della funzione `gradientDescent` precedentemente definita, è necessario disporre delle opportune variabili `X` e `theta` da passare come argomenti insieme a `y`, `alpha` e `iterations` ottenute invece allo step 2. La variabile `X` dev'essere infatti un array bidimensionale di dimensione m`x`$(n+1)$ composto da una prima colonna di `m` valori uguali a `1`, affiancata dalle colonne (una o due) dei valori di input normalizzati, salvate precedentemente in `x` e - eventualmente - `x2`. Va costruita dunque la colonna di `1` mancante, mediante la funzione `ones_like` di NumPy, che restituisce un'array di tutti valori `1`, con dimensione e forma uguale all'array `x` passato come parametro:
`ones = np.ones_like(x)`
Infine per ottenere `X` è adoperata la funzione `hstack`, che permette di "impilare orizzontalmente" gli array passati come argomenti e cioè unirli ordinatamente come colonne restituendo l'array bidimensionale desiderato. Anche in questo caso un blocco condizionale gestisce il numero variabile di

ingressi adoperati per il gradient descent:

```
if feature_x2 != "":
     X = np.hstack((ones,x,x2))
else:
     X = np.hstack((ones,x))
```

Per quanto riguarda invece `theta`, i parametri θ_k sono inizializzati randomicamente per mezzo della funzione `random.randn(dim)` di NumPy che in questo caso restituisce un array riga di valori casuali della dimensione `dim` passata come argomento. Anche in questo caso è necessario un *if statement* per gestire la presenza della seconda variabile di input e per questo motivo sono semplicemente aggiunte due istruzioni per l'inizializzazione dei θ_k al blocco condizionale riportato precedentemente:

```
if feature_x2 != "":
     X = np.hstack((ones,x,x2))
     theta = np.random.randn(3)
else:
     X = np.hstack((ones,x))
     theta = np.random.randn(2)
```

A questo punto è calcolato e stampato[15] il valore iniziale della Funzione di Costo mediante la funzione **computeCost**:
`computeCost(X, y, theta)`
Finalmente è eseguito l'algoritmo Gradient Descent invocando la funzione **gradientDescent** precedentemente definita, che restituisce i valori finali dei θ_k i quali sono dunque assegnati alla variabile `theta`. Questi valori rappresentano la funzione ipotesi $h_\theta(x)$ migliore che l'algoritmo Gradient Descent, parametrizzato dall'utente, è stato in grado di trovare. La funzione stampa ad ogni iterazione il valore della Funzione di Costo e al termine di essa sono anche stampati i valori finali dei θ_k nella finestra dei messaggi di Concerto:

```
print("Theta found by gradient descent:  ")
print(theta[0],"\n", theta[1])
if feature_x2 != "":
     print(theta[3]
```

Infine è effettuato anche il calcolo degli indici d'errore MAE, MSE e RMSE tramite la funzione **computeErrors** precedentemente definita, che contestualmente li stampa nella finestra dei messaggi di Concerto:
`computeErrors(X, y, theta)`

[15] L'istruzione di print fa parte della definizione della funzione `computeCost`.

Complessivamente questa terza fase è stata così implementata:

```
print("Running Gradient Descent ...\n")
ones = np.ones_like(x)
if feature_x2 != "":
     X = np.hstack((ones,x,x2))
     theta = np.random.randn(3)
else:
     X = np.hstack((ones,x))
     theta = np.random.randn(2)

computeCost(X, y, theta)
theta = gradientDescent(X, y, theta, alpha, iterations)
print("Theta found by Gradient Descent:  ")
print(theta[0],"\n", theta[1])
if feature_x2 != "":
     print(theta[2]
computeErrors(X, y, theta)
```

4. Giunti a questo punto, sono disponibili tutti gli strumenti necessari a **effettuare la predizione sul risultato da validare**, a partire dai relativi valori di input appartenenti alle variabili selezionate. Nello step 1, a seguito dell'immissione dei parametri da parte dell'utente, sono state create le variabili `x_case`, `x_case2` (condizionale) e `y_case`, alle quali sono stati assegnati i valori di input per la funzione ipotesi e quello di output da confrontare con la predizione. Prima di sottoporre `x_case` e `x_case2` alla funzione $h_\theta(x)$, è necessario normalizzare anche il loro valore con il metodo della Z-Score Normalization, ma stavolta non è necessario ricorrere alla funzione **featureNormalize** come invece è stato fatto allo step 2 per tutti gli elementi delle colonne degli input, estratte dal training set. La funzione `featureNormalize` è infatti concepita per gestire un intero array NumPy, di cui calcola **media** e **deviazione standard**, ma in questo caso per effettuare la normalizzazione di `x_case` e `x_case2` è sufficiente disporre proprio di questi valori, precedentemente calcolati dalla funzione e assegnati alle variabili `mean`, `std_dev`, `mean2` e `std_dev2` in corrispondenza della normalizzazione[16]. Di conseguenza per scalare per esempio `x_case` è sufficiente che gli sia sottratto `mean` e che il risultato sia poi diviso per `std_mean`, ottenendo così `x_case_norm`. Naturalmente è necessario anche in questo caso un blocco condizionale per l'eventuale presenza di una seconda variabile di input per la predizione:

```
x_case_norm=(x_case-mean)/std_dev
```

[16] Questa operazione è effettuata allo step 2.

```
if feature_x2 != "":
    x_case_norm2=(x_case2-mean2)/std_dev2
```

Nel caso di due variabili di input, questi valori sono utilizzati per creare un array riga di tre elementi, costituito da un `1` seguito appunto dai valori di `x_case_norm` e `x_case_norm2`, costituendo in questo modo i coefficienti dell'equazione di $h_\theta(x)$, che vanno dunque semplicemente moltiplicati per le componenti dell'array `theta` per poi sommare infine i due risultati. Per effettuare questo *prodotto scalare* è sufficiente ricorrere alla funzione `np.dot` per ottenere quindi il valore della predizione assegnato a `prediction`. Diversamente, per una sola variabile di input la procedura sarà analoga, ma senza ricorrere a `x_case_norm2`:

```
if feature_x2 != "":
    prediction = np.dot([1, x_case_norm, x_case_norm2],theta)
else:
    prediction = np.dot([1, x_case_norm],theta)
```

Il valore predetto è stampato nella finestra dei messaggi di Concerto e da esso è ottenuta la differenza `diff` rispetto al risultato realmente calcolato `y_case` e analogamente è restituita anche la differenza percentuale `diff_pc` rispetto a `y_case`. Entrambi questi parametri, fondamentali per la valutazione della predizione e dunque per la validazione, sono mostrati nella finestra dei messaggi.
Si riporta dunque il codice complessivo di questo quarto step:

```
x_case_norm=(x_case-mean)/std_dev
if feature_x2 != "":
    x_case_norm2=(x_case2-mean2)/std_dev2

if feature_x2 != "":
    prediction = np.dot([1, x_case_norm, x_case_norm2],theta)
else:
    prediction = np.dot([1, x_case_norm],theta)
if feature_x2 != "":
    print("For",feature_x,"=",x_case,"and",feature_x2,"=",
x_case2,", the Predicted value of",feature_y,"is:",prediction)
else:
    print("For",feature_x,"=",x_case,", Predicted value of",
feature_y, "is:", prediction)
print("The value calculated from Concerto Layout is:", y_case)
diff = y_case - prediction
diff_pc = (y_case - prediction) * 100 / prediction
```

```
print("The difference between calculated and predicted values
is", diff, "wich corresponds to the", diff_pc, "%")
```

5. In questo ultimo step è necessario associare ai risultati della predizione, già mostrati all'utente attraverso la finestra dei messaggi di Concerto, una **rappresentazione grafica dell'andamento della funzione ipotesi rispetto ai dati del training set**. Per questo scopo risulta opportuno sfruttare nuovamente le strutture che Concerto mette a disposizione, perciò è stata realizzata una Window contenente un **Diagram** che mostra un piano cartesiano di cui l'asse delle ascisse è associato ai dati della prima variabile di input tratti dal training set, mentre sull'asse delle ordinate ci sono i valori di output registrati nel training set e quelli restituiti dalla funzione di predizione ottenuta con il Gradient Descent. Questa finestra è stata salvata nella solita directory `lib` con il nome **AITV_Diagram.cdi** ed è richiamata e caricata mediante un altro script ausiliario in linguaggio Concerto, analogamente a quanto visto al punto 1 per la finestra Dialog.
   Per poter tuttavia mostrare sul Diagram i dati del training set e della predizione, è necessario che siano contenuti in un **data file** caricato in Concerto e a cui sia associato dunque un Alias. Allo stato attuale disponiamo del DataFrame `data` ottenuto dal training set che, nel caso in cui l'utente abbia scelto due variabili di input, è composto dalle colonne `X` e `X2`, contenenti i valori delle due feature di input, e dalla colonna `y`, contenente i valori del dataset risultato. É necessario dunque innanzitutto rinominare queste colonne con il metodo `data.rename` (già adoperato nello step 2) per assegnare a `X` e `X2` i nomi delle variabili di input selezionate dall'utente, assegnati a `feature_x` e `feature_x2`, mentre alla colonna `y` è associato il nome della variabile relativa al risultato da validare assegnato a `feature_y`[17]. Questa operazione è effettuata passando a `data.rename` un apposito dictionary che indica le regole di traduzione delle label, assegnando il nuovo DataFrame a `data_pred`. É riportato dunque di seguito il blocco condizionale per la gestione di una o due variabili di input:

   ```
   if feature_x2 != "":
       data_pred = data.rename(columns="X":  feature_x,
   "X2":  feature_x2, "y":  feature_y)
   else:
       data_pred = data.rename(columns="X":  feature_x,
   "y":  feature_y)
   ```

   Successivamente è necessario associare a `data_pred` una terza colonna associata ai valori predetti dall'ipotesi $h_\theta(x)$ ottenuta con il Gradient Descent,

[17] Si ricordi che la variabile `feature_y` è stata precedentemente modificata in modo tale che non riportasse anche il nome della chiave.

mediante l'apposita sintassi. La colonna di valori è ottenuta mediante `np.dot` che effettua il prodotto righe per colonne della matrice `X`, dotata della colonna dei coefficienti 1 di bias, per l'array `theta`:
`data_pred["prediction"]=np.dot(X,np.transpose(theta))`
Infine il DataFrame `data_pred` è trascritto su un altro file testuale all'interno della cartella `lib` associato al tipo di prova eseguita (`<nomeprova>_TrainingSet_Pred.txt`), che è sovrascritto ad ogni nuova predizione e si attende il completamento dell'operazione con un'attesa di un secondo (`time.sleep(1)`):

```
path=startPath+"lib\\"+test_type+"_TrainingSet_pred.txt"
data_pred.to_csv(path, sep="\t", index=False)
time.sleep(1)
```

A questo punto è possibile aprire il file testuale così generato attraverso il DataSource `AdaMo_Files`, assegnando come Alias `TRAINPRED`:

```
ts_file=conc.data.select_file("AdaMo_Files" +"\\lib\\"+
test_type+"_TrainingSet_temp.txt")
ts_file.open("TRAINPRED")
```

In conclusione è lanciato lo script `LoadAITVDiagram.csf`, anch'esso posizionato nella cartella `lib`, che si occupa di caricare e gestire la finestra Diagram.
`conc.execute_concerto_script(startPath+`
`"lib\\LoadAITVDiagram.csf")`
Complessivamente l'implementazione di quest'ultimo step di `AITV.py` è la seguente:

```
ts_file=conc.data.select_file("TRAINPRED")
if ts_file is not None:
    ts_file.close()
if feature_x2 != "":
    data_pred = data.rename(columns="X":  feature_x,
"X2":  feature_x2, "y":  feature_y)
else:
    data_pred = data.rename(columns="X":  feature_x,
"y":  feature_y)
data_pred["prediction"]=np.dot(X,np.transpose(theta))
path=startPath+"lib\\"+
test_type+"_TrainingSet_Pred.txt"
data_pred.to_csv(path, sep="\t", index=False)
time.sleep(1)
```

```
ts_file=conc.data.select_file("AdaMo_Files" +"\\lib\\"+
test_type+"_TrainingSet_Pred.txt")
ts_file.open("TRAINPRED")
conc.execute_concerto_script(startPath+"lib
LoadAITVDiagram.csf")
```

Si noti che le due righe aggiunte in cima a questa porzione dello script hanno lo scopo di chiudere il file con Alias `TRAINPRED` nel caso in cui sia stato caricato nel corso di una precedente esecuzione di AITV, per garantire una gestione pulita del layout. Per fare questo si prova a istanziare l'oggetto `ts_file` mediante il metodo `conc.data.select_file`, per poi verificarne la validità nell'if statement che segue: solo nel caso in cui `ts_file` non sia un *None Object*, sarà eseguito il metodo di chiusura del file `TRAINPRED`. Poniamo infine l'attenzione sullo script ausiliario **LoadAITVDiagram.csf** e sulla finestra **AITV_Diagram.cdi** che mostra la relazione tra input e output reale e predetto. Lo script consiste semplicemente nelle seguenti righe:

```
wnd=SelWin("AITV Prediction Diagram")
wnd.Close()
startPath = %CWF_AdaMoFilePath
startPath = StrReplaceAll(startPath,"\\","\")
LoadWindow(startPath+"lib\AITV_Diagram.cdi")
wnd=SelWin("AITV Prediction Diagram")
wnd.Activate()
diag=wnd.SelObj("AITVDiagram")
ydiag = diag.SelObj()[1]
feat_y = StrTokenize(%CWF_AITV_out,"'").y[2]
ydiag.yDSName = "TRAINPRED:D'"+feat_y
diag.xDSName = "TRAINPRED:D'"+%CWF_AITV_in
```

Ancora una volta si utilizza la User Variable `%CWF_AdaMoFilePath` per ottenere il path del file associato alla finestra, rimpiazzando come sempre i doppi \\ con un solo \ ed è successivamente caricato il Diagram con la funzione `LoadWindow`. Infine è creato l'oggetto `wnd` relativo alla finestra con la funzione `SelWin` adoperando il nome identificativo `"AITV Prediction Diagram"` scelto per il Diagram, il quale è infine mostrato all'utente con il metodo `wnd.Activate()`. Si noti che le prime due righe di questo script servono proprio a chiudere la stessa finestra "AITV Prediction Diagram" nel caso sia stata aperta nel corso di una precedente esecuzione di AITV, esattamente come è stato fatto per il file con Alias `TRAINPRED`. Infine le ultime cinque righe di questo script associano i corretti dataset del file `TRAINPRED` agli assi delle ascisse e delle ordinate: mentre `TRAINPRED:D'prediction`

sarà sempre presente nei file `<NomeProva>_TrainingSet_Pred.txt`, i dataset relativi alla feature di input e a quella di output del training set, seguiranno la nomenclatura dei canali di provenienza, che naturalmente non può essere associata staticamente al Diagram. Per questo motivo è necessario istanziare l'oggetto `diag` della classe `Object` relativo all'oggetto grafico del Diagram ed il sotto-oggetto `ydiag` relativo alla scala di sinistra (fig 3.20), utilizzando in questo caso l'omonimo metodo `SelObj`, associato tuttavia alla classe `Object` a differenza di quanto visto precedentemente. Utilizzando dunque `diag.SelObj()[1]` si ottiene il primo dei sotto-oggetti di `diag`, che viene dunque assegnato a `ydiag` e in questo modo è possibile assegnare il dataset desiderato all'asse sinistro delle ordinate, mediante l'attributo `ydiag.yDSName`, secondo il classico costrutto. É dunque necessario estrapolare dalla User Viariable `%CWF_AITV_out` solamente il nome del dataset risultato sottraendo dunque dalla stringa che essa contiene il nome della chiave: analogamente a quanto visto per il metodo `split` in ambiente Python, è utilizzata la funzione `StrTokenize(str,sep)` della libreria **String**, che permette di dividere la stringa `str` in più elementi delimitati dal separatore `sep` (nel nostro caso il singolo apice `'`), restituendo una variabile-dataset, dalla quale è ottenuta la substring desiderata con il metodo `y[index]`. La variabile `feat_y`, alla quale è dunque assegnata la stringa a destra dell'apice, nel costrutto parziale `KEY'DATASET` che contraddistingue `%CWF_AITV_out`, è infine adoperata per la definizione del dataset associato alla feature di output di `TRAINPRED`, da assegnare all'attributo `ydiag.yDSName`. Per associare invece la corretta variabile alle ascisse, è sufficiente assegnare la stringa `"TRAINPRED:D'"+%CWF_AITV_in` all'attributo `diag.xDSName`, associato in questo caso direttamente all'oggetto Diagram `diag`.

A questo punto l'utente può consultare questi dataset con tutte le metodologie di esplorazione che Concerto mette a disposizione, confrontando i dati della predizione con i risultati del Gradient Descent precedentemente stampati nella finestra dei messaggi. In fig. 3.20 è mostrato un Diagram di esempio con dati fittizi sui quali è stato eseguito l'algoritmo di validazione ricorrendo ad una sola variabile di input. Nel caso in cui sia stato selezionato anche un secondo ingresso per la predizione, è lasciata all'utente la libertà di manipolare questa finestra per associare l'altra feature all'asse delle ascisse o valutare diversamente la funzione di predizione ad esempio con grafici 3D **scatter** o di tipo **surface**.

In conclusione è finalmente possibile per l'utente adoperare le informazioni fornite da AITV per giudicare i risultati calcolati dalla post-elaborazione, determinando in questo modo se tali valori possano essere considerati affidabili e dunque validati. Qualora invece non sia soddisfatto della predizione effettuata da AITV, è possibile lanciare nuovamente l'algoritmo parametrizzandolo in modo più opportuno, per personalizzare al meglio il processo di validazione.

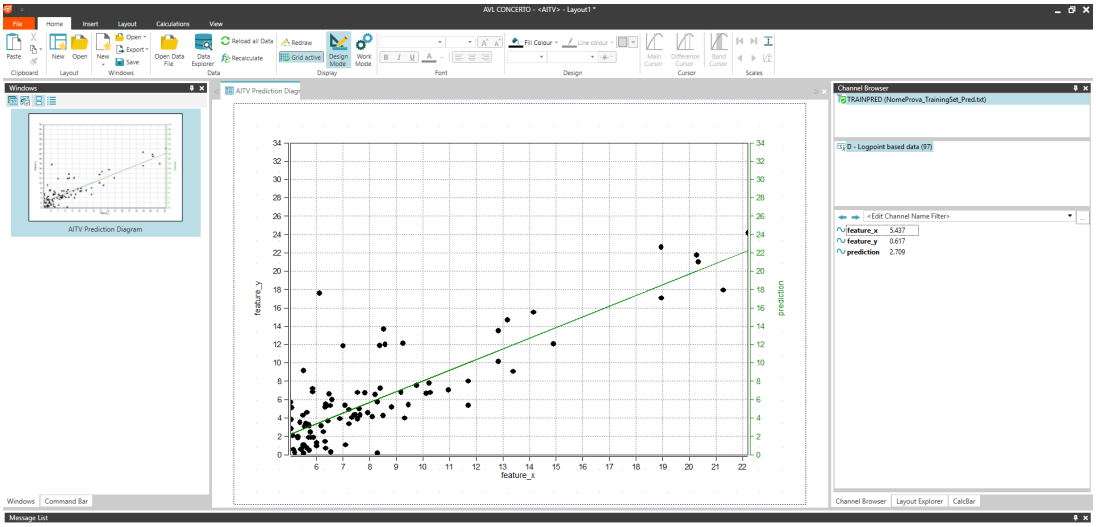

Figura 3.20: Diagram per la valutazione della predizione di AITV.

Capitolo 4

Caso di studio: Lambda Step

L'esposizione di un caso di studio è il metodo migliore per fornire un esempio concreto del funzionamento della struttura descritta nel paragrafo precedente. É stata scelta a tale scopo una tipologia di test frequentemente richiesta, che permette di valutare un parametro fondamentale per descrivere lo stato attuale di un catalizzatore. Il sistema **aftertreatment** costituisce infatti un componente di grande interesse per le sperimentazioni condotte in sala prova, dal momento che la sua qualità e lo stato in cui versa condizionano in modo determinante il quantitativo di inquinanti emessi dallo scarico. Lo sviluppo di un motore e il raggiungimento di una *calibrazione* ottimale della ECU, passano necessariamente dalla scelta del sistema aftertreatment più adatto a garantire che, per tutta la sua vita utile, le emissioni rilasciate in ambiente siano al di sotto dei limiti della normativa vigente. Lo stato di un catalizzatore rappresenta dunque un parametro fondamentale che bisogna valutare attraverso prove apposite, come quella del **Lambda Step**, che serve a calcolare il valore di **Oxygen Storage Capacity** (OSC), cioè la capacità del catalizzatore di immagazzinare ossigeno, il quale fornisce un'indicazione dell'invecchiamento del componente.
Tra i diversi sensori di cui è dotata la linea di scarico, uno dei più importanti è la **sonda lambda**, un misuratore di concentrazione di ossigeno che permette di determinare se la miscela adoperata dal motore sia *ricca*[1] ($\lambda < 1$) - cioè c'è un eccesso di combustibile rispetto all'aria comburente - o se al contrario sia *povera*[2] ($\lambda > 1$). I propulsori a metano sviluppati nel nostro centro sono infatti motori ad accensione comandata, caratterizzati da un funzionamento ideale in corrispondenza di una miscela *stechiometrica* ($\lambda = 1$) e oscillazioni nel rapporto *aria su combustibile* (AFR, Air to Fuel Ratio) - e dunque sul valore di lambda - comportano una variazione sensibile di prestazioni, consumi ed emissioni. La ECU gestisce un apposito **controllore del titolo** che permette di *arricchire* o *smagrire* la miscela nei transitori che si verificano in condizioni di normale utiliz-

[1] Si dice anche "grassa".
[2] Si dice anche "magra".

zo del motore, utilizzando come feedback il valore misurato dalla sonda lambda, che tipicamente indica solo se una condizione di "ricco" o "magro", senza quantificare un valore effettivo. Questo tipo di sensore è chiamato *lambda switch*, ma è utilizzata non di rado anche la *sonda lambda lineare*, un trasduttore più sofisticato in grado di indicare appunto il valore effettivo di lambda.
Un metodo che si può adoperare per il calcolo dell'Oxygen Storage, consiste nell'analisi di un evento transitorio da "ricco" a "magro", che vede dunque la lambda passare da un valore minore di 1 a un valore maggiore di 1. La quantità d'ossigeno in eccesso dovuta al repentino smagrimento della miscela, viene immagazzinata per alcuni istanti nel catalizzatore, per poi essere rilasciata anche a valle. La massa d'ossigeno accumulata in questo intervallo di *breakthrough* costituisce l'Oxygen Storage di cui il catalizzatore in prova è capace [6], che ha l'effetto benefico di limitare le emissioni in condizioni di perturbazione del rapporto aria-combustibile: in questo modo durante una fase di combustione magra l'ossigeno trattenuto limita le reazioni di produzione degli NOx a valle del catalizzatore, mentre nelle fasi di combustione ricca facilita l'ossidazione di CO, CH4 e HC.
Nel prossimo paragrafo 4.1 sarà descritta la procedura di test del Lambda Step, teorizzata per replicare questi fenomeni transitori in modo ripetibile e osservabile, in modo tale da calcolare il valore di Oxygen Storage su diversi punti operativi. Il paragrago 4.2 introduce brevevemente alcune delle tecniche utilizzate in fase di post-elaborazione per il calcolo dell'OSC, mentre nell'ultimo paragrafo 4.3 sarà riportata la procedura di validazione mediante AITV, secondo le tecniche illustrate nella sezione 3.4.2.

4.1 Algoritmo di test

La procedura di test relativa al Lambda Step è teorizzata e trascritta su una **Norma Interna FPT**, un tipo di documento utilizzato per descrivere le attività di ricerca in tutti i loro aspetti, dalle specifiche per l'algoritmo di test alla definizione dei controlli e calcoli per determinare correttezza ed esito delle prove. Ponendo l'attenzione in particolare sull'algoritmo di test, è prescritto che si ripeta un'**interazione elementare** su **quindici punti operativi**, determinati da tre Velocità e cinque valori di carico, per ciascuno dei quali è dunque effettuata una serie di transizioni da "ricco" a "magro" - e viceversa - permanendo in ciascuno stato per un dato tempo. La procedura si snoda dunque in due loop annidati, tramite i quali è fissato un valore di Velocità impostata e per ciascuno dei cinque valori di carico previsti, che si traducono dunque in cinque corrispondenti valori di Coppia impostata, è effettuata l'interazione elementare.
L'automazione di sala prova ha dunque il compito di fare in modo che il motore permanga nel punto operativo *demanded* secondo le tecniche descritte nel paragrafo 1.2.1, controllando Velocità e Coppia del motore in modo tale che

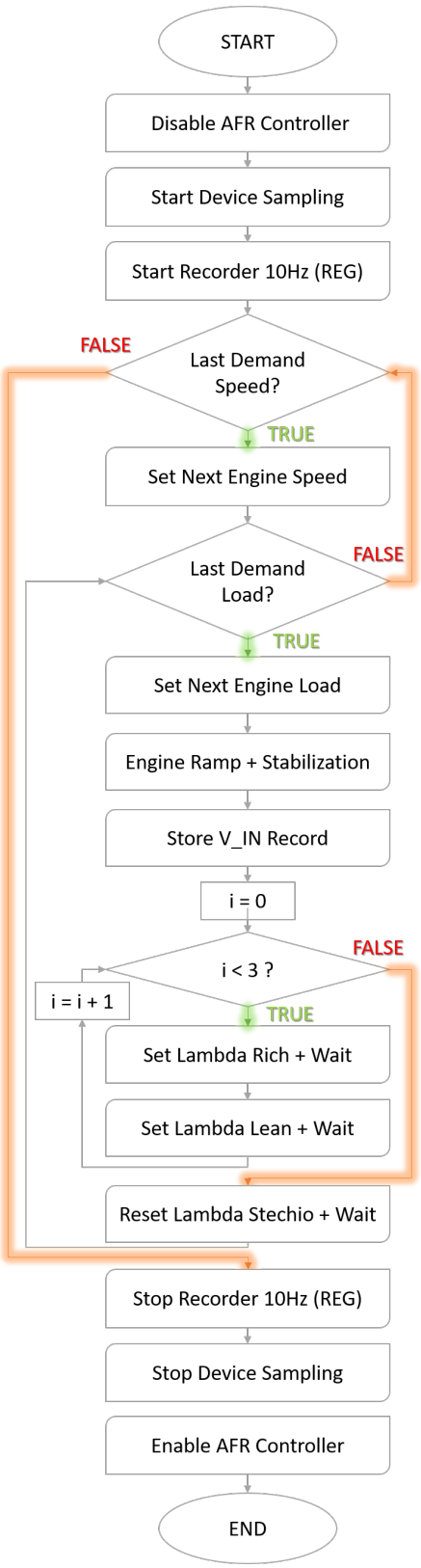

Figura 4.1: Algoritmo di test del Lambda Step.

si assestino intorno ai valori impostati per un dato tempo di stabilizzazione. Per effettuare invece le transizioni che portano alternatamente da un eccesso d'aria a un eccesso di combustibile nella miscela, è necessario **trasmettere all'application system**, che gestisce la ECU, **il valore di lambda obiettivo**, ricordando che $\lambda < 1$ implica una combustione ricca e al contrario $\lambda > 1$ implica una combustione magra. Questo significa che sarà inviato un messaggio con protocollo ASAP3 dal sistema d'automazione a quello di controllo ECU, imponendo istantaneamente uno dei due valori desiderati di lambda λ_{rich} e λ_{lean}, riportati nella Norma, eseguendo rispettivamente uno **step** di lambda obiettivo ricco e magro. Sono così effettuate tre transizioni con una sequenza $\lambda_{rich} \rightarrow \lambda_{lean} \rightarrow \lambda_{rich} \rightarrow \lambda_{lean} \rightarrow \lambda_{rich} \rightarrow \lambda_{lean}$, al termine delle quali è ripristinato il corretto valore di lambda obiettivo stechiometrico $\lambda = 1$, seguito da un tempo di stabilizzazione per ripristinare la condizione iniziale.
Analizziamo dunque in modo sintetico l'algoritmo del Lambda Step, rappresentato dal flowchart riportato in figura 4.1.

- Innanzitutto è necessaria una fase di inizializzazione che consiste nella **disabilitazione del controllore di titolo della ECU**, centrando manualmente il valore stechiometrico. In questo modo è disattivato il controllo della dosatura ed è possibile imporre in *open loop* un rapporto aria-combustibile, di cui si ha riscontro mediante la misura di una sonda lambda lineare.

- É necessario porre in stato di *sampling* tutti gli strumenti di misura disponibili, tra quelli richiesti per questo tipo di prova. In particolare è mandatorio l'utilizzo di un **banco analisi emissioni** con doppio prelievo simultaneo (a monte e a valle del catalizzatore), oltre che due sonde lambda lineari (anche in questo caso una a monte e una a valle), che sono invece costantemente in misura.

- É avviata un'**acquisizione continua** con frequenza a 10Hz (cioè un campione ogni 100 millisecondi), riferita alla chiave `REG`, nella quale sono convogliati tutti i dati necessari alla post-elaborazione.

- É fissato il prossimo valore di Velocità dei tre prescritti dalla norma e successivamente il prossimo valore di carico dei cinque prescritti dalla norma ed è calcolato il relativo valore di Coppia. Il motore è a questo punto **controllato nel punto operativo**, definito da tali valori di demand di Velocità e Coppia, con una rampa a gradiente costante ed è posto in stabilizzazione per un certo tempo.

- Viene eseguita una **misura stazionaria** che produce un **nuovo punto nella chiave** `V_IN` per la validazione con AITV, nel quale sono contenuti i valori mediati su trenta secondi delle seguenti quantity:

  - `T_ATS` associata alla Temperatura dell'ATS, misurata da una termoresistenza;
  - `qm_air` associata alla portata massica dell'aria in ingresso al motore, misurata da un apposito strumento;
  - `Conc_O2_EngineOut` associata alla concentrazione di ossigeno nel gas in uscita dal collettore di scarico e a monte dell'ATS, misurato dal banco analisi emissioni.

- É effettuata l'iterazione elementare che prevede una serie di tre transizioni ricco-magro intervallate dall'attesa di un certo tempo, seguita dal ripristino del valore di lambda obiettivo stechiometrico pari a 1.

- É infine interrotto il campionamento degli strumenti adoperati, fermata l'acquisizione continua su `REG` e ripristinato il controllore del titolo.

Questa procedura, semplificata in questa illustrazione perché contenga solo gli elementi essenziali, è implementata nei due sistemi d'automazione **AdaMo** e **PUMA**, secondo le rispettive tecniche di programmazione dei test introdotte nei capitoli precedenti. Nel fare questo è naturalmente riprodotto rigidamente l'algoritmo descritto e sono osservati tutti i dettami della Norma, in modo tale da produrre un file di risultati ugualmente strutturato e neutrale rispetto all'automazione di provenienza. Contestualmente è comunque assicurata una certa parametrizzazione della procedura di test mediante la preliminare richiesta all'utente di immissione di valori di input, relativi per esempio alla durata delle varie attese o ai valori di lambda obiettivo ricco e magro.
Infine per quanto riguarda le sale AdaMo, la cartella di risultati contenente i file prodotti da questa prova è caricata nell'apposita area di archiviazione sulla risorsa condivisa nella rete aziendale, in modo tale che sia possibile convertirla in formato ASAM mediante il **FileConverter**, che posizionerà il file ATF restituito nella directory target del **Datasource relativo alla sala prova di provenienza**.
Nell'ambito della descrizione di questo progetto, non risulta di particolare interesse riportare ulteriori dettagli relativi all'implementazione del Lambda Step sui sistemi d'automazione AdaMo e PUMA, ma è sufficiente la definizione dell'algoritmo precedentemente riportata, che fornisce le basi per la comprensione dei prossimi paragrafi.

4.2 Elaborazione dei risultati

I file di risultati relativi alle prove di Lambda Step, prodotti dall'automazione di sala prova e creati dal FileConverter nel caso specifico delle sale AdaMo, sono resi disponibili dai Datasource relativi alle sei sale prova che costituiscono il Database

centralizzato descritto nel paragrafo 3.2.2. Dal Data Explorer di Concerto è dunque possibile accedere a questi dati e caricarli in un layout di post-elaborazione opportunamente realizzato, che è in grado di calcolare il valore di Oxygen Storage per ciascuna iterazione elementare del Lambda Step, corrispondente ad un determinato punto operativo. Nel paragrafo precedente è stato specificato inoltre che per ognuna di queste serie di gradini di lambda è acquisito un record della chiave `V_IN`, che contiene dunque i parametri preliminari della singola iterazione che possono determinarne l'esito in termini di Oxygen Storage. Si intuisce dunque che per ciascun punto della chiave `V_IN` sarà calcolato un valore risultante dalla post-elaborazione, garantendo così una corretta struttura dei dati per la fase di validazione con AITV.
Il calcolo vero e proprio del valore di **Oxygen Storage Capacity**, in una transizione da ricco a magro, è effettuato mediante la seguente formula:

$$OSC_{rich2lean}[mg] = \frac{10^6}{3600} \cdot \chi_{ossigeno} \cdot \int_{t_0}^{t_1} qm_{air} \cdot (1 - \frac{1}{\lambda})\, dt$$

dove

- $\chi_{ossigeno} = 0.232$ è la frazione molare di ossigeno rispetto all'aria;
- il tempo t_0 corrisponde all'istante in cui il valore misurato dalla sonda lambda lineare a **monte** dell'ATS supera il valore di 1;
- il tempo t_1 corrisponde all'istante in cui il valore misurato dalla sonda lambda lineare a **valle** dell'ATS supera il valore di 1;
- qm_{air} è la portata massica (in kg/h) dell'aria in ingresso al motore, chiamata come la quantity adoperata nell'automazione di sala prova;
- λ è il valore misurato dalla sonda lambda lineare a monte dell'ATS.

Nella realizzazione delle finestre che compongono il layout e delle formule che elaborano i risultati fino a restituire un valore di Oxygen Storage, è dunque necessario individuare l'intervallo temporale che intercorre tra t_0 e t_1 in cui effettuare il calcolo dell'integrale.
Naturalmente questo metodo rappresenta una delle tecniche che è possibile adoperare per calcolare l'OSC ed è possibile ricorrere ad altre metodologie nel caso in cui non siano disponibili le grandezze necessarie al calcolo precedentemente descritto. Se ad esempio non si disponesse di sonde lambda lineari, si potrebbe utilizzare la concentrazione di ossigeno a monte e a valle dell'ATS, misurata dai banchi analisi, adoperando una formula di calcolo leggermente diversa:

$$OSC_{rich2lean}[mg] = \frac{10^6}{3600} \cdot \chi_{ossigeno} \cdot qm_{air} \cdot \int_{t_0}^{t_1} (Conc_{O_{2_{monte}}} - Conc_{O_{2_{valle}}})\, dt$$

dove

- $Conc_{O_{2monte}}$ è la concentrazione (percentuale in volume) di ossigeno a monte dell'ATS misurata dal banco analisi;
- $Conc_{O_{2valle}}$ è la concentrazione (percentuale in volume) di ossigeno a valle dell'ATS misurata dal banco analisi;
- il tempo t_0 corrisponde all'istante in cui $Conc_{O_{2monte}}$ supera il valore di 700ppm;
- il tempo t_1 corrisponde all'istante in cui $Conc_{O_{2valle}}$ supera il valore di 700ppm.

Nel cosiderare la misura della concentrazione di ossigeno a monte e valle del catalizzatore è necessario tenere in conto i ritardi di campionamento tipici dell'analizzatore all'interno del banco analisi emissioni, che comportano uno *shift* significativo nelle tracce delle relative quantity riportate nel recorder *REG*.
Ad ogni modo, il risultato dell'elaborazione è la formula `OSC` che corrisponde ad un dataset contenente un punto per ogni iterazione di Lambda Step effettuata, ciascuna delle quali può essere oggetto di validazione mediante l'algoritmo di predizione di AITV. Anche in questo caso sono omessi ulteriori dettagli relativi alla realizzazione del layout di Concerto post-elaborazione, che non risultano utili nell'ottica della descrizione di questo progetto. Al contrario nel prossimo paragrafo sarà approfondita la procedura di validazione di un risultato ottenuto da un Lambda Step effettuato in sala prova, mediante gli strumenti descritti nel capitolo 3.

4.3 Validazione con AITV

I dettagli forniti nei paragrafi 4.1 e 4.2 aiutano a comprendere le caratteristiche del caso di studio in esame, dal punto di vista della struttura dell'algoritmo di test e dei calcoli necessari ad ottenere i valori di Oxygen Storage. Queste premesse costituiscono la base fondamentale per l'analisi di un esempio vero e proprio di esecuzione di un test di tipo Lambda Step, con conseguente elaborazione e validazione dei risultati. Si intuisce che i valori numerici legati ai parametri preliminari di temperatura dell'ATS, portata d'aria in ingresso al motore e concentrazione d'ossigeno a monte del catalizzatore, così come i valori di Oxygen Storage risultanti, costituiscono dati sensibili relativi a prodotti della mia azienda. Per questo motivo sarà operata una normalizzazione di tutti questi dati, dividendo ogni valore di ciascuna variabile per il massimo[3] ottenuto relativamente alla grandezza di riferimento.

[3] Ci si riferisce al valore massimo incontrato in questo caso di studio, cioè il valore più grande tra quelli acquisiti e calcolati in *LambdaStepTest* e contenuti nel training set.

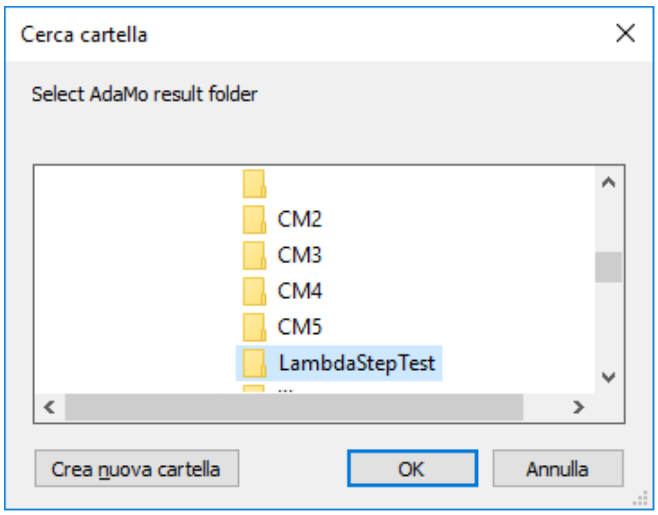

Figura 4.2: Selezione della folder di risultati *LambdaStepTest* per il FileConverter.

Si considera quindi il caso di una prova di Lambda Step eseguita nella sala prova CM4, coordinata dal sistema d'automazione AdaMo. Come è stato descritto nei capitoli precedenti, le prove implementate su questo software generano una folder come *test result*, che contiene i diversi file di testo associati alle chiavi di memorizzazione. In particolare a seguito del Lambda Step eseguito in questa sala, è stata generata una directory chiamata **LambdaStepTest**, che contiene i seguenti file:

- *V_IN.ASCII*, associato alla chiave logpoint-based `V_IN`, contenente i dati preliminari di ciascuna iterazione elementare del Lambda Step, da associare ai relativi risultati calcolati in post-elaborazione;
- *Eng.ASCII*, associato alla chiave logpoint-based `Eng`, è il risultato di un'istruzione di `SNAPLOG` che memorizza i principali parametri caratteristici della Unit Under Test, cioè il motore in prova;
- *MD.ASCII*, associato alla chiave logpoint-based `MD`, è il risultato di un'istruzione di `SNAPLOG` che memorizza alcune quantity relative a grandezze che caratterizzano lo stato della sala prova all'inizio del test (pressione atmosferica, temperatura dell'aria, etc.);
- *REG.ASCII*, associato alla chiave time-based `REG`, è l'acquisizione continua totale[4] della prova, nella quale sono memorizzate le tracce delle quantity necessarie al calcolo dell'Oxygen Storage.

Questa folder è esportata dal filesystem del PC di sala prova a quello della risorsa d'archiviazione condivisa nella rete aziendale ed è posizionata nell'area associata

[4]La chiave `REG` sarà dunque composta di un solo measurement ID.

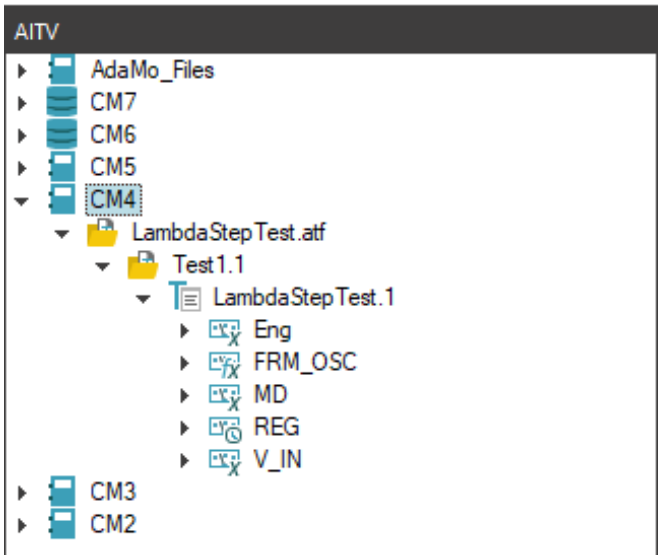

Figura 4.3: Inserimento automatico nel Datasource `CM4` del file *LambdaStepTest.ATF* generato dal FileConverter.

al Datasource `AdaMo_Files`, in modo tale che sia disponibile per il **FileConverter** per la generazione del file ATF. É dunque eseguito il Job Concerto relativo al FileConverter (dal menu mostrato in fig. 3.9), selezionando la cartella di nostro interesse come in fig. 4.2 ed è ottenuto il file **LambdaStepTest.ATF**, posizionato automaticamente nel Datasource `CM4` associato all'omonima sala prova di provenienza. É bene ricordare che il corretto funzionamento dell'algoritmo di conversione si basa anche sul corretto assegnamento delle stringhe che identificano la sala prova di provenienza e la tipologia di test effettuato, rispettivamente alle quantity `TestBench` e `Test_Type`. Nel caso della nostra prova **LambdaStepTest** sarà assegnato, per ogni punto di `V_IN`, a `TestBench` la stringa "CM4" e a `Test_Type` la stringa "LStep". In questo modo è possibile identificare il file di mapping delle chiavi logpoint-based e time-based che caratterizzano la prova di Lambda Step *LStep_Mapping.txt*, indispensabile per la conversione, e la directory in cui creare il file ATF per permetterne l'accesso tramite il Datasource della sala prova CM4.
La fig. 4.3 mostra il menu ad albero relativo al Data Environment **AITV**, in cui si può navigare grazie al Data Explorer di Concerto, che presenta il file **LambdaStepTest.ATF** creato dal FileConverter. Si può notare che sono presenti tutte le chiavi corrispondenti ai file testuali di partenza contenuti nella directory di risultati, restituita dal sistema AdaMo di CM4 a seguito dell'esecuzione della prova di Lambda Step, ma è presente inoltre la chiave di formule `FRM_OSC` associata generalmente al Datasource di ogni sala prova, per rendere

RESULT1:	V_IN'			FRM_OSC'
Logpt	T_ATS	qm_air	Conc_O2_EngineOut	OSC
-	°C	kg/h	ppm	mg
1	0.883754	0.947867	0.749664	0.475884
2	0.864146	0.817874	0.717362	0.486602
3	0.840336	0.668923	0.676985	0.486602
4	0.788515	0.494584	0.644684	0.534532
5	0.760504	0.346649	0.643338	0.62701
6	0.861345	0.920785	0.573351	0.46881
7	0.851541	0.744753	0.664872	0.470311
8	0.812325	0.590386	0.592194	0.486602
9	0.752101	0.446852	0.640646	0.49732
10	0.679272	0.297901	0.648721	0.482315
11	0.717087	0.514557	0.604307	0.398714
12	0.693277	0.417062	0.660834	0.407288
13	0.661064	0.330399	0.764468	0.390139
14	0.619048	0.238321	0.869448	0.300107
15	0.890756	0.912661	1	0.131083

Tabella 4.1: Associazione tra i valori d'ingresso di V_IN e il valore risultante della formula OSC

disponibili i dataset di calcolo[5] basati sui canali d'acquisizione contenuti nei file. All'interno di questa chiave sarà disponibile il dataset formula `OSC`, dotato dello stesso numero di punti di ciascun dataset acquisito della chiave `V_IN`, che presenterà il valore calcolato di Oxygen Storage per ciascuna iterazione elementare del Lambda Step, a seguito del corretto impiego del layout di post-processing, secondo le tecniche descritte nel paragrafo precedente. Il file **LambdaStepTest.ATF** può dunque essere caricato mediante il Datasource `CM4` con l'automatica associazione dell'Alias `RESULT1`, perché sia applicato ad esso il layout Concerto in grado di assegnare al dataset `RESULT1:FRM_OSC'OSC` i valori calcolati dalla relativa formula, a partire dalle tracce acquisite nella chiave `RESULT1:REG`. Nella tabella 4.1 è riportata l'associazione, per ogni punto del LambdaStep, tra i valori di input delle variabili `RESULT1:V_IN'T_ATS`, `RESULT1:V_IN'qm_air` e `RESULT1:V_IN'Conc_O2_EngineOut` memorizzate in `V_IN` e il corrispondente valore di Oxygen Storage calcolato dalla formula `OSC` della chiave `FRM_OSC`, a seguito dell'elaborazione. Si ricorda che tali valori sono stati normalizzati dividendo, per ogni grandezza, ciascun valore per il massimo ottenuto in questa prova e in tutte quelle registrate nel training set.
Si nota inoltre che sono riportati tutti i quindici punti effettuati dalla prova Lambda Step, generati dalla combinazione delle tre Velocità e dei cinque carichi

[5] Questo gruppo di formule contiene esclusivamente quelle necessarie all'elaborazione delle prove di tipo Lambda Step.

richiesti dalla Norma, che dimostrano dunque che tutte le iterazioni elementari sono state compiute con successo dall'automazione e di conseguenza è stato possibile ottenere tutti i valori riportati di `RESULT1:FRM_OSC'OSC`. Per ciascuno di essi è a questo punto possibile effettuare una predizione mediante AITV da confrontare con il corrispondente valore calcolato di `OSC`, che ci permette di considerare valido e affidabile il risultato del test, con maggiore confidenza.
Il training set di riferimento per questa prova conta 105 campioni al momento dell'esecuzione della prova **LambdaStepTest** ed è contenuto nel file *LStep_TrainingSet.txt*, collocato nella solita cartella *lib*[6], contenente file, script, finestre e altri elementi necessari al funzionamento del sistema realizzato. Tramite questa collezione di campioni con l'associazione tra i valori dei tre dataset d'ingresso **T_ATS**, **qm_air** e **Conc_O2_EngineOut** e il valore di uscita **OSC**, è possibile ottenere una buona predizione, mediante una o due variabili di input tra le tre disponibili.
Si procede dunque con la validazione dei risultati ottenuti e a titolo d'esempio si prova ad utilizzare AITV per **predire il risultato del quarto punto effettuato**, corrispondente alla prima Velocità di demand e al quarto set di carico. Nalla tabella 4.1 si legge che per questa iterazione del Lambda Step:

- il valore iniziale (normalizzato) di `T_ATS`, cioè la temperatura dell'ATS, è `0.788515`;
- il valore iniziale (normalizzato) di `qm_air`, cioè la portata d'aria in ingresso al motore, è `0.494584`;
- il valore iniziale (normalizzato) di `Conc_O2_EngineOut`, cioè la concentrazione volumetrica d'ossigeno a monte dell'ATS, è `0.644684`;
- il valore corrispondente (normalizzato) di `OSC`, cioè la Oxygen Storage Capacity calcolata dalla post-elaborazione, è `0.534532`.

É dunque lanciato il Job relativo al Validatore AITV, che presenta la finestra di Dialog con cui sono esplicitati i parametri per la predizione, per la quale **si vuole utilizzare come feature di input la temperatura dell'ATS e la portata dell'aria in ingresso al motore**. Com'è mostrato in fig. 4.4, è allora selezionata come prima variabile d'ingresso `T_ATS` ed è spuntato il CheckBox per l'abilitazione anche della seconda variabile d'ingresso, la cui scelta ricade su `qm_air`, mentre il dataset risultato di output, che costituisce l'oggetto della validazione, è naturalmente `FRM_OSC'OSC`. Dal ListBox contenente tutti i punti del dataset risultato, è dunque selezionato il quarto elemento e di conseguenza saranno aggiornate le caselle di testo sulla sinistra con i valori di `T_ATS` e `qm_air` relativi al quarto punto del Lambda Step effettuato. Infine è impostato un **Learning Rate pari a 0.3** e sono richieste **1500 iterazioni**. A questo punto non

[6]Posizionata nel filesystem della risorsa di archiviazione condivisa nella rete aziendale, in particolare nell'area associata al Datasource `AdaMo_Files`.

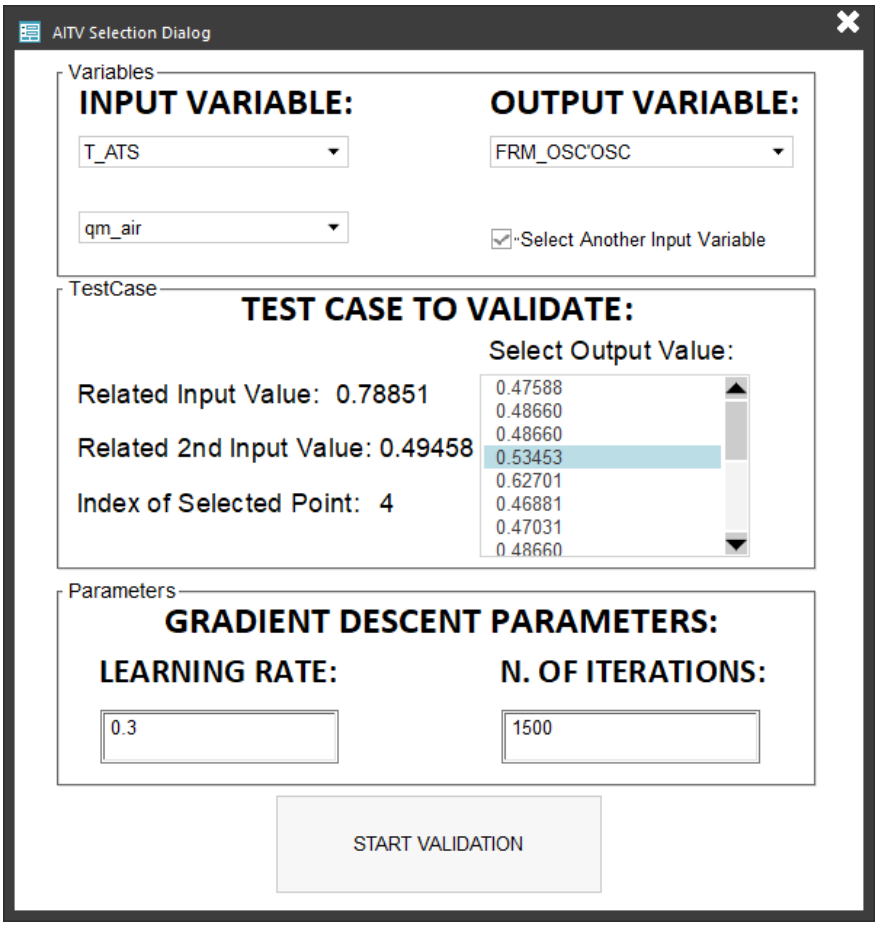

Figura 4.4: Selezione dei parametri per la predizione di AITV sul quarto campione della prova *LambdaStepTest*.

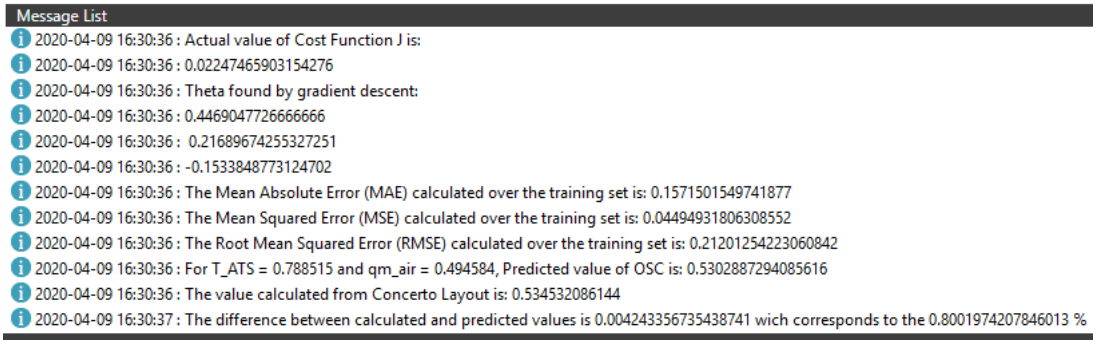

Figura 4.5: Output testuale di AITV sulla finestra dei messaggi di Concerto, nel corso della validazione del quarto campione della prova *LambdaStepTest*.

resta che confermare l'avvio dell'algoritmo di validazione con il Button *START VALIDATION* e attendere che il software crei il modello predittivo con il Gradient Descent secondo i parametri selezionati e lo applichi per la combinazione di input selezionata.
Il risultato di questo processo consiste in primo luogo nella raccolta di informazioni mostrate nella *Message Window* di Concerto, stampate a mano a mano che l'algoritmo di AITV proseguiva nelle varie fasi descritte nel capitolo precedente. L'output testuale, visibile anche in fig. 4.5 nella Message List, è il seguente:

Actual value of Cost Function J is:
0.022474659
Theta found by gradient descent:
0.446904773
0.216896743
-0.153384877
The Mean Absolute Error (MAE) calculated over the training set is:
0.1571501549741877
The Mean Squared Error (MSE) calculated over the training set is:
0.04494931806308552
The Root Mean Squared Error (RMSE) calculated over the training set is:
0.21201254223060842
For T_ ATS = 0.788515 and qm_ air = 0.494584 , Predicted value of OSC is:
0.5302887294085616
The value calculated from Concerto Layout is: 0.534532086144
The difference between calculated and predicted values is 0.004243356735438408
wich corresponds to the 0.8001974207846013 %

Si commentano di seguito i buoni risultati di questa predizione, andando per ordine.

1. É riportato l'ultimo valore calcolato della funzione di costo che fa riferimento alla $J(\Theta)$ associata all'ipotesi finale $h_\theta(x)$ ottenuta dal Gradient Descent. Tale valore è approssimativamente pari a 0.02247 e possiamo dire che potrebbe rappresentare il minimo assoluto della funzione di costo, dal momento che abbiamo adoperato un **Gradient Descent di tipo Batch** che assicura la convergenza dell'algoritmo, sebbene preveda un costo computazionale maggiore (sono infatti di volta in volta adoperati tutti i campioni del training set per l'aggiornamento dei θ_k, com'è stato descritto nel paragrafo 3.4.1).

2. Sono trascritti i valori finali dei θ_k, che identificano la funzione ipotesi finale $h_\theta(x)$:

    - $\theta_0 = 0.4469$ circa

- $\theta_1 = 0.2169$ circa
- $\theta_2 = -0.15338$ circa

3. Sono calcolati i tre indici d'errore relativi alla Regressione Lineare a due variabili realizzata dal Gradient Descent:

   - $MAE = 0.15715$ circa
   - $MSE = 0.04495$ circa
   - $RMSE = 0.21201$ circa

   Nel paragrafo 3.4.1 è stato spiegato che il MAE è un indicatore di semplice interpretazione, molto robusto rispetto alla presenza di eventuali outlier ma proprio per questo non tiene in conto importanti differenze tra il valore predetto sul training case e quello effettivamente registrato. Al contrario MSE e RMSE penalizzano maggiormente gli errori molto elevati e proprio per questo motivo, dato che il valore di Oxygen Storage costituisce una caratteristica cruciale dell'ATS per analizzarne l'impatto sulle emissioni, si preferisce porre maggior risalto su questi ultimi due indicatori pur tenendo in conto la loro suscettibilità alla presenza di outlier. I valori ottenuti sono comunque buoni, ma indicano la presenza di alcuni training case sui quali il modello non performa bene: il fatto che sia molto basso l'errore ottenuto nella predizione del quarto elemento scelto per questo esempio in particolare, non ci fa trascurare la necessità di doverlo adattare meglio ai campioni del training set, magari lanciando un'altra predizione con AITV nella quale sia scelta un'altra variabile di input che possa dimostrarsi più idonea.

4. É finalmente espresso il valore predetto per `OSC` a partire dai dati di `T_ATS` $= 0.788515$ e `qm_air` $= 0.494584$, che è pari a circa 0.53029, a fronte di un Oxygen Storage di circa 0.53453, calcolato invece dalla post-elaborazione. La differenza in termini assoluti è approssimativamente 0.00424, pari dunque a circa lo 0.8002 %. Indubbiamente **ottenere un errore al di sotto dell'1% ci permette di dire che il modello ha effettuato un'ottima predizione**, grazie alla quale possiamo ritenere attendibile il risultato restituito dai calcoli del post-processing.

In conclusione, il processo di predizione del quarto valore di Oxygen Storage relativo alla prova *LambdaStepTest* ha prodotto un risultato molto buono, poiché molto vicino a quello calcolato dalla formula `FRM_OSC'OSC`, addirittura a meno dell'1%. Questo ci permette di considerare valido il risultato dell'elaborazione e di conseguenza ci consente di ritenere correttamente realizzata tutta la procedura di test:

- il sistema d'automazione, secondo l'algoritmo di prova implementato, ha eseguito un buon controllo sul motore e ha coordinato adeguatamente l'application system per il set del lambda obiettivo;

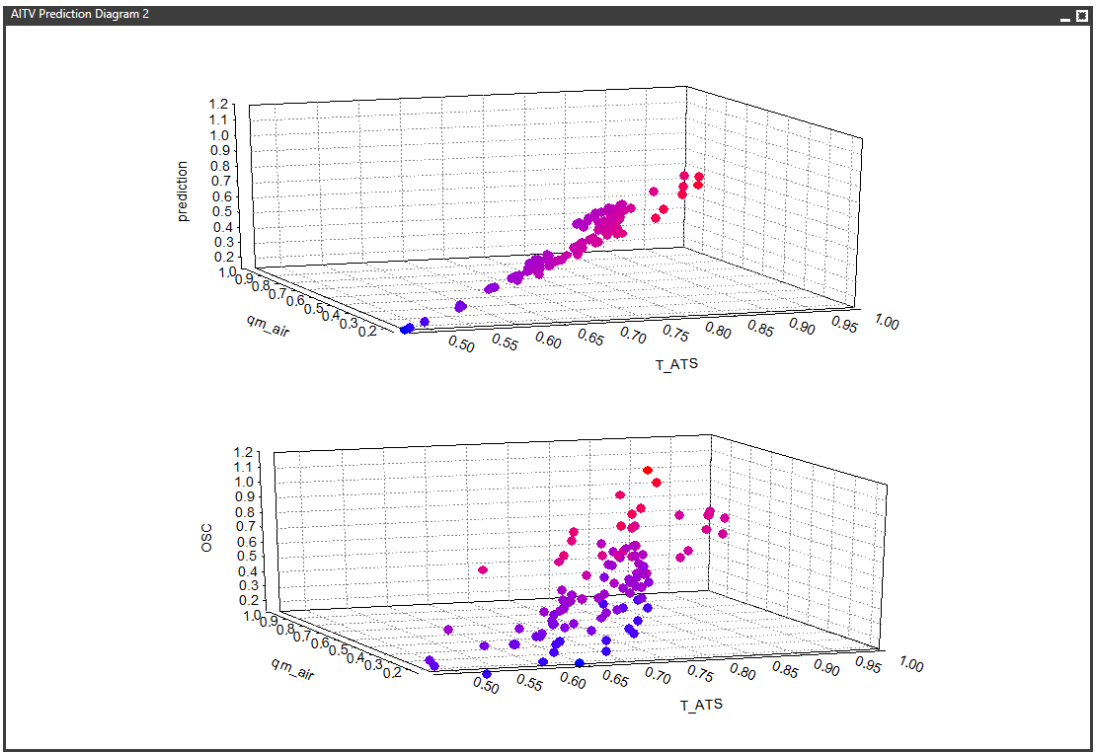

Figura 4.6: Diagram Scatter 3D che mostra il modello ottenuto per la validazione del quarto campione della prova *LambdaStepTest*.

- l'application system si è mostrato efficiente nel corso della prova, eseguendo la richiesta di lambda obiettivo e raccogliendo tutti i parametri da mandare via ASAP3 al sistema d'automazione;
- tutti gli strumenti di misura e i sensori coinvolti nella prova per il calcolo di `OSC` hanno trasmesso valori attendibili all'automazione, permanendo nello stato di sampling senza errori;
- il formato dei dati si è dimostrato correttamente strutturato nel file di risultati, che è completo e consistente, dal momento che il layout e le formule di post-elaborazione, opportunamente realizzati, lo hanno processato senza intoppi.

In fig. 4.5 è riportata una rappresentazione grafica del modello regressivo ottenuto in questo caso di studio, sotto forma di **Diagram Scatter 3D**, ottenuto da un'elaborazione della window automaticamente restituita al termine dell'algoritmo di AITV, già illustrata nel paragrafo 3.4.2 (fig. 3.20). Nella parte inferiore della finestra è visibile la distribuzione dei valori di Oxygen Storage raccolti nel training set, in funzione delle due variabili di ingresso scelte per la regressione, che corrispondono alla temperatura dell'ATS e alla portata d'aria in ingresso al motore. Il grafico superiore invece mostra come varia la predizione in funzione

degli stessi ingressi, rendendo evidente una buona risposta nel caso di campioni che ricadono nella stessa zona di quello su cui è stata effettuata la validazione, che presentava `T_ATS` = 0.788515 e `qm_air` = 0.494584.
A questo punto è possibile continuare il processo di validazione sui restanti punti della prova di Lambda Step effettuata, in modo analogo a quanto descritto per il quarto campione di questo *LambdaStepTest*. Infine possono essere aggiunti tutti i dati acquisiti e calcolati nel training set tramite il Job del **Parser**, che aggiunge al training set contenuto in *LStep_ TrainingSet.txt* una nuova entry per ogni riga della tabella 4.1, composta dai valori di **T_ATS**, **qm_air** e **Conc_O2_EngineOut**, associati ad ogni iterazione elementare effettuata in *LambdaSteptest*, insieme al corrispondente valore di **OSC** calcolato. In questo modo tali dati, precedentemente validati, contribuiranno a corroborare il processo di validazione performato da AITV, a vantaggio di test futuri.

Capitolo 5

Conclusioni e sviluppi futuri

Questo elaborato ambisce a costituire una stesura il più possibile completa di tutti i dettagli del corposo lavoro, condotto nel corso di questi mesi, necessario a realizzare l'architettura del sistema proposto. Nel primo capitolo sono stati introdotti i concetti basilari legati alle attività di ricerca in sala prova motori, alla programmazione degli algoritmi di test sui sistemi d'automazione, alle metodologie di archiviazione e all'implementazione delle strutture di post-processing dei dati, gettando così le fondamenta per la comprensione dell'architettura delineata nel secondo capitolo e approfondita sugli aspetti più tecnici nel terzo. La prima grande sfida di questo progetto consiste proprio nel tracciare una linea di continuità tra tutte le discipline contemplate dall'attività di testing in sala prova, che coinvolgono molteplici strumenti, competenze diversificate e un personale numeroso. Una seconda sfida riguarda invece l'integrazione di una nuova tecnica, da porre a servizio di tutte le competenze già messe in gioco, per creare un valore aggiunto che contribuisca ad accrescere la qualità dell'attività svolta: l'Intelligenza Artificiale.
Si può intendere facilmente quanto sia stato impegnativo, ma al contempo proficuo, arrivare a creare il sistema descritto in questa tesi, che è in grado di contare su un'implementazione coerente degli algoritmi di test e delle relative maschere di elaborazione, su un meccanismo di standardizzazione del formato dei dati, su una struttura d'archiviazione centralizzata e su una nuova procedura di validazione dei risultati mediante il Machine Learning. Sebbene la realizzazione di quest'ultimo strumento costituisca il principale obiettivo di questo lavoro, possiamo affermare che la struttura su cui si poggia il validatore di **AITV** riveste un ruolo ugualmente importante nella presentazione dei risultati del progetto, grazie all'**efficienza** mostrata nel suo impiego sul caso di studio, che promette di garantire in generale una gestione ancora più fluida della piattaforma di test del nostro centro. Il capitolo 4 ha infatti descritto tutte le procedure e il flusso di dati teorizzati e descritti in precedenza, dal punto di vista di una reale applicazione, di estremo interesse data la ricorrente necessità di effettuare prove di

Lambda Step.
In questa sede, tuttavia, si vuole soprattutto discutere dell'esito del processo di validazione, applicato ad uno dei risultati del test preso in esame, che ha mostrato un'ottima risposta di AITV. La **predizione effettuata sul valore sottoposto a validazione si è discostata di meno dell'1% dal quello realmente calcolato** e questo permette sicuramente di considerare consistente l'esito della prova e valida l'intera procedura di test, come è stato spiegato al termine del paragrafo precedente. Non va però dimenticato che gli indici d'errore MAE, MSE e RMSE, ottenuti dal modello predittivo in risposta allo stesso training set su cui è stato allenato, testimoniano la presenza di alcuni training case per i quali la funzione ipotesi finale non si adatta al meglio. D'altro canto anche una comparazione dei grafici tridimensionali, riportati in fig. 4.6, dimostra che la Regressione Lineare a due variabili effettuata mediante il Gradient Descent risulta più accurata in una zona circoscritta del piano definito dalle variabili di input T_ATS e qm_air, nella quale la stima di OSC si avvicina molto al vero valore. Da un lato eseguire nuovamente l'algoritmo di apprendimento, scegliendo altre feature d'ingresso o impostando una diversa parametrizzazione di Learning Rate e numero di iterazioni, potrebbe portare a un migliore adattamento al training set, ma d'altronde possono verificarsi circostanze in cui con ogni combinazione di ingressi il modello non risponde adeguatamente né a nuovi campioni né a quelli del training set. Sono state individuate alcune soluzioni più robuste per affrontare questo margine di miglioramento, che sono proposte di seguito insieme alle altre **principali prospettive di sviluppo futuro**.

1. Potrebbe essere utile potenziando l'espressività del modello regressivo proposto, aumentandone la complessità. In prima istanza l'implementazione di una **funzione ipotesi di grado maggiore di 1** permetterebbe di articolare l'espressione di $h_\theta(x)$ in modo tale da conferire alla superficie descritta dai valori predetti, in funzione delle due variabili di input, una forma più simile a quella descritta dai valori reali. A questa soluzione potrebbe essere abbinata in secondo luogo **la possibilità di utilizzare più di due variabili** per la Regressione con Gradient Descent, per raggiungere un grado di espressività del modello decisamente superiore. Ci sono tuttavia non poche difficoltà legate a queste iniziative: se è vero che un'espressività insufficiente può portare all'*underfitting*, che comporta uno scarso adattamento del modello sia ai training case sia a nuovi campioni, è altrettanto vero che un'espressività eccessiva può causare l'*overfitting*, che invece consiste in un'estrema aderenza del modello al training set, compromettendo così la sua efficacia su nuovi dati. In merito alla ricerca di un trade-off che costituisca il giusto mezzo tra questi due estremi, è necessario un procedimento di *overfitting addressing*, imponendo ad esempio la riduzione del numero di feature d'ingresso, tramite selezione manuale o algoritmica. La maggiore espressività del modello è tuttavia la ragione che ci spinge a cercare di sfruttare nel modo più opportuno tutte le variabili di input a disposizione

e, per non rinunciarvi, la teoria del Machine Learning suggerisce di ricorrere alla tecnica di **regolarizzazione**, che permette di smussare una curva di predizione in *overfitting* rendendo più piccoli i θ_k delle feature polinomiali. Per fare questo è necessario moltiplicare ogni termine della funzione ipotesi h_θ per una costante λ molto elevata, in modo tale che l'algoritmo di Gradient Descent tenda a minimizzare i θ_k relativi alle feature di grado maggiore. La funzione di costo apparrebbe così modificata:

$$J(\Theta) = \frac{1}{2m}(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{k=1}^{n} \theta_k^2)$$

dove:

- m è il numero di istanze del training set;
- $\frac{1}{2m}$ è il fattore di normalizzazione;
- $(h_\theta(x^{(i)}) - y^{(i)})$ è l'errore tra l'i-esimo valore stimato - a partire dall'i-esimo valore di input $x^{(i)}$ - e l'i-esimo valore vero $y^{(i)}$ del training set;
- n è il numero di feature d'ingresso e dunque $n + 1$ è il numero dei θ_k;
- λ è il fattore di regolarizzazione.

Si noti che il fattore di regolarizzazione non è applicato a θ_0, poiché è una costante e dunque un termine di grado zero. Per questo motivo un valore di λ troppo elevato potrebbe ricondurre il modello a una condizione di *underfitting*, dal momento che tutti i θ_k sarebbero quasi nulli, ad eccezione proprio di θ_0, riconducendo la curva dell'ipotesi ad una sorta di retta. Al contrario un valore troppo basso di λ non produrrebbe alcuno smussamento, causando inevitabilmente l'*overfitting*.
Alla luce di quest'analisi, è immediato dedurre che sarebbe necessario un studio molto approfondito della questione, soprattutto in virtù di tutte le peculiarità che riguardano le grandezze misurate e calcolate nei test motoristici di sala prova, le cui correlazioni sono molto spesso di natura estremamente complessa. Solo un'indagine accurata, che combini diverse sperimentazioni alle premesse teoriche relative alle tecniche di Machine Learning e al funzionamento dei motori a combustione interna, potrebbe consentire di realizzare e padroneggiare al meglio modelli predittivi di maggiore espressività. É perciò più opportuno fare esperienza del nuovo strumento proposto in questa tesi così com'è stato concepito e sviluppato, adoperandolo su un numero adeguato di test, in modo tale da esplorarne i limiti e facilitare le scelte progettuali su una futura implementazione di modelli più espressivi.

2. Un altro passaggio evolutivo interessante a cui sottoporre lo strumento di validazione AITV, consiste nel rendere AITV **un Machine Learning System a tutti gli effetti**, dotato cioè di ogni caratteristica e funzionalità necessaria a massimizzare la qualità dei modelli generati. Alcuni dei fondamentali accorgimenti prescritti a tale scopo sono in realtà già state adottati in questo progetto, basti pensare alla **normalizzazione delle feature d'ingresso** con il metodo **Z-Score** o alla procedura di **selezione delle feature** demandata all'utente in quanto esperto del campo d'applicazione o, ancora, alla valutazione dell'ipotesi per mezzo del **calcolo degli indici d'errore MAE, MSE e RMSE**. Ci sono tuttavia almeno altri due procedimenti che possono conferire maggiore completezza allo strumento di predizione, i quali sono di seguito riportati come proposte per un futuro sviluppo:

   - Come in più occasioni è stato evidenziato in questa tesi, molto spesso i dati memorizzati nel training set possono apparire "sporchi", poiché affetti dalla presenza di **outlier**, i quali possono compromettere le prestazioni del modello generato portando a predizioni errate sulla base di un apprendimento errato, un fenomeno noto come **GIGO** (Garbage In, Garbage Out). Per arginare questo problema, è possibile "ripulire" il set di campioni utilizzando una nuova rappresentazione dei valori mediante il metodo **Box Plot**, con il quale il range di ogni feature è suddiviso in *quartili*: si calcola la mediana Q_2 di tutti i valori, da cui si ottengono le altre mediane Q_1 e Q_3 dei valori rispettivamente al di sotto e al di sopra di Q_2. La differenza $IQR = Q_3 - Q_1$, detta **interquartile**, è usata per identificare l'intervallo $[Q_1 - 1.5 \cdot IQR; Q_3 + 1.5 \cdot IQR]$ di valori attendibili e ogni dato al di fuori di esso è ritenuto un *outlier*.
     In questo progetto non è stato ritenuto opportuno adottare questa tecnica di manipolazione dei dati, in primo luogo perché sono ancora da valutare le potenzialità attuali di AITV sui dati attualmente presenti nel training set, ma in secondo luogo è necessario uno studio dell'impatto che questa procedura potrebbe avere sui valori salvati delle grandezze di nostro interesse, dato che comporta il **rischio di eliminare dati comunque significativi** per i test effettuati.
   - Attualmente la fase di **valutazione del modello** ottenuto da AITV consiste nel semplice calcolo degli indici d'errore sullo stesso set di dati d'addestramento, ma non consente di identificare la **migliore ipotesi possibile**, tra quelle ottenute dai diversi utilizzi del Machine Learning System. Per fare questo, andrebbe diviso il set di dati disponibili, attualmente dedicato per intero all'apprendimento diretto, in tre *subset*:

(a) **Training Set**, contenente il 60% dei campioni totali, è utilizzato per l'addestramento del modello attraverso il Gradient Descent;

(b) **Validation Set**, contenente il 20% dei campioni totali, è utilizzato per valutare su un nuovo insieme di dati ciascuna ipotesi ottenuta, per mezzo dei tre indici d'errore MAE, MSE e RMSE;

(c) **Test Set**, contenente il 20% dei campioni totali, è utilizzato per valutare l'ipotesi che meglio ha performato sul Validation Set.

In alternativa si potrebbe adoperare la tecnica di **Cross Validation**, che suggerisce suddividere il set di campioni disponibili semplicemente in **Training Set** (80%) e **Tesst Set** (20%), con il metodo dell'**Holdout Splitting**, avendo premura di dividere equamente le occorrenze dei valori con la procedura di *stratificazione*. Ciascun modello è dunque addestrato sul Training Set e valutato sul Test Set, per permettere la scelta dell'ipotesi migliore.
La tecnica che però sarebbe più consigliata è quella della **K-folds Cross Validation**, che per ciascuna ipotesi divide il dataset in K parti uguali, ognuna delle quali è usata a turno come Test Set, mentre le restanti $K-1$ costituiscono il Training Set. Il processo è iterato in modo tale che ogni osservazione è utilizzata sia per l'addestramento che per la valutazione. Per permettere inoltre una migliore suddivisione dei campioni, è possibile scegliere in modo casuale gli elementi che compongono ciascun *k-fold* con il metodo del **Random Subsampling**.
La dimensione attuale dei set di dati osservati per ogni tipologia di test di sala prova è, tuttavia, troppo ridotta per consentire una suddivisione del genere, rendendo poco efficace l'addestramento o poco significativa la valutazione. Nel momento in cui saranno accumulati dati a sufficienza per mezzo del frequente uso di AITV e soprattutto del Parser che immagazzina i nuovi campioni, si potrà strutturare una procedura di ricerca del miglior modello possibile, per mezzo di successive parametrizzazioni dell'algoritmo, secondo le metodologie esposte nel paragrafo 3.4.2.

3. L'attuale rappresentazione dei risultati dei test di sala prova, come grandezze numeriche, vincola l'implementazione del Machine Learning System all'utilizzo della Regressione Lineare, concepita per predire valori numerici reali. Se si escogitassero nuove metodologie per esprimere gli esiti dei test, in modo tale da ridurli a una suddivisione di valori discreti o **classi**, sarebbe possibile ricorrere anche alle tecniche di **Regressione Logistica** o, appunto, **Classificazione**. Si consideri ad esempio il caso di un ciclo omologativo: il fatto che il quantitativo di CO2 emessa, espresso in g/kWh, sia al di sotto o al di sopra del limite della normativa vigente potrebbe essere ritenuto un risultato *booleano* e dunque discreto, che può valere `True` o

`False`.

La prospettiva che si apre diventa ancora più interessante se si considera che la Classificazione è il metodo su cui si basano le **Reti Neurali**, che in sostanza replicano la Regressione Logistica su più livelli, permettendo di esprimere in questo modo **relazioni molto complesse** tra le variabili utilizzate come feature d'ingresso e il corrispondente valore di output da predire.

Naturalmente tutte le iniziative riportate in precedenza costituiscono un lavoro che può degnamente essere tema di un altro corposo progetto, al pari di questa tesi. Nella maggior parte dei casi si tratta di prospettive che necessitano di un'importante fase preliminare di raccolta di dati e informazioni, ma soprattutto di esperienza del sistema realizzato in questa tesi.
In conclusione, i risultati raggiunti vanno considerati sicuramente positivi, dal momento che questo lavoro ha consegnato una struttura solida e molto utile a prescindere dalle nuove tecniche di validazione dei test adoperate. Inoltre la qualità ed espressività dei modelli attualmente generabili con AITV rendono questo strumento già abbastanza affidabile e di conseguenza funzionale per le attività di ricerca nel nostro centro, di cui costituisce da ora un'importante risorsa. L'elemento di novità introdotto, gli ottimi risultati ottenuti sul caso di studio e le interessanti prospettive suscitate costituiscono motivo di soddisfazione per il lavoro compiuto.

Bibliografia

[1] M. Apschner and G. Hauser. Puma open—prüfsystem für motor und antriebsstrang. *MTZ-Motortechnische Zeitschrift*, 62(3):228–234, 2001.

[2] AVL List GmbH. *AVL Concerto 5 Next Level Data Processing.* March 2019. Version 5 R3.1.

[3] R. Bartz. Introduction: Asam-ods (open data services), 2000.

[4] C. Elliott, V. Vijayakumar, W. Zink, and R. Hansen. National instruments labview: a programming environment for laboratory automation and measurement. *JALA: Journal of the Association for Laboratory Automation*, 12(1):17–24, 2007.

[5] I. ETAS. Software-produkte. *URL:< http://www. etas. com/de/products/inca_ software_ products. php>, verfügbar am*, 22, 2009.

[6] J. Gong, D. Wang, A. Brahma, J. Li, N. Currier, A. Yezerets, and P. Chen. Lean breakthrough phenomena analysis for twc obd on a natural gas engine using a dual-site dynamic oxygen storage capacity model. Technical report, SAE Technical Paper, 2017.

[7] L. Jirong and Z. Zaimin. Overview of asap standards for calibration of automotive onboard controllers [j]. *Automobile Technology*, 10:1–4, 2004.

[8] C. Kammerer, R. Schmidt, and G. Hochmann. A common testing platform for engine and vehicle testbeds. *ATZ worldwide*, 111(11):40–44, 2009.

[9] W. Melder and R. Mueller. Asam/ods: Standardized exchange of offline data; asam/ods: standardisierter austausch von offline-daten. 2001.

[10] R. No. 49. uniform provisions concerning the measures to be taken against the emission of gaseous and particulate pollutants from compression-ignition engines and positive ignition engines for use in vehicles. *Off J Eur Union*, 2013.

[11] A. M. P. PEMS. 494. *US EPA APPROVED COMBINATION OF REAL TIME SOOT AND INTEGRAL PM MEASUREMENT FOR IN-USE TESTING, AVL MSS*.

[12] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[13] A. Scheibelmasser, J. Menhart, and B. Eichberger. Improvements in the field of device integration into automation systems with embedded web interfaces. In *ICINCO-ICSO*, pages 94–99. Citeseer, 2008.

[14] B. Virnich, J. Hobelsberger, and C. Rucker. Das asam/ods transport format, auch kurz'atf', als firmenubergreifendes datenformat. *FORTSCHRITTE DER AKUSTIK*, 32(1):173, 2006.

[15] V. Yodaiken et al. The rtlinux manifesto. In *Proc. of the 5th Linux Expo*, 1999.