ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea Triennale in Informatica per il Management

MUTATION TESTING PER SISTEMI PLC:
Progettazione e validazione di un framework per il linguaggio Structured Text

Tesi di Laurea in Ingegneria del Software

Relatore:
Prof.
Davide Rossi

Presentata da:
Alessandro Capra

Sessione Marzo 2026
Anno Accademico 2025/2026

Indice

Introduzione

L'automazione industriale oggi si trova ad affrontare sistemi di controllo sempre più complessi, che richiedono software di alta qualità. Al centro di questo settore c'è il controllore logico programmabile (PLC). Per programmarne la logica, lo standard di riferimento è il linguaggio *Structured Text* (ST), definito dalla norma IEC 61131-3. [1]

In un contesto produttivo ad alta precisione, l'affidabilità del codice non è solo un requisito tecnico, ma una condizione necessaria per la sicurezza. Questa tesi nasce da una collaborazione con il gruppo Coesia, leader nelle macchine automatiche, con l'obiettivo di analizzare e migliorare i processi di verifica del software per i PLC. Spesso, infatti, non basta che il software superi i test previsti: bisogna capire se quei test siano davvero capaci di individuare errori sottili o critici.

Per rispondere a questa esigenza, il lavoro si focalizza sul *Mutation Testing*. Si tratta di una tecnica che inserisce piccoli errori (mutanti) nel codice sorgente per misurare quanto il sistema di verifica sia robusto. [2] Se i test non rilevano l'errore introdotto, significa che la suite di test va migliorata.

L'obiettivo della tesi è lo sviluppo di un framework per la *Mutation Generation*. Il tool realizzato non si occupa dell'esecuzione dei test, ma della fase cruciale della generazione automatica dei mutanti: partendo dal codice sorgente in *Structured Text*, il sistema ne analizza la struttura sintattica e produce diverse varianti difettose. Questo permette di mettere sotto sforzo i test aziendali e valutarne l'efficacia reale.

Il lavoro è strutturato in cinque capitoli:

- Il **Capitolo 1** descrive il contesto tecnologico, lo standard IEC 61131-3 e le criticità attuali nella validazione del linguaggio *Structured Text*.
- Il **Capitolo 2** approfondisce la teoria del *Mutation Testing* e gli strumenti per l'analisi sintattica, in particolare l'uso di ANTLR4 per generare l'*Abstract Syntax Tree* (AST).
- Il **Capitolo 3** illustra l'architettura del sistema, l'ambiente di sviluppo e le scelte metodologiche fatte per gestire il progetto.
- Il **Capitolo 4** descrive la parte tecnica del motore di mutazione: come vengono visitati gli alberi sintattici e come sono stati implementati i diversi operatori di mutazione.
- Il **Capitolo 5** espone i risultati della fase di test e validazione, discutendo le prestazioni del motore di mutazione e l'impatto degli operatori definiti sulla valutazione della robustezza del codice.

L'elaborato si conclude con un'analisi dei risultati raggiunti e una riflessione sulle prospettive future, con particolare attenzione all'integrazione di questi strumenti nei moderni flussi di sviluppo assistiti da modelli di intelligenza artificiale.

Capitolo 1

Il Contesto Tecnologico e Normativo

1.1 L'automazione industriale e lo standard IEC 61131-3

Nel settore dell'automazione, la programmazione dei controllori logici (PLC) fa riferimento alla normativa internazionale **IEC 61131-3**.[1] Questo standard è nato per rispondere a una necessità pratica: mettere ordine in un mercato dove ogni produttore utilizzava linguaggi proprietari. Definendo uno standard comune, la norma permette di uniformare lo sviluppo del software che gestisce i movimenti e le logiche delle macchine automatiche.

In realtà tecnologicamente avanzate come Coesia, lo *Structured Text* (ST) si è imposto come il linguaggio principale tra quelli previsti dalla norma. Essendo un linguaggio testuale simile a quelli di alto livello (come il C o il Pascal), lo ST è ideale per gestire la complessità della meccatronica moderna: dai calcoli matematici per il controllo dei motori alla gestione dei grandi flussi di dati che le macchine attuali devono elaborare. L'uso dello ST permette di scrivere codice modulare e riutilizzabile su diversi modelli di macchine. Tuttavia, questa flessibilità aumenta la complessità del software, richiedendo strumenti di analisi sempre più precisi per garantirne la qualità.

1.2 Il linguaggio Structured Text (ST): caratteristiche e criticità

Lo ST eredita la chiarezza dei linguaggi procedurali, permettendo di strutturare algoritmi complessi con costrutti familiari come i cicli `FOR` o le istruzioni `IF-THEN-ELSE`. Se da un lato questo facilita il lavoro del programmatore, dall'altro introduce un rischio: la libertà espressiva richiede una disciplina rigorosa per evitare che il codice diventi difficile da mantenere nel tempo.

Una delle criticità maggiori riguarda gli errori logici "silenziosi". Poiché un PLC opera ad altissima velocità, un errore banale (come usare `>` al posto di `>=`) potrebbe non bloccare subito la macchina, ma causare comportamenti instabili o un'usura meccanica precoce che emerge solo dopo mesi di lavoro. A differenza dei linguaggi grafici, dove un errore può essere visibile visivamente, nello ST l'errore logico rimane "nascosto" tra migliaia di righe di testo. Diventa quindi fondamentale utilizzare tecniche di analisi sintattica per scomporre il codice e applicare metodi di verifica avanzati.

1.3 Verso l'Automazione Intelligente: l'uso degli LLM nella generazione di codice industriale

L'evoluzione tecnologica sta portando il settore dell'automazione verso l'integrazione di modelli di intelligenza artificiale avanzati, noti come *Large Language Models* (LLM). Questi modelli, addestrati su enormi quantità di dati testuali e codice sorgente, stanno dimostrando capacità sorprendenti nella generazione automatica di script e programmi.[3] In un contesto come quello di Coesia, l'IA può assistere i programmatori nella scrittura del codice ST, con l'obiettivo di ridurre i tempi di sviluppo e standardizzare la qualità della produzione.

Tuttavia, l'automazione industriale ha vincoli molto più severi rispetto ad altri ambiti informatici. Una riga di codice errata può causare danni fisici o rischi per la sicurezza.[4] L'IA non può quindi essere usata "da sola", ma deve far parte di un ecosistema di validazione rigoroso. Per rendere questi modelli affidabili, occorrono dataset che contengano non solo esempi corretti, ma anche una casistica dettagliata di errori. Serve, in sostanza, un sistema capace di generare varianti errate del codice in modo controllato, così da testare quanto l'IA (e chi la controlla) sia capace di riconoscerle.

1.4 Il problema della validazione del software generato automaticamente

L'uso dell'intelligenza artificiale cambia il modo in cui dobbiamo testare il software. Se un programmatore umano tende a fare certi tipi di errori, un LLM potrebbe generarne di diversi, spesso molto sottili e difficili da intercettare con i controlli standard. In questo scenario, la robustezza della suite di test diventa l'ultima difesa per l'integrità del sistema.

Il problema è che un test può confermare che il software "funziona", ma non garantisce di saper trovare ogni bug. Se la suite di test non è abbastanza "aggressiva", il codice generato dall'IA potrebbe essere approvato nonostante errori latenti.[2] Non basta più testare il software: bisogna validare l'efficacia degli strumenti di test stessi.

Il *Mutation Testing* risponde esattamente a questa esigenza. Inserendo sistematicamente dei "mutanti" (errori artificiali), possiamo sottoporre i test aziendali a uno stress-test oggettivo.[5] Se i test non riescono a identificare queste piccole modifiche, significa che il sistema di validazione va migliorato. Lo sviluppo di un motore di *Mutation Generation* per lo Structured Text, obiettivo di questa tesi, è la base per costruire un processo di controllo qualità solido e adatto alle sfide dell'industria moderna.

Capitolo 2

Mutation Testing e Analisi Sintattica

2.1 Fondamenti del Mutation Testing: mutanti, operatori e Mutation Score

Il *Mutation Testing* è una tecnica di analisi del software basata sull'iniezione di guasti (*fault-based*). Il suo scopo non è testare il codice, ma misurare quanto sia efficace la suite di test esistente, valutando se sia in grado di accorgersi di errori introdotti artificialmente.[2]

A differenza delle metriche di copertura tradizionali, come la *statement coverage* (che si limita a verificare se una riga viene eseguita) o la *branch coverage* (che controlla se ogni ramo decisionale viene percorso), il *Mutation Testing* va più in profondità. Non ci dice solo se il test "passa" per una certa riga, ma se quel test è davvero capace di distinguere tra una esecuzione corretta e una errata.

Il processo si divide in tre fasi principali:

1. **Generazione:** Partendo dal programma originale, vengono create diverse varianti chiamate mutanti. Ciascun mutante contiene una singola modifica sintattica rispetto all'originale.

2. **Esecuzione:** La suite di test viene eseguita su ogni mutante.

3. **Classificazione:** Se almeno un test fallisce, il mutante è considerato ucciso (*killed*). Se invece tutti i test passano nonostante l'errore, il

mutante è sopravvissuto (*lived*), segnalando una potenziale lacuna nei test.

Per automatizzare la creazione di tali varianti, si usano gli "Operatori di Mutazione", cioè regole che definiscono come modificare il codice (ad esempio cambiando un `+` con un `-`). Tuttavia, l'applicazione sistematica di tutti gli operatori possibili può generare un numero di mutanti estremamente elevato, rendendo il processo computazionalmente oneroso.

A tal proposito, il lavoro di **Offutt et al.**[6] risulta determinante: attraverso uno studio sperimentale dimostrano che è possibile selezionare un sottoinsieme di operatori "sufficienti" capaci di offrire una precisione di analisi quasi identica all'insieme completo. Questa strategia, nota come *selective mutation*, consente di ridurre significativamente il numero di mutanti da generare e analizzare, mantenendo elevata l'efficacia nella valutazione delle suite di test.

Successivamente, **Zhang et al.**[7] hanno analizzato se la selezione dei mutanti basata sugli operatori fosse effettivamente superiore a una selezione casuale di pari dimensione. I risultati mostrano che, a parità di numero di mutanti, la selezione casuale riesce spesso a ottenere risultati molto simili in termini di accuratezza. Questo suggerisce che non sia tanto importante il tipo di operatore scelto, quanto piuttosto il numero complessivo di mutanti considerati, rendendo quindi possibili strategie di riduzione più semplici ma comunque efficaci.

L'efficacia della suite di test viene infine riassunta dal **Mutation Score (MS)**, definito come:

$$MS = \left(\frac{K}{M - E} \right) \times 100$$

Dove:

- **K** rappresenta il numero di mutanti uccisi.
- **M** è il numero totale di mutanti generati.

- **E** indica i mutanti equivalenti, ovvero varianti che, pur differendo sintatticamente, mantengono un comportamento logico identico all'originale e sono, per definizione, impossibili da identificare per qualsiasi suite di test.[2]

Alla luce di queste considerazioni, il framework sviluppato in questa tesi automatizza la generazione dei mutanti "sufficienti", includendo un meccanismo di selezione configurabile che permette di controllare il numero di mutanti prodotti. L'obiettivo è trovare un buon equilibrio tra qualità dell'analisi e costo computazionale, applicando in modo pratico i principi di riduzione descritti in letteratura.

2.2 Teoria dei linguaggi: Lexing, Parsing e Abstract Syntax Tree (AST)

Per poter modificare il codice *Structured Text* in modo automatico, il framework non può limitarsi a leggere il file come una sequenza di caratteri. Deve "capire" la struttura logica del programma, trasformando il testo in una rappresentazione che il computer possa manipolare. Questo risultato si ottiene attraverso strumenti di analisi lessicale e sintattica, in grado di trasformare il testo del programma in una rappresentazione strutturata.

Questo processo avviene in tre passaggi:

- **Analisi Lessicale (Lexing):** Il *Lexer* scompone il codice in unità atomiche chiamate *token*. Ad esempio, una riga come `IF x > 10 THEN` viene frammentata in componenti riconosciuti come parole chiave (`IF`, `THEN`), variabili (`x`), operatori (`>`) o numeri (`10`).

- **Analisi Sintattica (Parsing):** Il *Parser* prende i *token* e verifica che rispettino le regole della grammatica IEC 61131-3[1]. Se la sintassi è corretta, organizza i pezzi in una struttura gerarchica.

- **Abstract Syntax Tree (AST):** È il risultato finale, un albero dove ogni nodo rappresenta un elemento del programma (un ciclo, un'assegnazione, ecc.).

L'uso dell'AST è fondamentale per questo progetto per due ragioni pratiche. Innanzitutto, garantisce la precisione: ci permette di essere certi che stiamo modificando un operatore logico e non, per esempio, un simbolo simile dentro un commento o una stringa. In secondo luogo, facilita la manipolazione: il motore di mutazione può navigare l'albero e sostituire un nodo (ad esempio un `>`) con il suo opposto (`<`), garantendo che il codice finale sia ancora sintatticamente valido.

2.3 ANTLR4: Il generatore di parser per il riconoscimento dello ST

Per implementare l'analisi descritta, il framework si affida ad **ANTLR4** (*ANother Tool for Language Recognition*)[8], un generatore di parser ampiamente impiegato per il riconoscimento di linguaggi strutturati.

È importante sottolineare che l'adozione di ANTLR4 non è stata una scelta *ex novo* di questo lavoro, ma deriva dal repository di partenza (che analizzeremo nel Capitolo 3) che già integrava questa tecnologia per il parsing dello *Structured Text*. La scelta è stata mantenuta per coerenza progettuale e per l'affidabilità dello strumento, che permette di gestire grammatiche complesse e generare automaticamente le classi Java necessarie per navigare l'AST.

Nel framework sviluppato, ANTLR4 funge da "traduttore": trasforma il codice ST in una struttura Java su cui il motore di mutazione può lavorare agevolmente, permettendoci di concentrarci sulla logica di generazione degli errori senza dover scrivere da zero un analizzatore sintattico.

Capitolo 3

L'Ambiente di Sviluppo e le Tecnologie Utilizzate

3.1 Analisi e personalizzazione del framework pre-esistente

L'implementazione di un motore di mutazione è un compito complesso che richiede, come base di partenza, un analizzatore sintattico estremamente preciso. Progettare un parser da zero per lo *Structured Text* avrebbe richiesto tempi di sviluppo incompatibili con gli obiettivi della tesi; per questo motivo, la fase iniziale del lavoro è stata dedicata alla ricerca e alla selezione di un framework open source affidabile.

Dopo una valutazione di diversi repository pubblici, la scelta è ricaduta sul progetto `iec61131-parser`,[9] sviluppato da V. Sitnikov. Si tratta di uno strumento robusto, basato su Java e ANTLR4, progettato specificamente per il riconoscimento dei linguaggi dello standard IEC 61131-3. I punti di forza che hanno determinato questa scelta sono stati la presenza di una grammatica ANTLR4 (file `.g4`) molto dettagliata e una struttura già predisposta per la trasformazione del codice in un albero sintattico (AST) navigabile.

Tuttavia, il framework originale era nato solo per "leggere" il codice e non per modificarlo. Risultava quindi troppo generico, pesante e non strutturato per la modifica dinamica del codice. Per usarlo nel mio progetto di tesi, ho dovuto effettuare un importante lavoro di pulizia e modifica.

Il primo intervento ha riguardato la rimozione di tutte le parti superflue.

Il framework originale, infatti, era progettato per gestire diversi linguaggi dell'automazione, compresi quelli grafici come il *Ladder Diagram* o il *Function Block Diagram*. Dato che il focus di questa tesi è esclusivamente sullo *Structured Text* (ST), ho eliminato i moduli relativi agli altri linguaggi, rendendo il programma finale molto più leggero, veloce e facile da gestire durante lo sviluppo.

Successivamente, mi sono concentrato sulla correzione della grammatica. Durante i primi test, è emerso che le regole originali del parser non riconoscevano correttamente alcuni costrutti dello *Structured Text*. Questo rappresentava un ostacolo, in quanto, se il codice non viene letto correttamente, è impossibile generare dei mutanti validi. Ho quindi modificato manualmente le regole sintattiche del parser, assicurandomi che ogni istruzione venisse interpretata senza errori.

Infine, l'aspetto più complesso ha riguardato la modifica della struttura Java. Nel progetto di partenza, l'albero sintattico era "statico", ovvero pensato solo per essere consultato. Per poter implementare il *Mutation Testing*, avevo invece bisogno di una struttura "dinamica", che permettesse di intervenire sui singoli nodi, ad esempio sostituendo un operatore di somma con uno di sottrazione, e di salvare poi il codice modificato in un nuovo file. Questo ha richiesto lo sviluppo di una logica specifica per la navigazione e la manipolazione dell'albero, un tema che verrà approfondito nel dettaglio nel Capitolo 4.

Questo lavoro ha permesso di trasformare una semplice libreria di lettura in un vero e proprio generatore di codice.

3.2 Stack Tecnologico: Java e Maven

Per lo sviluppo del software ho scelto strumenti standard, molto diffusi nel mondo del lavoro, per garantire che il tool sia facile da usare e da aggiornare in futuro:

- **Java 17:** Ho scelto la versione 17 di Java perché è una versione "stabile" (chiamata anche LTS, ovvero supportata per lungo tempo). Questo garantisce che il programma funzioni correttamente anche tra diversi anni senza bisogno di continue correzioni dovute agli aggiornamenti del linguaggio.
- **Apache Maven:** È uno strumento che serve a gestire automaticamente tutte le librerie esterne necessarie al progetto. Si occupa anche di "costruire" il programma finale, assicurandosi che tutte le parti (comprese le regole di ANTLR4) siano collegate correttamente tra loro.
- **Ambiente di sviluppo:** La fase di codifica e debugging è stata condotta in ambiente Eclipse. Grazie alla sua integrazione con Maven, mi ha permesso di tenere il progetto sempre organizzato.

3.3 Metodologia di sviluppo: Gestione del codice e Versioning con Git

Per non perdere traccia delle modifiche fatte, soprattutto durante le delicate modifiche alla grammatica, ho utilizzato Git per il controllo della versione. Git mi ha consentito di tracciare l'evoluzione del codice, permettendomi il ripristino di versioni precedenti in caso di errori.

Dal punto di vista operativo, il funzionamento del tool è stato progettato come un processo lineare suddiviso in tre fasi principali. Tutto inizia con la fase di lettura, in cui il file sorgente in *Structured Text* (`.st`) viene analizzato e convertito in un albero sintattico (AST). Una volta che il codice

è rappresentato sotto forma di albero, si passa alla fase di trasformazione: qui il motore di mutazione interviene per applicare le logiche di modifica che verranno descritte nel dettaglio nel prossimo capitolo.

Infine, il processo si conclude con la fase di emissione. In questo passaggio, il sistema prende l'albero modificato e lo "traduce" nuovamente in file di testo leggibili, rigenerando i sorgenti mutati. Per mantenere il lavoro ordinato e facilmente consultabile, il tool organizza automaticamente tutti i file prodotti all'interno di una directory dedicata (`target/mutants`), rendendo i mutanti pronti per essere utilizzati nelle suite di test.

3.4 Gestione delle Licenze

Un ultimo aspetto importante riguarda le regole di utilizzo del software (le licenze). Il parser originale che ho usato è distribuito con la licenza **MIT**, che è molto libera e permette a chiunque di modificare il codice.

Per il mio lavoro ho invece scelto la licenza **Apache 2.0**. È una licenza molto usata a livello professionale perché chiarisce bene i diritti di chi scrive il codice e di chi lo usa. Le due licenze sono compatibili tra loro: l'importante è aver citato l'autore originale del parser, cosa che ho fatto nei file del progetto. In questo modo, il tool può essere utilizzato e integrato in altri progetti senza dubbi o problemi legali.

Capitolo 4

Implementazione del Motore di Mutazione

4.1 Architettura del sistema: Integrazione del parser ANTLR nel motore Java

L'architettura del sistema è stata progettata per trasformare il codice *Structured Text* in un formato su cui si possa lavorare facilmente. Il cuore di questo meccanismo è la classe `StParserDemo`, che coordina le diverse parti del framework. Il suo compito principale è gestire l'integrazione del parser ANTLR4: questo strumento non si limita a leggere il testo, ma permette di scomporre il codice in una rappresentazione logica a nodi (AST) che possiamo modificare liberamente. Grazie a questa struttura, ogni operazione diventa un elemento indipendente che il sistema può individuare e variare senza dover riscrivere manualmente l'intero file. Il sistema segue quattro passaggi principali.

4.1.1 La fase di Lexing e Parsing

In questa fase iniziale, il file sorgente `.st` viene convertito da un semplice flusso di caratteri in una struttura gerarchica. Il processo inizia con la *Tokenizzazione*: il componente `IEC61131Lexer` scansiona il codice per isolare i *token*. Successivamente, il parser riceve questi *token* e verifica che la loro sequenza rispetti lo standard IEC 61131-3.[1] Il risultato finale di questo stadio è il *ParseTree*, una mappa molto dettagliata del file che però contiene

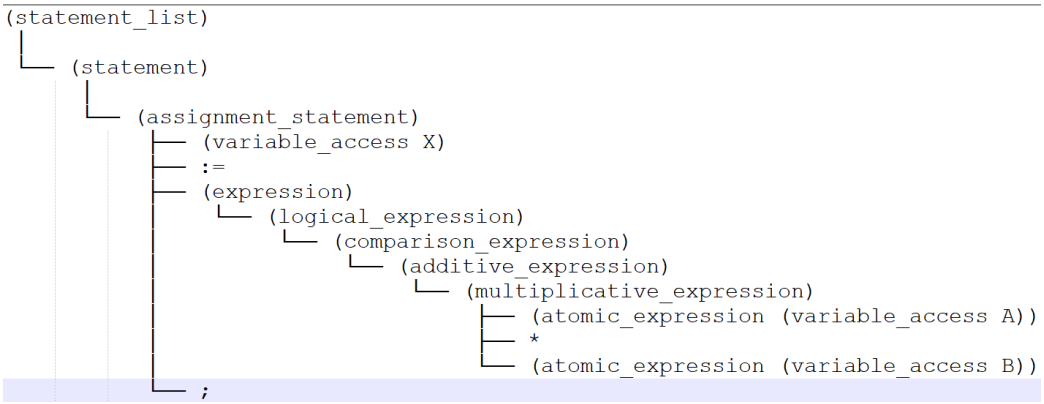
ancora elementi inutili per la mutazione, come i punti e virgola o le parentesi, e risulterebbe poco maneggevole per operazioni di modifica rapida.

4.1.2 Creazione dell'Abstract Syntax Tree (AST)

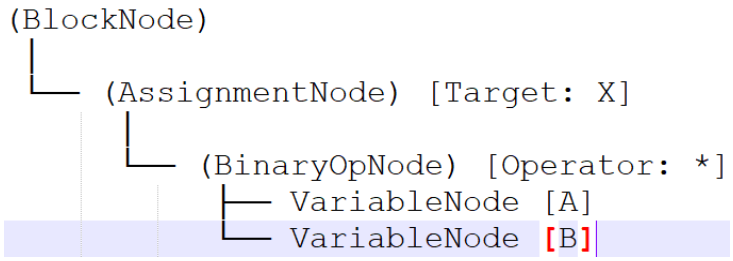
Per poter operare in modo efficiente, il sistema converte il *ParseTree* in un *Abstract Syntax Tree* (AST). Attraverso la classe `ASTBuilder`, viene eseguita una visita dell'albero sintattico per generare una struttura di nodi Java più intuitiva, come `BinaryOpNode` per le operazioni o `AssignmentNode` per gli assegnamenti.

La conversione da *Parse Tree* ad AST rappresenta un passaggio cruciale. La Figura 4.1 mostra in maniera efficace la differenza tra le due strutture: mentre il *Parse Tree* (generato da ANTLR) tiene traccia di ogni singolo dettaglio del testo, inclusa la punteggiatura e le parole chiave come `;`, `THEN` o `END_IF` necessarie solo al compilatore, l'AST estrae esclusivamente la logica del programma. In questo modo, il motore di mutazione smette di vedere il codice come una sequenza di parole e inizia a trattarlo come una gerarchia di operazioni logiche, rendendo la modifica degli operatori un'operazione immediata e sicura.

Riassumendo, questa rappresentazione offre tre vantaggi principali. In primo luogo, semplifica l'albero, eliminando le informazioni ridondanti del parsing e lasciando solo la logica pura. In secondo luogo, garantisce maggiore facilità nella modifica: trattando il codice come una serie di oggetti Java, possiamo cambiare un operatore (ad esempio da `+` a `-`) semplicemente modificando una proprietà dell'oggetto, senza dover riscrivere il testo a mano. Infine, fornisce un'identificazione univoca: ogni nodo dell'AST riceve un ID unico, permettendo al sistema di rintracciare esattamente lo stesso punto del codice anche dopo aver clonato l'albero per creare i diversi mutanti.



(a) Parse Tree



(b) Abstract Syntax Tree (AST)

Figura 4.1: Confronto strutturale tra il Parse Tree e l'AST per l'espressione: `X := A * B;`.

4.1.3 Identificazione e Filtraggio dei candidati

Una volta ottenuta la radice dell'albero (`rootAST`), il sistema avvia la logica di mutazione vera e propria, che avviene in due momenti. Per prima cosa, un metodo ricorsivo percorre l'intero albero e ogni volta che incontra un'operazione binaria la aggiunge a una lista di potenziali candidati alla mutazione. Il sistema è progettato per "scendere" in profondità anche dentro strutture complesse come cicli o rami condizionali, garantendo una copertura totale del codice sorgente.

In questa fase interviene la classe `MutationPlan`: per ogni punto individuato, viene creato un oggetto che memorizza l'ID del nodo, l'operatore originale e il suo sostituto. Questa lista di "piani" rappresenta l'insieme di tutti i mutanti potenziali.

Il secondo passaggio riguarda la Strategia di Selezione. Per evitare che il sistema generi un numero eccessivo di file, ho implementato un meccanismo di *Mutation Rate*, impostato di default al 50%, ma configurabile. Rendere il *Mutation Rate* configurabile permette di adattare il tool a progetti di diverse dimensioni. Nei programmi PLC industriali molto lunghi, generare ogni possibile mutante richiederebbe troppo tempo per eseguire i test. Grazie a questo parametro, l'utente può scegliere di generare solo una percentuale dei mutanti (ad esempio il 50%), trovando il giusto equilibrio tra la qualità del controllo e la velocità di esecuzione.

Il programma rimescola la lista dei candidati e ne seleziona solo una parte. Questa scelta si basa sulle conclusioni dello studio di Zhang et al.,[7] citato nel Capitolo 2, che evidenzia come una selezione casuale dei mutanti possa fornire risultati comparabili, in termini di efficacia, a strategie di selezione più complesse o complete. Questa filtrazione permette quindi di ridurre drasticamente i costi in termini di tempo e memoria, senza però compromettere la capacità del tool di valutare la robustezza della suite di test.

4.1.4 Generazione e Code Emission

L'ultimo stadio si occupa della creazione fisica dei mutanti di "Primo Ordine". Per ogni mutazione approvata, viene creata una copia esatta dell'albero originale; questo assicura che ogni file generato contenga una e una sola variazione, permettendo di isolare l'effetto di ogni singolo errore. A questo scopo, interviene il metodo `trovaNodoPerId`: partendo dalla radice del clone, il sistema ricerca ricorsivamente l'ID univoco assegnato in precedenza per individuare il punto esatto in cui fare la modifica. Una volta localizzato il punto nel clone, il sistema ne sostituisce l'operatore (ad esempio trasformando un `>` in un `<`).

Infine, la classe `CodeEmitter` percorre l'albero mutato e lo riconverte in una stringa di testo *Structured Text* valida. I file vengono salvati automaticamente nella directory "`target/mutants`" con nomi numerati progressivamente (come `mutant_1.st`, `mutant_2.st`, ecc.). Per garantire l'integrità della sperimentazione, il sistema azzera la cartella di output prima di ogni nuova sessione, evitando la sovrapposizione di mutanti appartenenti a versioni precedenti del codice. Questi file sono "pronti all'uso": possono essere inseriti direttamente nell'ambiente del PLC per verificare se i test aziendali sono in grado di identificarli.

4.2 Costruzione dell'AST tramite Visitor Pattern

Per muoversi all'interno del codice e trasformarlo, ho utilizzato una tecnica di programmazione chiamata **Visitor Design Pattern**, implementato nella classe `ASTBuilder`. Questo approccio permette di attraversare l'albero sintattico fornito da ANTLR4[8] in modo ordinato e di costruire, passo dopo passo, la mia rappresentazione personalizzata.

Dal punto di vista implementativo, il *Visitor Pattern* permette di separare la struttura dell'albero (fornita da ANTLR) dalla logica di costruzione del mio AST personalizzato. La classe `ASTBuilder`, infatti, estende `IEC61131BaseVisitor`, una classe generata automaticamente da ANTLR4 a partire dalla grammatica dello *Structured Text*. I metodi di "visita" (come ad esempio `visitIf_statement` o `visitAssignment_statement`) sono definiti in questa classe base e vengono poi sovrascritti per istruire il sistema su come trasformare ogni specifico elemento del *ParseTree* in un corrispondente nodo del mio *Abstract Syntax Tree*. Ogni volta che il *Visitor* incontra un nodo nel *ParseTree*, decide come trasformarlo: se il nodo rappresenta un'operazione, il *Visitor* estrae gli operandi e l'operatore, incapsulandoli in un nuovo oggetto Java.

Questo approccio rende il codice estremamente pulito: se in futuro volessi supportare una nuova funzione del linguaggio, mi basterebbe effettuare l'override del relativo metodo di visita fornito da ANTLR, senza dover modificare la logica di navigazione dell'albero o il motore di mutazione.

4.2.1 Strategia di Navigazione: Automatica vs Manuale

Per muoversi all'interno delle strutture dati, il framework utilizza due diverse modalità di "navigazione" dell'albero, distinte per finalità e precisione:

- **Navigazione Automatica (Visitor di ANTLR):** Viene utilizzata esclusivamente nella fase di costruzione dell'AST. In questo stadio, il sistema si affida al motore nativo di ANTLR che percorre ogni singola "foglia" del *Parse Tree* (incluse virgole e punti e virgola) per trasformarla in un oggetto Java. È una visita esaustiva, necessaria a non perdere alcun dettaglio sintattico del codice originale.

- **Navigazione Manuale Ricorsiva:** Una volta ottenuto l'AST personalizzato, per la fase di mutazione ho scelto di implementare algoritmi di ricerca manuale. Questa scelta è fondamentale per la sicurezza del codice PLC: mentre il *Visitor* automatico "vede tutto", la ricerca manuale è istruita per scendere solamente dove risiede la logica (ad esempio entrando nel corpo di un ciclo `FOR` per cercare calcoli), ignorando invece gli elementi strutturali che, se mutati, renderebbero il programma non compilabile.

In particolare, questa strategia si basa su due metodi:

- **Metodo raccogliPunti:** Scansiona l'intero albero per trovare e raccogliere tutti i nodi mutabili e popolare la lista dei candidati.

- **Metodo trovaNodoPerId:** Una volta selezionata una mutazione, questo metodo permette di rintracciare il nodo all'interno della copia

clonata dell'albero originale. Poiché ogni nodo possiede un ID univoco, la funzione può navigare ricorsivamente nel clone fino a trovare l'esatta corrispondenza, permettendo un'iniezione del difetto precisa e isolata.

4.2.2 Gestione della logica e delle precedenze

Uno degli aspetti più complessi dello *Structured Text* è rispettare le precedenze tra gli operatori (ad esempio, risolvere prima le moltiplicazioni e poi le somme). L'`ASTBuilder` affronta questa sfida attraverso metodi di visita specifici per ogni livello della gerarchia. Quando il sistema trova un'operazione tra due elementi, usa un metodo chiamato `buildBinaryOpNode` per raggrupparli correttamente. Se invece non trova operazioni, continua a scendere ricorsivamente verso il livello successivo. Questo garantisce che la struttura dell'albero rispetti fedelmente la logica matematica originale.

4.2.3 Controllo del flusso: IF e FOR

Il sistema non si ferma alle semplici operazioni matematiche, ma riconosce anche le strutture che controllano il flusso del programma. Nel caso del nodo `IF`, vengono estratti separatamente la condizione, il blocco di istruzioni del ramo "allora" (`THEN`) e quello del ramo "altrimenti" (`ELSE`). Questo è fondamentale perché permette al motore di mutazione di agire sia sulla condizione (cambiando ad esempio un `=` in `<>`), sia nei calcoli eseguiti all'interno. Anche per il nodo `FOR` viene ricostruita l'intera struttura, mappando la variabile di controllo e i limiti del ciclo. Avere queste informazioni separate è utile perché in futuro permetterà di aggiungere nuove tipologie di errori specifiche per i cicli, come la modifica del numero di volte che un'operazione viene ripetuta.

4.2.4 Normalizzazione e gestione delle parentesi

Una funzione importante dell'`ASTBuilder` è la capacità di mettere in ordine i dati che arrivano dal parser. Il metodo `buildBinaryOpNode` funziona come un "centro di smistamento": controlla se i pezzi che sta analizzando

sono variabili o numeri e si assicura che gli operatori estratti siano validi. Inoltre, un'attenzione particolare è stata data alle espressioni tra parentesi. Invece di eliminare le parentesi, il sistema le salva come parte della struttura. Questo assicura che, quando il codice verrà riscritto nel file finale, le parentesi originali siano ancora al loro posto, preservando l'integrità sintattica necessaria alla compilazione sul PLC.

4.3 Definizione degli Operatori di Mutazione (Aritmetici, Logici, Relazionali)

La classe `AstMutator` rappresenta il componente operativo che inietta gli errori nell'albero. Il suo compito è modificare i nodi senza compromettere la struttura del programma. Ho scelto di implementare tre classi di errori, che come abbiamo discusso nel Capitolo 2 (citando lo studio di Offutt[6]), sono considerati "sufficienti" per testare bene un sistema senza dover creare migliaia di varianti inutili.

4.3.1 Classificazione delle mutazioni

Le tre categorie di mutazione che il sistema supporta sono:

- **AOR (Arithmetic Operator Replacement):** Sostituisce gli operatori aritmetici scambiando tra loro addizione e sottrazione, o moltiplicazione e divisione. Questo serve a capire se i test rilevano errori nei calcoli matematici.
- **ROR (Relational Operator Replacement):** Modifica gli operatori di confronto (come `>`, `<`, `=`). È una delle mutazioni più importanti per un PLC, in quanto sbagliare un limite di sicurezza può causare danni meccanici.

- **LOR (Logical Operator Replacement):** Scambia i connettivi logici `AND` e `OR`, fondamentali per testare le condizioni di sicurezza che permettono alla macchina di muoversi.

L'implementazione di queste logiche di sostituzione è affidata al metodo `mutaOperatore`, illustrato nella Figura 4.2. Come si può osservare dal codice, il sistema utilizza una struttura `switch` per mappare ogni operatore originale verso il suo corrispettivo mutato, garantendo la coerenza tra le categorie AOR, ROR e LOR descritte in precedenza.

```
protected String mutaOperatore(String op) {
    return switch (op) {
        case "+" -> "-";
        case "-" -> "+";
        case "*" -> "/";
        case "/" -> "*";

        case ">" -> ">=";
        case "<" -> "<=";
        case ">=" -> "<";
        case "<=" -> ">";
        case "=" -> "<>";
        case "<>" -> "=";

        case "AND" -> "OR";
        case "OR" -> "AND";

        default -> op;
    };
}
```

Figura 4.2: Metodo mutaOperatore

4.3.2 Strategia di iniezione ricorsiva

Il mutatore adotta una strategia ricorsiva che gli permette di "entrare" in ogni livello del codice. Nel caso degli assegnamenti, il motore è istruito per colpire solo il lato destro della riga (dove risiede il calcolo) lasciando intatta la variabile di destinazione a sinistra. Questo evita di creare mutanti poco significativi. Nelle espressioni binarie complesse, il sistema non si limita a

mutare l'operatore principale, ma continua a cercare candidati anche all'interno degli operandi. Infine, il mutatore è capace di scendere all'interno dei blocchi di codice dei cicli e delle istruzioni `IF`, colpendo la logica decisionale del programma.

Questa precisione è garantita dal trattamento dei nodi "foglia": quando il mutatore incontra una variabile o un numero, si ferma. In questo modo si ha la certezza che il sistema non tenti di modificare nomi di variabili o costanti, operazione che rischierebbe di rendere il codice non compilabile.

4.4 Generazione dei mutanti di Primo Ordine (First-Order Mutants)

L'ultima fase consiste nella produzione dei file. In linea con la letteratura del *Mutation Testing* approfondita nel Capitolo 2,[2] dove si è stabilito che ogni variante deve contenere una singola modifica sintattica, il tool genera esclusivamente *First-Order Mutants* (**FOM**), ovvero programmi che differiscono dall'originale per una sola modifica.

Ho scelto di generare mutanti di Primo Ordine per due motivi principali. In primo luogo, tale scelta è coerente con la teoria dell'effetto di accoppiamento (*coupling effect*) descritta in letteratura,[2] secondo cui i difetti complessi possono essere considerati come combinazioni di difetti più semplici. Di conseguenza, una suite di test capace di individuare la maggior parte dei mutanti elementari ha un'elevata probabilità di individuare anche errori più articolati. In secondo luogo, questo approccio rende il tool molto più utile per il programmatore PLC: se un test fallisce, sappiamo esattamente quale operatore lo ha messo in crisi, evitando che più modifiche simultanee rendano difficile l'analisi del problema, come accadrebbe nel caso di *Higher-Order Mutants* (**HOM**).

Per ogni mutazione selezionata dopo la fase di filtrazione, il sistema esegue una clonazione integrale dell'AST originale. Lavorare su una copia "pulita" è stata una scelta fondamentale per assicurare che ogni file contenga un

unico errore isolato, evitando che le mutazioni si accumulino nello stesso file (trasformandolo in un mutante di ordine superiore). Una volta individuato il nodo tramite il suo ID univoco e applicata la modifica nel clone, interviene la classe `CodeEmitter`.

Questa classe si occupa di trasformare l'albero modificato in testo. Non si limita a scrivere una stringa continua, ma gestisce correttamente l'indentazione e le parole chiave di chiusura (come `END_IF;` o `END_FOR;`). Questo rende i file mutati non solo corretti per il compilatore del PLC, ma anche facilmente leggibili per uno sviluppatore umano. Ogni mutante viene salvato come file indipendente e compilabile, facilitando così il caricamento in un sistema di test aziendale, permettendo di identificare quali punti del software sono meno controllati e hanno bisogno di test più severi.

4.5 Analisi dei limiti del framework e gestione della complessità

A conclusione dell'analisi tecnica del sistema, è fondamentale definire con precisione il perimetro entro cui il framework opera. In questa fase di ricerca, la priorità è stata data alla costruzione di una base architettonica solida, capace di gestire con estrema affidabilità i costrutti fondamentali dello *Structured Text*, piuttosto che cercare una copertura totale ma potenzialmente instabile di ogni singola variante sintattica.

Allo stato attuale, il sistema è in grado di processare e mutare con successo i costrutti "base" del linguaggio, ovvero:

- Assegnazioni semplici e calcoli matematici lineari;
- Operatori relazionali e connettivi logici;
- Strutture di controllo del flusso (`IF-THEN-ELSE`);
- Cicli iterativi (`FOR`);
- Gestione delle precedenze tramite parentesi tonde.

Tuttavia, è emerso che aumentando la complessità sintattica del codice sorgente, ad esempio attraverso l'uso di strutture dati vettoriali (`array`), l'utilizzo di altri tipi di cicli oppure la concatenazione di calcoli arimetici con la medesima precedenza senza l'ausilio delle parentesi tonde, il parser può riscontrare difficoltà di interpretazione. Tali criticità non sono dovute a un limite del motore di mutazione in sé, ma alla necessità di una mappatura più ampia e precisa di ogni singola eccezione grammaticale dello standard IEC 61131-3 all'interno dell'`ASTBuilder`.

L'attuale versione del tool va quindi intesa come un motore di mutazione pienamente funzionante per la logica di base, la cui capacità di analisi può essere estesa attraverso l'implementazione di nuovi metodi di visita nel *Visitor Pattern*. Come verrà discusso nelle conclusioni e negli sviluppi futuri, l'integrazione di costrutti avanzati rappresenta il naturale passo successivo, essendo l'attuale struttura a nodi (AST) già predisposta per accogliere e gestire nodi sintattici di maggiore complessità.

Capitolo 5

Validazione e Analisi dei Risultati

Una volta completato lo sviluppo del framework, è stato necessario sottoporre il sistema a una fase di verifica. L'obiettivo non era solo accertare che il tool funzionasse, ma garantire che i mutanti generati fossero logicamente sensati e, soprattutto, utilizzabili in un contesto reale.

Questa fase di analisi mi ha permesso di testare il software passo dopo passo, partendo dai singoli componenti fino alla prova su programmi più complessi.

5.1 Test Unitari con JUnit: validazione del generatore

Prima di procedere alla generazione dei mutanti, è stato fondamentale validare singolarmente ogni componente del framework. A tale scopo, è stata implementata la classe `MutationLogicTest` utilizzando il framework **JUnit 5**.[10] Questa suite di test ha permesso di verificare la correttezza della trasformazione del codice in AST, l'efficacia delle mutazioni e la precisione della ricostruzione del testo originale.

5.1.1 Validazione del Parsing e dell'AST

Il primo punto su cui mi sono concentrato è stata la capacità del sistema di "leggere" e interpretare il codice *Structured Text*. Attraverso il metodo

helper `parseSnippet`, sono state passate stringhe di codice *Structured Text* per verificare la creazione dei nodi. Ho sottoposto al software alcuni esempi elementari, come semplici assegnamenti o brevi calcoli matematici. Era fondamentale assicurarsi che il programma riconoscesse correttamente ogni elemento.

Il test `testAssignmentParsing` conferma che una stringa come `Pressione := 100;` venga correttamente riconosciuta come un oggetto `AssignmentNode`, verificando che l'identificatore e il valore siano estratti correttamente. Con `testForLoopParsing`, il sistema doveva essere in grado di mappare l'intera istruzione garantendo che la struttura esterna del ciclo rimanesse intatta, focalizzando l'attenzione solo sulle operazioni interne da mutare.

5.1.2 Verifica della logica di Mutazione

Dopo aver accertato che la lettura del codice fosse precisa, ho testato l'efficacia delle mutazioni. Ho verificato che gli scambi tra gli operatori (come la trasformazione di un `+` in `-` o di un `AND` in `OR`) rispettassero le regole di sostituzione definite nel capitolo 4.

Per un PLC, la differenza tra "maggiore" e "maggiore o uguale" è minima a livello di codice ma enorme a livello di sicurezza; per questo ho verificato che il mutatore fosse estremamente preciso nel generare questi errori, attraverso il test `testComparisonMutation`. Inoltre ho verificato anche la correttezza delle mutazioni aritmetiche, attraverso il test `testArithmeticMutation`, e le mutazioni logiche, attraverso il test `testBooleanMutation`.

5.1.3 Test di Ciclo Completo e Ricorsione

L'aspetto più avanzato della validazione ha riguardato i test cosiddetti "end-to-end", ovvero prove che coprono l'intero percorso del codice, dall'analisi iniziale fino alla riscrittura finale. Questo passaggio è stato essenziale per confermare che l'intera catena di lavoro, composta dal `Parser`, dall'`ASTBuilder`, dal `Mutatore` e infine dal `CodeEmitter`, funzionasse come

un unico meccanismo sincronizzato, senza perdere informazioni durante i vari passaggi.

In primis, mi sono concentrato sulla validazione delle strutture condizionali per assicurarmi che il sistema non ne compromettesse la sintassi. Attraverso il test `testFullCycleIfStatement` ho potuto confermare che, dopo aver mutato una condizione, il generatore di codice fosse in grado di ricostruire l'intera struttura mantenendo correttamente le parole chiave `THEN` e `END_IF`, oltre agli assegnamenti interni. Questo garantisce che il mutante prodotto sia sintatticamente pronto per la compilazione.

Ancora più impegnativa è stata la verifica dell'iniezione di errori in strutture annidate, che rappresenta lo scenario più complesso e vicino alla realtà industriale. Nel test `testFullCycleForMutation` ho messo alla prova il sistema con un ciclo `FOR` che ospitava al suo interno un'istruzione `IF`, un test cruciale per dimostrare la capacità del mutatore di "scendere" ricorsivamente tra i vari livelli dell'albero. La prova ha confermato che il framework riesce ad ignorare i nodi che non deve modificare, come le variabili di controllo del ciclo, per andare a colpire con precisione solo l'operatore relazionale annidato in profondità. Questo risultato assicura che il framework mantenga intatta l'architettura esterna del programma e le relative indentazioni, preservando l'integrità del codice anche di fronte a logiche di controllo stratificate e complesse.

La Figura 5.1 illustra l'esito della suite di test JUnit 5 al termine dell'esecuzione: il superamento completo (7/7) di tutti i test attesta la solidità del sistema e la sua affidabilità nell'operare su scenari di test realistici.

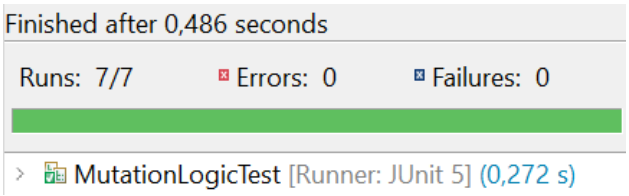


Figura 5.1: Esito dell'esecuzione della suite di test JUnit 5.

5.2 Sperimentazione pratica: generazione di mutanti da esempi ST

In questa fase di sperimentazione, il framework è stato testato su esempi di codice *Structured Text* per valutare il comportamento del sistema. L'obiettivo è osservare la capacità del tool di mappare un numero elevato di mutanti potenziali e l'efficacia del filtraggio casuale.

Per fornire una validazione completa, la sperimentazione è stata divisa in tre casi di studio con obiettivi differenti:

1. **Validazione Funzionale**: volta a dimostrare la capacità del framework di individuare e mutare operatori critici in una logica di controllo completa e strutturata.

2. **Validazione Strutturale**: focalizzata sulla tenuta sintattica del sistema in presenza di cicli iterativi.

3. **Validazione Tecnica**: uno "stress test" sulla profondità dell'Abstract Syntax Tree e sulla gestione della ricorsione.

5.2.1 Caso di Studio 1: Analisi multi-operatore e gestione dei blocchi condizionali

Il primo scenario di prova è stato progettato con un approccio prettamente funzionale. Ho utilizzato un esempio di codice che simula una logica industriale standard: la lettura di un segnale analogico, la sua conversione in unità ingegneristiche (*Bar*) e la successiva attivazione di una valvola di scarico in caso di sovrappressione o emergenza. Il codice presenta un insieme bilanciato di operatori: aritmetici per la taratura, relazionali per le soglie e logici per le condizioni di sicurezza.

Sottoponendo il file al sistema, l'`ASTBuilder` ha individuato correttamente cinque candidati ideali per la mutazione. Nello specifico, la struttura del codice ha permesso di rilevare un operatore logico, due operatori relazionali

e due operatori aritmetici. Questo scenario è stato ideale per testare con precisione la flessibilità del *Mutation Rate*.

Effettuando una generazione con rate al 100%, il sistema ha prodotto l'intero set di cinque mutanti distinti, ognuno con una singola modifica isolata. Successivamente, per simulare una necessità di ottimizzazione dei tempi di test, ho impostato il limite al 50%. In questa configurazione, ho potuto verificare come il programma fosse in grado di rimescolare l'elenco dei candidati e selezionarne casualmente solo due (applicando l'arrotondamento per difetto). Questa funzione risulta estremamente utile nella pratica industriale, poiché permette allo sviluppatore di decidere il volume di mutanti da generare, riducendo i carichi di lavoro senza però sacrificare la varietà statistica degli errori iniettati.

Un esempio concreto di questo output è visibile nella Figura 5.2, in cui viene mostrato il confronto tra il codice originale e il codice mutato. In questo caso, il sistema ha agito sulla soglia di allarme, trasformando l'operatore "maggiore" (`>`) in un "maggiore o uguale" (`>=`). Sebbene la variazione a livello testuale sia minima, l'integrità del file (indentazione e sintassi) è rimasta corretta, garantendo la totale compilabilità del codice. Si tratta di una mutazione particolarmente insidiosa poiché crea un "errore di frontiera", costringendo la suite di test aziendale a verificare il comportamento del sistema esattamente sul limite della soglia di sicurezza, un punto critico dove spesso i test meno accurati tendono a fallire.

```
PressioneBar := (SegnalePressione * 0.01) + Offset;

IF (PressioneBar > SogliaMassima) OR (Emergenza = TRUE) THEN
    ValvolaScarico := TRUE;
    PompaMarcia := FALSE;
ELSE
    ValvolaScarico := FALSE;
    PompaMarcia := TRUE;
END_IF;
```

(a) Codice originale

```
PressioneBar := (SegnalePressione * 0.01) + Offset;
IF (PressioneBar >= SogliaMassima) OR (Emergenza = TRUE) THEN
    ValvolaScarico := TRUE;
    PompaMarcia := FALSE;
ELSE
    ValvolaScarico := FALSE;
    PompaMarcia := TRUE;
END_IF;
```

(b) Codice mutato

Figura 5.2: Esempio di applicazione di un operatore di mutazione: trasformazione dell'operatore di confronto `>` in `>=`.

5.2.2 Caso di Studio 2: Integrità strutturale in cicli iterativi

A differenza del primo caso, il secondo esperimento si sposta su un piano di validazione strutturale. Il codice è progettato per accumulare dieci campioni consecutivi di un segnale analogico (`ValoreIngresso`) all'interno di un ciclo `FOR`, calcolandone la media finale. L'obiettivo principale di questo test è stato valutare la stabilità del framework di fronte a strutture iterative, verificando che la manipolazione degli operatori non compromettesse la ciclicità del codice.

Sottoponendo il codice all'analisi, il sistema ha mappato correttamente i punti da mutare sia all'interno del corpo del ciclo `FOR` (l'operatore aritmetico `+`), sia nella fase conclusiva di calcolo (l'operatore di divisione `/`). In questo scenario, il mutatore ha dimostrato di saper distinguere con precisione tra gli

operatori aritmetici mutabili e le variabili di controllo del ciclo (come l'indice `i` o i limiti del range), che devono invece rimanere invariate per non generare errori di runtime o loop infiniti.

L'analisi dei mutanti generati ha confermato la robustezza del `CodeEmitter`. Un esempio pratico di questa trasformazione è riportato in Figura 5.3, dove la somma algebrica all'interno del ciclo `FOR` è stata trasformata in una sottrazione: un errore logico classico che porterebbe a risultati inconsistenti ma che non impedisce l'esecuzione del codice. La conservazione della parola chiave `END_FOR` e della sintassi del ciclo garantisce che il mutante sia un candidato perfetto per testare la capacità di rilevamento di errori all'interno di loop industriali.

```
Somma := 0.0;
Media := 0.0;
ValoreIngresso := 15.5;

FOR i := 0 TO 9 DO
    Somma := Somma + ValoreIngresso;
END_FOR;

Media := Somma / 10.0;
```

(a) Codice originale

```
Somma := 0.0;
Media := 0.0;
ValoreIngresso := 15.5;
FOR i := 0 TO 9 DO
    Somma := Somma - ValoreIngresso;
END_FOR;
Media := Somma / 10.0;
```

(b) Codice mutato

Figura 5.3: Esempio di applicazione di un operatore di mutazione: trasformazione dell'operatore aritmetico `+` in `-`.

5.2.3 Caso di Studio 3: Navigazione ricorsiva in strutture annidate

Per spingere il framework al limite della complessità sintattica, ho predisposto un terzo scenario. In questo caso il codice, illustrato nella figura 5.4, presenta un'istruzione `IF` nidificata all'interno di un ciclo `FOR`, una configurazione comune nei sistemi di filtraggio dati in tempo reale. Questo scenario funge da "stress test" per la navigazione ricorsiva dell'*Abstract Syntax Tree*.

Il sistema ha dovuto gestire una gerarchia di nodi complessa: è stato necessario "scendere" all'interno del nodo del ciclo, identificare il blocco con-

dizionale e, solo a quel punto, mappare gli operatori relazionali e aritmetici presenti al secondo livello di profondità. La sfida tecnica principale è rappresentata dalla ricostruzione del codice: il tool deve essere in grado di chiudere i blocchi nell'ordine esatto (`END_IF` seguito da `END_FOR`), mantenendo la corretta struttura nidificata.

I risultati di questa mutazione hanno dimostrato che il framework non effettua una scansione lineare del testo, ma ne comprende la reale struttura gerarchica. I mutanti prodotti, illustrati nella figura 5.5, pur non avendo una perfetta indentazione, mantengono una coerenza strutturale assoluta. Questo risultato è fondamentale per l'applicazione del *Mutation Testing* su software PLC reali, dove la logica di controllo è raramente piatta e spesso richiede più livelli di nidificazione per gestire correttamente i processi fisici.

```
Somma := 0.0;
Soglia := 10.0;
ValoreIngresso := 12.5;

FOR i := 1 TO 5 DO
    IF ValoreIngresso > Soglia THEN
        Somma := Somma + ValoreIngresso;
    END_IF;
END_FOR;

Risultato := Somma;
```

Figura 5.4: Codice originale

```
Somma := 0.0;
Soglia := 10.0;
ValoreIngresso := 12.5;
FOR i := 1 TO 5 DO
    IF ValoreIngresso >= Soglia THEN
    Somma := Somma + ValoreIngresso;
END_IF;
END_FOR;
Risultato := Somma;
```

(a) mutante 1

```
Somma := 0.0;
Soglia := 10.0;
ValoreIngresso := 12.5;
FOR i := 1 TO 5 DO
    IF ValoreIngresso > Soglia THEN
    Somma := Somma - ValoreIngresso;
END_IF;
END_FOR;
Risultato := Somma;
```

(b) mutante 2

Figura 5.5: Esempio di applicazione di due operatori di mutazione: trasformazione dell'operatore relazionale `>` in `>=` e trasformazione dell'operatore aritmetico `+` in `-`.

5.3 Analisi dei risultati e Valutazione dell'efficacia

Dopo aver completato le sessioni di test sui tre casi di studio, è stata effettuata un'analisi per valutare le prestazioni complessive del framework. La valutazione si è concentrata sulla fedeltà sintattica del codice rigenerato e sull'impatto logico che le diverse tipologie di mutazione hanno sulle logiche *Structured Text*.

5.3.1 Integrità del codice e Compilabilità

Il primo dato rilevante emerso dalla sperimentazione riguarda l'affidabilità del `CodeEmitter`. In tutti i mutanti generati non è stato riscontrato alcun errore di sintassi, indipendentemente dalla complessità dello script di partenza.

Questo risultato conferma la superiorità dell'approccio basato su *Abstract Syntax Tree* rispetto alla manipolazione testuale diretta. Mentre un semplice sistema "cerca e sostituisci" rischierebbe di corrompere la punteggiatura o le parole chiave (come `;`, `:=` o le chiusure dei blocchi), il framework sviluppato opera sui nodi logici. Nel Caso 1, la gestione degli spazi nelle istruzioni `IF-THEN-ELSE` è rimasta impeccabile, mentre nei Casi 2 e 3 l'integrità dei blocchi iterativi è stata preservata integralmente. Garantire che ogni mutante sia immediatamente compilabile è un requisito fondamentale in ambito industriale, poiché evita interruzioni nel flusso di test automatico.

5.3.2 Analisi qualitativa dell'impatto delle mutazioni

L'analisi ha evidenziato come le diverse classi di operatori mutati (**AOR**, **ROR**, **LOR**) producano effetti profondamente diversi sul comportamento del PLC.

Le mutazioni di tipo relazionale (**ROR**) si sono rivelate le più critiche e insidiose. Trasformare un confronto da "maggiore" a "maggiore o ugua-

le" (come visto nel Caso 1) introduce un errore che non rompe la logica del programma, ma ne sposta la soglia di intervento. Questi "errori di frontiera" sono estremamente difficili da individuare con test funzionali generici e richiedono suite di test progettate per validare i valori limite.

Al contrario, le mutazioni aritmetiche (**AOR**) colpiscono direttamente il calcolo del dato. Nel Caso 2, la trasformazione di una somma in una sottrazione all'interno del calcolo della media produce un errore macroscopico. Sebbene più facile da rilevare, questa tipologia di mutazione è essenziale per verificare che i sistemi di monitoraggio del PLC siano in grado di intercettare dati palesemente incoerenti. Infine, le mutazioni logiche (**LOR**) agiscono sulla struttura decisionale stessa: cambiare un `OR` in un `AND` può alterare in modo significativo il comportamento delle condizioni di sicurezza, mettendo alla prova l'efficacia dei meccanismi di protezione implementati nel programma.

5.3.3 Ottimizzazione tramite Mutation Rate e Considerazioni sulla selezione

Un punto centrale della sperimentazione ha riguardato l'efficacia del *Mutation Rate* come strumento di ottimizzazione. Impostare un rate ridotto (ad esempio al 50%) non significa semplicemente dimezzare il numero di test, ma limitare l'applicazione delle mutazioni a una percentuale degli operatori individuati, selezionati casualmente. Questo consente di ridurre il carico computazionale, mantenendo comunque una distribuzione rappresentativa delle tipologie di errore.

Il sistema rimescola l'elenco dei candidati prima della selezione. Questo approccio garantisce una distribuzione equa delle mutazioni lungo tutto il codice, evitando che il processo si concentri solo sulle prime righe. In contesti industriali, dove i programmi PLC possono contare migliaia di istruzioni, questa capacità di "dosare" i mutanti è essenziale per rendere sostenibili i tempi di esecuzione delle campagne di test.

Tuttavia, è importante sottolineare una caratteristica intrinseca della selezione casuale: il sistema non possiede una consapevolezza della criticità degli operatori. Sebbene la selezione casuale garantisca una varietà statistica, in una configurazione a rate ridotto esiste la possibilità che alcuni operatori particolarmente critici per la sicurezza (come i connettori logici di emergenza) non vengano selezionati a favore di mutazioni aritmetiche meno impattanti.

Questa osservazione apre la strada a future evoluzioni del framework, come l'implementazione di algoritmi di selezione "pesati", in cui lo sviluppatore potrebbe assegnare una priorità maggiore a determinate classi di operatori rispetto ad altri. Nonostante questo limite teorico, la sperimentazione ha confermato che il *Mutation Rate* rimane uno strumento indispensabile per la scalabilità del sistema, permettendo di calibrare lo sforzo di validazione in base alle risorse computazionali disponibili senza rinunciare a un'analisi strutturata dell'intero codice.

Nel complesso, l'analisi condotta sui casi di studio ha evidenziato la coerenza tra l'architettura progettata e il comportamento osservato in fase sperimentale. I risultati ottenuti costituiscono quindi una base solida per le considerazioni conclusive sul valore e sui possibili sviluppi del framework.

Conclusioni e Sviluppi Futuri

Il presente lavoro di tesi ha portato alla progettazione e alla realizzazione di un framework automatizzato per la generazione di mutanti applicato al linguaggio *Structured Text* (ST). L'obiettivo centrale è stato rispondere a una sfida estremamente attuale: l'integrazione dei modelli linguistici di grandi dimensioni (LLM) nei processi di automazione industriale, fornendo uno strumento capace di misurare oggettivamente l'affidabilità delle suite di test quando il codice viene generato automaticamente.

L'adozione di una strategia basata sulla manipolazione dell'*Abstract Syntax Tree* (AST) tramite il parser ANTLR4 si è rivelata la scelta vincente. Rispetto ai sistemi di mutazione testuale "grezza", questo approccio ha permesso al framework di "comprendere" realmente la gerarchia del codice. Come analizzato nel Capitolo 4 e validato sperimentalmente nel Capitolo 5, la capacità di iniettare errori logici (**AOR**, **ROR**, **LOR**) mantenendo la totale integrità sintattica ha eliminato il problema dei "mutanti nati morti", ovvero codici non compilabili che avrebbero solo rallentato il processo di validazione.

La sperimentazione condotta sui tre casi di studio ha confermato che il sistema è in grado di gestire scenari industriali reali, dalle logiche condizionali ai cicli iterativi più complessi. Inoltre, l'introduzione del *Mutation Rate* ha dimostrato che è possibile rendere il testing sostenibile: ridurre il carico computazionale non significa sacrificare la qualità, ma operare una selezione statistica controllata che rende il framework scalabile e pronto per l'applicazione su larga scala.

Sviluppi Futuri

Nonostante la solidità del framework realizzato, questo lavoro rappresenta un punto di partenza che apre la strada a diverse evoluzioni:

- **Completamento del supporto Structured Text:** Sebbene i nuclei fondamentali del linguaggio siano stati implementati, il primo passo futuro consiste nell'estendere il supporto a tutte le strutture sintattiche dello standard IEC 61131-3 non ancora coperte. L'obiettivo è rendere il tool capace di processare qualsiasi blocco di codice ST, garantendo una copertura universale del linguaggio.

- **Integrazione negli ambienti di sviluppo:** Un'evoluzione naturale del progetto riguarda lo sviluppo di interfacce che permettano di integrare il motore di mutazione direttamente negli IDE industriali. Questo faciliterebbe l'adozione quotidiana del *Mutation Score* da parte degli sviluppatori PLC, rendendo il testing parte integrante del workflow di programmazione.

- **Selezione Intelligente delle Mutazioni:** È auspicabile l'evoluzione dell'attuale algoritmo di selezione casuale verso un sistema "pesato".Un sistema capace di identificare con priorità le sezioni di codice più critiche per la sicurezza o quelle che, statisticamente, risultano più soggette a errori logici da parte degli LLM, ottimizzando ulteriormente lo sforzo di validazione.

In conclusione, il framework sviluppato rappresenta un contributo concreto al miglioramento dei processi di validazione del software PLC. L'approccio basato sull'analisi sintattica e sulla generazione controllata di mutanti dimostra come sia possibile introdurre metriche quantitative anche in contesti industriali tradizionalmente legati a pratiche di verifica manuale.

Questo lavoro pone quindi le basi per un'evoluzione dei processi di testing verso modelli più strutturati e integrabili con le nuove tecnologie di generazione automatica del codice.

Bibliografia

[1] IEC (International Electrotechnical Commission). (2013). *IEC 61131-3:2013: Programmable controllers - Part 3: Programming languages*. Geneva: IEC.

[2] Jia, Y., & Harman, M. (2011). *An analysis and survey of the development of mutation testing*. IEEE Transactions on Software Engineering, 37(5), 649-678.

[3] Chen, M., Tworek, J., Jun, H., Yuan, F., Pinto, H. P. d. O., Kaplan, J., ... & Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv preprint arXiv:2107.03374.

[4] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, X., Li, L., ... & Lo, D. (2023). *Large language models for software engineering: A systematic literature review*. arXiv preprint arXiv:2308.10620.

[5] DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). *Hints on test data selection: Help for the practicing programmer*. IEEE Computer, 11(4), 34-41.

[6] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., & Zapf, C. (1996). *An experimental determination of sufficient mutant operators*. ACM Transactions on Software Engineering and Methodology (TOSEM), 5(2), 99-118.

[7] Zhang, L., Hou, S. S., Hu, J. J., Xie, T., & Mei, H. (2010). *Is operator-based mutant selection superior to random mutant selection?*. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (Vol. 1, pp. 435-444).

[8] ANTLR Org. *ANTLR 4 Documentation and Features*. Disponibile all'indirizzo: `https://www.antlr.org/`

[9] Sitnikov, V. *iec61131-parser*. Repository GitHub. Disponibile su: `https://github.com/vlsi/iec61131-parser`

[10] JUnit Team. (2023). *JUnit 5 User Guide (v. 5.10.0)*. Disponibile all'indirizzo: `https://docs.junit.org/5.10.0/user-guide/index.html`

Ringraziamenti

In primis, vorrei ringraziare la mia famiglia. Grazie per il vostro amore incondizionato, per i sacrifici fatti e per aver creduto in me in ogni momento, spingendomi sempre a dare il meglio. Grazie per avermi permesso di fare sempre quello che mi piaceva ma soprattutto di aver sempre sopportato le mie lamentele e i miei attacchi d'ira. Vi voglio bene.

Grazie agli amici di una vita, Zac, Berto e Tino. Ci conosciamo da tantissimo tempo ormai eppure continuiamo a condividere bellissimi momenti ed esperienze. Spero continueremo per sempre, riuscite sempre a strapparmi un sorriso e farmi sentire bene.

Grazie agli amici del "cinema", Mauro, Irene, Coli e Burja, per aver reso indimenticabili le nostre serate e per il vostro sostegno, che non è mai mancato. So di poter contare sempre su di voi.

Grazie ai miei compagni di squadra del Granarolo, con cui gioco ormai da 4 anni. Grazie per farmi sentire parte del gruppo, per tutte le vittorie e le risate fatte insieme, riuscendomi sempre a distrarre dallo stress.

Devo fare un ringraziamento particolare alla persona con la quale ho vissuto questi tre anni e mezzo di ansie, viaggi, risate e delusioni: Dejvi. Sono stato fortunato, dopo aver trascorso 4 anni di superiori insieme, a ritrovarti anche qui. Sapevo di non essere mai da solo: nei viaggi in treno, in classe, a pranzo dal nostro piadinaro di fiducia, in biblioteca a studiare e a lamentarci di qualsiasi cosa su Whatsapp. Purtroppo non ci laureeremo insieme come volevamo, ma so che ci sarai comunque e spero che tu sappia che io, quando toccherà a te, ci sarò.

Ringrazio il mio relatore, il Prof. Davide Rossi, per la sua disponibilità e il suo supporto dimostrato lungo tutto il periodo della tesi. Ringrazio anche il Dott. Stefano Sinigardi, per avermi dato l'opportunità di visitare Coesia, dandomi l'opportunità di vedere da vicino come si lavora in un contesto

industriale reale.

Grazie, infine, a tutti coloro che, direttamente o indirettamente, hanno lasciato un segno nel mio percorso, rendendo questi anni un'esperienza di vita unica.