# Realizzazione di un ambiente di Fault Injection per applicazione ridondata [Manuale di Sviluppo]

Carlo Migliaccio[1], Federico Pretini[1], Alessandro Scavone[1], and Mattia Viglino[1]

[1]Laurea Magistrale in Ingegneria Informatica, Politecnico di Torino

Gennaio 2025

**Abstract**

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

## Contents

# 1 Introduzione

Nella società attuale si fa, in generale, un utilizzo capillare di sistemi computerizzati, questi nello specifico sono coinvolti in settori in cui un guasto al sistema potrebbe essere critico, mettendo potenzialmente a rischio vite umane. L'implementazione di sistemi *safety-critical* espone lo sviluppatore ad affrontare problemi non trascurabili che coinvolgono la valutazione della **dependability** [3] e della **tolleranza ai guasti** (*fault tolerance*). Le tecniche di test standard e l'utilizzo di benchmark non bastano in quest'ambito, in quanto per valutare certi aspetti del sistema (quali la dependability) bisognerebbe osservarne il comportamento dopo che il guasto si verifica.
In ambito 'fault-tolerance' vengono utilizzate delle metriche specifiche per quantificare affidabilità e robustezza del sistema in analisi. Per citarne una significativa, riportiamo l'*MTBF* (Mean Time Between Failure); questa per i sistemi concepiti, per essere tolleranti ai guasti, potrebbe essere associata ad un periodo di tempo molto lungo, anche anni! Da tempo la ricerca va nella direzione di trovare un modo per 'accelerare' in simulazione l'occorrenza di questi guasti/difetti prima che accadano naturalmente, dal momento che questa lunga latenza rende difficile anche solo identificare la causa di un potenziale difetto/guasto.
In molti casi l'approccio utilizzato è quello basato su esperimenti di **fault injection**, che – come riportato in [5] – vengono usati non solo durante l'implementazione, ma anche durante la fase prototipale e operativa, permettendo quindi di coprire un ampio spettro di casistiche.
La survey [4] individua principalmente due grandi famiglie di *tecniche di fault injection*: (i) Fault Injection Hardware di cui non ci occupiamo; (ii) Fault Injection Software, è la famgilia di tecniche che negli ultimi anni ha attirato l'interesse dei ricercatori in quanto tali metodi non richiedono hardware costoso. Inoltre nei contesti in cui il target sia un *applicativo* o, ancora peggio, il *sistema operativo*, costituiscono l'unica scelta.
Nel lavoro qui presentato si adotta un approccio *software* che si pone principalmente **due obiettivi**:

1. La modifica del codice per *casi di studio scelti* volta ad introdurre ridondanza nel sistema tramite la **duplicazione di tutte le variabili**;
2. La realizzazione di un **ambiente software di fault injection** per simulare l'occorrenza di guasti nel sistema irrobustito e valutarne l'*affidabilità*. Il **modello di fault** analizzato è il *transient single bit-flip fault*, su cui si basano molti tool di fault injection.

Il presente documento si pone l'obiettivo di evidenziare i passaggi salienti dell'implementazione delle tecniche di *Software fault tolerance* e dell'*ambiente di fault injection* riportando schemi e *code snippet* che insieme al codice sorgente – in **linguaggio Rust** – permettono di seguire le diverse fasi del lavoro svolto.
Nella **Sezioni 2-3** viene introdotto e successivamente implementato un **set di trasformazioni del codice** sfruttando parte dei concetti presentati in [7]. Si anticipa che l'implementazione di queste regole è integrata nella realizzazione di un nuovo tipo generico(`Hardened<T>`) di cui si descrivono gli aspetti chiave. La **Sezione 4** fornisce un'analisi comparativa del codice non irrobustito rispetto a quello che utilizza le variabili di tipo generico `Hardened<T>`. Nella **Sezione 5** viene definita la struttura dell'ambiente di fault injection, mentre le **Sezioni 7, 8, 9** si occupano di analizzarne i componenti. Infine la **Sezione 11** descrive brevemente la struttura del codice sorgente e dei test d'unità.

# Part I
# Irrobustimento del codice

# 2 Software fault-tolerance

In generale le tecniche di software fault-tolerance, tendono a sfruttare modifiche di alto livello al codice sorgente in modo da poter rilevare comportamenti irregolari (faults) che riguardano **sia il codice che i dati**. Qui invece *poniamo l'attenzione esclusivamente su fault che riguardano i dati*, senza peraltro preoccuparci del fatto che questi si trovino in memoria centrale, memoria cache, registri o bus. Al codice target infatti vengono applicate semplici trasformazioni di alto livello che sono completamente indipendenti dal processore che esegue il programma.

## 2.1 Tre regole per la trasformazione del codice

Le regole di trasformazione del codice citate sono quelle proposte in [6]. Riportiamo qui quelle mirate al rilevamento di **errori sui dati**:

1. **Regola #1**: Ogni variabile `x` deve essere duplicata: siano `cp1` e `cp2` i nomi delle due copie;
2. **Regola #2**: Ogni operazione di scrittura su `x` deve essere eseguita su entrambe le copie `cp1` e `cp2`;
3. **Regola #3**: dopo ogni operazione di lettura su $x$, deve essere controllata la consistenza delle copie `cp1` e `cp2`, nel caso in cui tale controllo fallisca deve essere sollevato un errore.

Anche i parametri passati a una procedura, così come i valori di ritorno, sono variabili come tutte le altre a cui si applicano le stesse trasformazioni. L'implementazione di queste caratteristiche – come spiegato in dettaglio nel paragrafo successivo – si basano sulla programmazione generica e polimorfismo offerti dal linguaggio Rust. Dopo una prima analisi si descrivono le principali caratteristiche e i metodi offerti dal nuovo tipo, in un secondo momento si entra nel dettaglio del linguaggio e si pone l'attenzione all'implementazione della semantica richiesta da **(R1)-(R3)**.

## 2.2 Il tipo `Hardened<T>`

Le tre regole di trasformazione appena esposte sono espletate tramite l'implementazione di un **nuovo tipo**, che chiamiamo `Hardened<T>`, definito come segue:

```
#[derive(Clone, Copy)]
struct Hardened<T>{
    cp1: T,
    cp2: T
}
```

Poiché si vuole porre l'attenzione sul comportamento/algoritmo di questo nuovo tipo, si usa la programmazione generica infatti le due copie `cp1` e `cp2` hanno un tipo generico `T` a cui viene posto l'unico vincolo di essere confrontabile e copiabile. Con l'obiettivo di coprire il maggior numero di casistiche possibili in cui il dato viene acceduto in lettura e/o scrittura, sono stati implementati per `Hardened<T>` un numero significativo di **tratti della libreria standard**, in particolare:

- `From<T>`: per ricavare una variabile ridondata a partire da una variabile 'semplice' di tipo T;
- I tratti per le **operazioni aritmetiche** `Add, Sub, Mul`. In particolare i primi sono stati implementati anche in *versione mista* `Add<usize>` e `Sub<usize>` per semplificare le operazioni di sottrazione tra un `Hardened<T>` e un valore *literal*;
- I tratti per le **operazioni di confronto** `Eq, PartialEq, Ord, PartialOrd`;
- I tratti `Index` e `IndexMut` sotto diverse forme per accedere alla singola copia della variabile (utile in fase di iniezione) e all'elemento i-esimo di una *collezione* di `Hardened<T>`.
- Il tratto `Debug` per la visualizzazione personalizzata di informazioni sul nuovo tipo di dato.

Oltre ai tratti della libreria standard appena elencati, si è rivelata utile l'implementazione di funzioni personalizzate di cui si riporta una breve descrizione.

```
impl<T> Hardened<T>{
    fn incoherent(&self)->bool;
    pub fn assign(&mut self, other: Hardened<T>)->Result<(), IncoherenceError>;
    pub fn from_vec(vet: Vec<T>)->Vec<Hardened<T>>;
    pub fn from_mat(mat: Vec<Vec<T>>) -> Vec<Vec<Hardened<T>>>;
    pub fn inner(&self)->Result<T, IncoherenceError>;
}
```

`fn incoherent(&self)->bool` Funzione privata per rilevare l'incoerenza tra le due copie della variabile irrobustita: in particolare viene utilizzata dai metodi di più alto livello che lavorano con i dati elementari.

`pub fn assign(&mut self, other: Hardened<T>)->Result<(), IncoherenceError>;` Asserisce all'**assegnazione** tra due variabili di tipo `Hardened<T>`. Questa è l'unica operazione che non si può ridefinire in Rust tramite l'implementazione del tratto opportuno, in quanto andrebbe ridefinita l'intera semantica legata al **movimento e possesso**.

`pub fn from_vec(vet: Vec<T>)->Vec<Hardened<T>>;` Per estrarre collezioni di dati irrobustiti da collezioni di dati elementari. Un ruolo simile è svolto da
`pub fn from_mat(mat: Vec<Vec<T>>) -> Vec<Vec<Hardened<T>>>;` Queste funzioni sono indispensabili sia per l'implementazione che per l'analisi dei risultati dei casi di studio.

`pub fn inner(&self)->Result<T, IncoherenceError>;` esegue una sorta di *unwrap* del dato irrobustito, cioè dato un `Hardened<T>` restituisce il dato `T` incapsulato a sua volta in un `Result` in quanto le copie memorizzate possono essere incoerenti (vedi paragrafo dopo).

# 3 Regole di trasformazione: implementazione

In questo paragrafo, tramite l'utilizzo di esempi significativi si presenta a grandi linee l'implementazione del set di trasformazioni che portano al tipo irrobustito. In particolare, dopo aver richiamato la regola, segue un esempio di codice con l'implementazione.

**R1**: ogni variabile `x` **deve essere duplicata**: siano `cp1` e `cp2` i nomi delle due copie

La realizzazione della prima regola è insita nella definizione del nuovo tipo, in quanto una dichiarazione l'inizializzazione di una variabile a partire da un dato elementare, crea una doppia copia del dato stesso. Si veda il seguente esempio:

```
let mut myvar=15;
let mut hard_myvar = Hardened::from(myvar);
```

Tramite il metodo `from()` del tratto `From` infatti vengono popolati i campi `cp1` e `cp2` della nuova variabile `hard_myvar` nel modo seguente:

```
impl<T> From<T> for Hardened<T> where T:Copy{
    fn from(value: T) -> Self {
        // Regola 1: duplicazione delle variabili
        Self{cp1: value, cp2: value}
    }
}
```

**R2**: ogni **operazione di scrittura** su `x` deve essere eseguita su entrambe le copie `cp1` e `cp2`

Come esempio significativo si consideri il frammento di codice dell'operazione di `assign()`:

```
pub fn assign(&mut self, other: Hardened<T>)->Result<(), IncoherenceError>{
    //                  [... ]

    //Regola 2: Ogni scrittura deve essere eseguita su entrambe le copie
    self.cp1 = other.cp1;
    self.cp2 = other.cp2;
    Ok(())
}
```

Dopo un controllo di coerenza della variabile da assegnare (paragrafo successivo), si scrive sia su una copia che sull'altra.

**R3**: dopo ogni **operazione di lettura** su $x$, deve essere controllata la consistenza delle copie `cp1` e `cp2`, nel caso in cui tale controllo fallisca deve essere sollevato un errore.

Per chiarire l'implementazione della terza regola, si riporta un frammento differente della funzione usata in precedenza:

```
//uso di assign()
let mut a = Hardened::from(4);
let mut b = Hardened::from(2);
a.assign(b);  //'a=b'

pub fn assign(&mut self, other: Hardened<T>)->Result<(), IncoherenceError>{
    //Regola 3: lettura, controllo di coerenza, errori
    if other.incoherent(){
        return Err(IncoherenceError::AssignFail)
    }
    // [...]
}
fn incoherent(&self)->bool{ self.cp1 != self.cp2 }
```

Usando la funzione `assign()`, poiché leggo la variabile `b` è necessario un controllo di consistenza delle due copie, questo è espletato dalla funzione `incoherent()` che ritorna un booleano. In caso in cui questo test non viene passato, si ritorna un `Err(IncoherenceError)`.

## 3.1 Gestione degli errori

La regola **R3** richiede che, nel caso il controllo di coerenza fallisca, venga sollevato un errore. Si sono utilizzati principalmente due meccanismi per asserire a questo task:

1. Propagazione di un errore di tipo `IncoherenceError`
2. **Uso della macro** `panic!(...)`

Il motivo per il quale si è avuta la necessità di distinguere questi due casi è legata alle caratteristiche del linguaggio Rust. In particolare, alcuni tratti della libreria standard permettono – usando la programmazione generica – di personalizzare sia il tipo dei dati su cui si opera sia il tipo dei valori di ritorno. In altre situazioni, ad esempi nei tratti associati alle *operazioni di confronto*, non è possibile modificare la *firma dei metodi*. Questo è il caso in cui ci si riserva la possibilità di generare un `panic!(...)` nel caso di anomalia rilevata, lasciando però invariata la firma dei metodi garantendo la correttezza sintattica. Di seguito si mostrano due esempio in cui si chiariscono meglio gli aspetti appena introdotti:

**Uso di `IncoherenceError`**

```
impl Add<usize> for Hardened<usize>{
  type Output = Result<Hardened<usize>,
                       IncoherenceError>;
  fn add(self, rhs: usize) -> Self::
    Output {
    if self.incoherent() {
        return Err(IncoherenceError::
    AddFail);
    }
    Ok(Self{
        cp1: self.cp1 + rhs,
        cp2: self.cp2 + rhs,
    })
  }
}
```

Si presenta qui l'implementazione del metodo `add()` del tratto omonimo. La presenza del tipo associato al tratto, permette di non essere vincolati sul tipo di ritorno che quindi è stato personalizzato secondo le nostre esigenze. In particolare poiché l'add come tutte le operazioni di lettura e modifica possono causare errori dovuti all'incoerenza tra le copie interne del dato, si sfrutta l'enumerazione generica `Result<T,E>` per gestire queste due situazioni. Il tipo `T` è `Hardened<T>` mentre il tipo `E` è quello personalizzato (`IncoherenceError` descritto in seguito). Un ragionamento analogo vale per tutti i metodi che hanno una struttura simile a quella presentata che abilita l'utilizzo di dati di ritorno personalizzati.

**Uso di `panic!(...)`**

```
impl<T> Ord for Hardened<T>{
    fn cmp(&self, other: &Self) ->
    Ordering {
        if other.incoherent(){
            panic!("Ord::cmp");
        }
        self.cp1.cmp(&other.cp1)
    }
}
```

Un esempio significativo del caso in cui siamo vincolati sulla scelta del tipo di ritorno viene presentato. In particolare la funzione `cmp(...)` del tratto `Ord` per ovvi motivi vincola il tipo di ritorno ad essere l'enumerativo `Ordering`. Nel caso in cui il dato letto sia "incoerente", viene sollevato un `panic!(...)`. Il messaggio è di cruciale importanza per l'invio dei risultati dell'iniettore verso l'analizzatore.

---

Le due casistiche, come si vedrà, nel *processo di fault injection* vengono gestite in modo diverso, mentre per l'analisi il tipo di informazione associato ai due eventi è analogo.

### 3.1.1 Il tipo di errore `IncoherenceError`

Al fine di personalizzare la semantica degli errori e di facilitare il processo di analisi dei risultati, si è pensato di implementare un **tipo di errore personalizzato** denominato `IncoherenceError`. Il crate `thiserror` permette di derivare l'implementazione del tratto `Error` richiesta da altri meccanismi interni al linguaggio quali la propagazione tramite *question mark operator* e la descrivibilità dell'errore stesso. Il tipo introdotto è un enumerativo:

```
#[derive(Error, Debug, Clone)]
pub enum IncoherenceError{
    #[error("IncoherenceError::AssignFail: assignment failed")]
    AssignFail,
    #[error("IncoherenceError::AddFail: due to incoherence add failed")]
    AddFail,
    #[error("IncoherenceError::MulFail: due to incoherence mul failed")]
    MulFail,
    #[error("IncoherenceError::Generic: generic incoherence error")]
    Generic
}
```

Questo costituisce il tipo `E` integrato nella variante `Err(E)` di `Result`. Con quest'ultimo dettaglio si ha il quadro completo delle trasformazioni del codice atte ad introdurre ridondanza nei dati con le operazioni associate e la gestione degli errori.
Nella prossima sezione si introducono i casi di studio, questi costituiscono l'applicazione di tutti i concetti che abbiamo visto finora sull'irrobustimento del codice.

# 4 Casi di studio

I tre casi di studio presi in considerazione sono:

- Selection Sort
- Bubble Sort
- Moltiplicazioni tra matrici

Per ciascuno di essi vengono implementate due versioni:

- La versione Non Hardened, implementazione standard del caso studio.
- La versione Hardened, basata sul tipo Hardeded e sulle tre regole di trasformazione del codice precedentemente enunciate.

Essendo la ridondanza dei dati e le operazioni associate integrate nell'implementazione del tipo Hardened, per tutti i casi studio le due versioni non differiscono nella strategia algoritmica, ma solamente in piccole varaiazioni sintattiche.
La logica di ordinamento e di calcolo infatti rimane sempre la stessa.

Di seguito vengono riportati i codici dei casi studio.

## 4.1 Selection Sort

L'algoritmo di ordinamento selection sort ordina un vettore trovando il minimo in ogni iterazione e scambiandolo con l'elemento corrente.

Versione Non Hardened:

```
    pub fn selection_sort(mut vet:Vec<i32>)->Vec<i32>{

    let n:usize = vet.len();
    let mut j=0;
    let mut min=0;

    let mut i=0;
    while i< n -1{
        min=i;
        j=i+1;

        while j< n {
            if vet[j] < vet[min]{ min=j; }
            j = j+1;
        }
        vet.swap(min,i);
        i=i+1;
    }
    vet
}
```

Versione Hardened:

```
pub fn selection_sort(vet: &mut Vec<Hardened<i32>>)->Result<(), IncoherenceError>{

    let n:Hardened<usize> = vet.len().into();
    let mut j= Hardened::from(0);
    let mut min = Hardened::from(0);

    let mut i= Hardened::from(0);
    while i<(n -1)?{
        min.assign(i)?;
```

```
        j.assign((i+1)?)?;

        while j< n {
            if vet[j]<vet[min]  {   min.assign(j)?; }
            j.assign((j+1)?)?;
        }
        vet.swap(i.inner()?, min.inner()?);
        i.assign((i+1)?)?;
    }
    Ok(())
}
```

## 4.2 Bubble Sort

L'algoritmo di ordinamento bubble sort è un algortimo che confronta e scambia elementi adiacenti finché i dati non risultano ordinati. La variabile swapped ottimizza il processo in quanto se non vengono effettuati scambi in un ciclo, significa che il vettore è già ordinato.

Versione Non Hardened:

```
    pub fn bubble_sort(mut vet: Vec<i32>) -> Vec<i32> {
        let n:usize = vet.len();
        let mut i = 0;

        while i < n {
            let mut swapped = false;
            let mut j = 0;

            while j < n - i - 1 {
                if vet[j] > vet[j + 1] {
                    vet.swap(j, j + 1);
                    swapped = true;
                }
                j += 1;
            }
            if !swapped {
                break;
            }
            i += 1;
        }
        vet
    }
```

Versione Hardened:

```
pub fn bubble_sort(vet: &mut Vec<Hardened<i32>>) -> Result<(), IncoherenceError> {

    let n = Hardened::from(vet.len());
    let mut i = Hardened::from(0);

    while i < n {
        let mut swapped = Hardened::from(false);
        let mut j = Hardened::from(0);

        while j < ((n - i)? - 1)? {
            if vet[j].inner()? > vet[(j + 1)?].inner()? {
                vet.swap(j.inner()?, (j + 1)?.inner()?);
                swapped = Hardened::from(true);
            }
            j.assign((j + 1)?)?;
```

```
        }
        if !swapped.inner()? {
            break;
        }
        i.assign((i + 1)?)?;
    }
    Ok(())
}
```

## 4.3 Moltiplicazioni tra matrici

L'algoritmo moltiplica due matrici quadrate, calcolando ogni elemento come il prodotto scalare delle righe di una matrice e delle colonne dell'altra.

Versione Non Hardened:

```
    pub fn matrix_multiplication(a: Vec<Vec<i32>>, b: Vec<Vec<i32>>) -> Vec<Vec<i32>>
    {
        let size:usize = a.len();
        let mut result: Vec<Vec<i32>> = Vec::new();

        let mut i = 0;
        let mut j = 0;
        let mut k = 0;

        while i < size {
            let mut row: Vec<i32> = Vec::new();
            j = 0;

            while j < size {
                let mut acc = 0;
                k = 0;

                while k < size {
                    acc += a[i][k] * b[k][j];
                    k += 1;
                }
                row.push(acc);
                j += 1;
            }
            result.push(row);
            i += 1;
        }
        result
    }
```

Versione Hardened:

```
    pub fn matrix_multiplication(a: &Vec<Vec<Hardened<i32>>>, b: &Vec<Vec<Hardened<i32
    >>>) -> Result<Vec<Vec<Hardened<i32>>>, IncoherenceError> {

        let size = Hardened::from(a.len());
        let mut result: Vec<Vec<Hardened<i32>>> = Hardened::from_mat(Vec::new());

        let mut i = Hardened::from(0);
        let mut j = Hardened::from(0);
        let mut k = Hardened::from(0);

        while i < size {
            let mut row: Vec<Hardened<i32>> = Vec::new();
```

```
            j.assign(Hardened::from(0))?;

            while j < size {
                let mut acc = Hardened::from(0);
                k.assign(Hardened::from(0))?;

                while k < size {
                    acc.assign((acc + Hardened::from(a[i.inner()?][k.inner()?].inner()
    ?   *   b[k.inner()?][j.inner()?].inner()?) )? )?;
                    k.assign((k + 1)?)?;
                }
                row.push(acc);
                j.assign((j + 1)?)?;
            }
            result.push(row);
            i.assign((i + 1)?)?;
        }
        Ok(result)
    }
```

# Part II
# Ambiente di fault injection

# 5 Implementazione e descrizione dell'ambiente

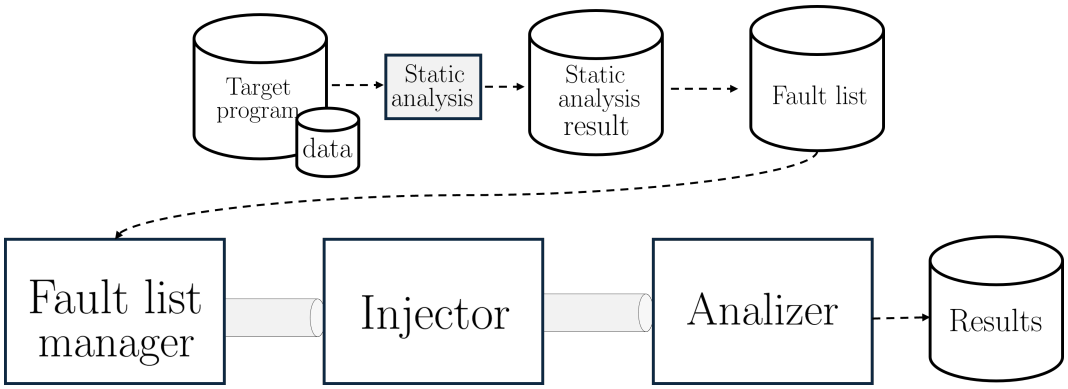


Figure 5.1: Struttura della pipeline

La fonte principale di ispirazione per la realizzazione della parte di applicazione devota ai fault injection è stato [2]. Si possono individuare in questa parte principalmente tre sezioni:

1. Fault List Manager (FLM): genera la **lista dei fault** *fault list* da iniettare nel **programma target**;
2. Injector *(Fault injection Manager)* (FIM): **inietta i fault** nel programma target;
3. Result Analyzer: raccoglie i **risultati**, ne calcola degli **aggregati** e genera un **report** riferito al singolo esperimento di fault injection.

Al fine di parallelizzare i compiti nei vari livelli si è deciso di adottare un *pattern architetturale* che in letteratura è noto come **pipes and filters** (si veda il libro [8] per una trattazione approfondita). Viene abbastanza naturale pensare al sistema in analisi in questi termini: il compito globale si divide naturalmente – come risulta evidente dall'introduzione – in **fasi separate di elaborazione**; inoltre tale struttura si presta molto bene allo sviluppo dell'applicativo in un team costituito da più persone. Per maggiore chiarezza dei concetti esposti si riporta qui una possibile definizione del pattern tratta da [8]:

> *"The Pipes and Filters architectural pattern provides a structure for systems that process a stream of **data**. Each processing step is encapsulated in a **filter component**. Data is passed through **pipes** between adjacent filters. Recombining filters allows you to build families of related systems"*

Di seguito si riporta la figura presentata all'inizio in cui in aggiunta vengono mappati i componenti citati nella definizione.

## 5.1 Implementazione in Rust

Ogni componente della pipeline mostrata trova implementazione in Rust in opportuni componenti software. La sorgente dei dati è costituita da un sottosistema esegue l'**analisi statica automatica** del **programma target** producendo un primo report (*Static analysis result)* (data preparation) che viene utilizzato a sua volta da una routine preposta che si occupa di **generare la fault list**. I tre filtri sono subroutine allocate in appositi moduli (vedi **Sezione 11** per la struttura del codice sorgente). Le due pipeline tramite le quali comunicano i tre stage sono implementate tramite 2 canali *multiple producer single channel* (`mpsc`). Ogni filtro[1] utilizza come struttura di **programmazione concorrente** i thread. In conclusione di questa sezione forniamo: (i) lo snippet di codice che riporta la **funzione di setup** della pipeline; una **tabella riassuntiva** (Tabella 1) di tutti gli elementi della pipeline descrizione, ruolo e sezione di codice a cui afferisce.

[1] Questi in letteratura sono noti come *filtri attivi* in quanto rispettano il seguente principio: "[...] the filter is active in a loop, pulling its input from and pushing its output down the pipeline."(cfr. [8])

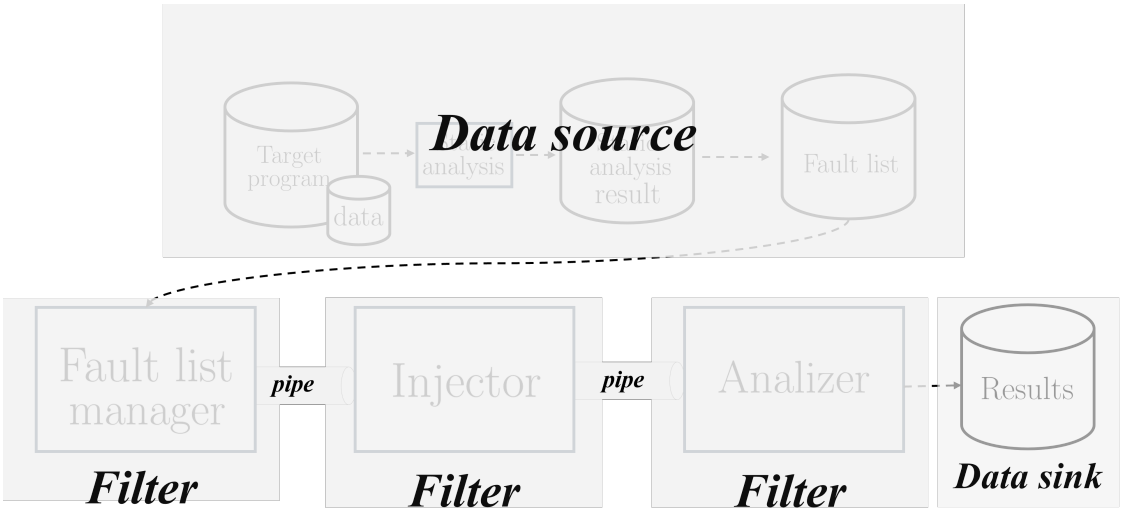


Figure 5.2: Pattern architetturale **pipes and filters**

```rust
pub fn fault_injection_env(fault_list: String,    //Data Source: Fault List
                           target: String,        //Data Source: Target program
                           file_path: String,     //Data Sink: Report
                           data: Data<i32>) {     //Data Source: Data

    let (tx_chan_fm_inj, rx_chan_fm_inj) = channel();          //pipe FLM-FIM
    let (tx_chan_inj_anl, rx_chan_inj_anl) = channel();        //pipe FIM-Analyzer

    fault_manager(tx_chan_fm_inj,fault_list);
    injector_manager(rx_chan_fm_inj, tx_chan_inj_anl, target, data);
    analyzer(rx_chan_inj_anl,file_path);
}
```

| Componente | Ruolo | Descrizione |
|---|---|---|
| Static analysis result | Data source | Dato il programma target lo analizza e effettua un conteggio delle istruzioni e ne estrae le informazioni sulle variabili (tipo, nome, dimensione...) Il sottomodulo `mod static_analysis` contiene tutte le funzioni collegate a questo task. |
| Target program | Data source | Codice sorgente dei casi di studio oggetto dell'analisi: irrobustimento dei dati e fault injection. La directory `fault_list_manager/file_fault_list` contiene i file `*.rs` input dell'analisi statica. |
| Data | Data source | Sono i dati su cui lavorano gli algoritmi selezionati come casi di studio. Nel file `data/input.txt` ci sono: (i) La matrice di Wilson e la sua inversa (per la moltiplicazione di matrici); (ii) un campione di vettore dal dataset di benchmark [1] (per gli algoritmi di ordinamento) |
| Fault list | Data source | Sfruttando il report prodotto dall'analisi statica viene generata la fault list contenente un numero di entry scelto dall'utente o prefissato. Il modulo `mod fault_list_manager` contiene le funzioni adibite a tale compito. |
| Fault list manager | Filter | Stage della pipeline che scorre la fault list, preleva una fault entry per volta e la spedisce nella *pipe* verso l'iniettore incapsulato in una struttura di tipo `FaultListEntry`. (`mod fault_list_manager`) |
| Injector | Filter | Prelevando dal canale le `FaultListEntry` esegue gli esperimenti di iniezione e spedisce i risultati grezzi da analizzare a valle nella pipeline. `(mod injector)` |
| Analizer | Filter | Stage della pipeline che riceve i risultati dei test dall'iniettore, calcola delle statistiche e produce il report finale dell'esperimento. (`mod analizer`) |
| Results | Data sinks | File contenente il report dell'esperimento di fault injection con grafici e tabelle. Viene memorizzato nella directory `result`. |
| Canali `mpsc` | pipes | Costituiscono il collante degli stage della pipeline. Le istruzioni 6-7 creano le estremità delle due pipeline, queste vengono poi distribuite ai vari filtri. |

Table 1: Tabella riassuntiva *Fault injection environment*

# 6 Data source

In questa sezione sono descritte le operazioni che precedono la generazione della fault list e l'init dello stage della pipeline preposto a generare le fault entry. Tutto il lavoro viene svolto essenzialmente da un sottomodulo di `mod fault_list_manager` denominato `mod static_analysis`.

## 6.1 Sottomodulo `mod static_analysis`: analisi statica automatica del codice

Come abbiamo chiarito dall'inizio, il nostro obiettivo – dopo l'irrobustimento dei dati tramite ridondanza – è quello di eseguire degli *esperimenti di fault injection* dove i target dei fault sono le variabili coinvolte negli algoritmi scelti come casi di studio. Tuttavia affinché si possa generare una fault list servono delle informazioni che descrivono il codice del caso di studio stesso. Nello specifico serve poter rispondere alle seguenti domande:

- ✔ **Quante istruzioni** ha l'algoritmo?
- ✔ **Quante e quali variabili**?
- ✔ Per ogni variabile, **qual è la sua etichetta**? La sua **dimensione**?
- ✔ Qual è la prima istruzione in cui compare quella variabile?

Il sottomodulo `static_anlysis` è preposto a rispondere a tutte queste domande. Il codice sorgente viene analizzato dalla **libreria di parsing** `syn` (vedi [10]); questa permette di trasformare una stringa contenente il codice stesso in un **syntax tree** da cui, utilizzando diversi metodi si possono ricavare le informazioni necessarie. In particolare, al modulo appena citato, vengono affidate in ordine le seguenti operazioni, dato il singolo caso di studio:

1. Lettura da un file di testo la sua implementazione in linguaggio Rust (sono tutte contenute in `file_fault_list/<casodistudio>`), dove

   `<casodistudio>`$\in$ {`selection_sort, bubble_sort, matrix_multiplication`}

2. Trasformazione della stringa contenente il codice in un **albero di sintassi**;
3. Esecuzione di una visita in profondità del **syntax tree** ottenuto al fine di ricavare le informazioni richieste; queste poi vengono salvate in una struttura dati di tipo `ResultAnalysis` di cui riportiamo la definzione dopo;
4. La struttura dati ottenuta viene infine **serializzata** in un file `json` usando il crate `serde` (si veda [9] per la documentazione ufficiale) contenuto nella directory corrispondente al caso di studio sotto `file_fault_list`. Così facendo, l'analisi statica del codice può essere fatta indipendentemente dalla generazione della fault list e dall'esperimento di fault injection.

```rust
#[derive(Serialize, Deserialize, Debug)]
pub struct Variable {
    pub name: String,        //nome
    pub ty: String,          //tipo
    pub size: String,        //dimensione
    pub start: usize         //tempo di dichiarazione
}
#[derive(Serialize, Deserialize, Debug)]
pub struct ResultAnalysis{
    pub num_inst: usize,
    pub vars: Vec<Variable>
}
```

Il tipo `Variable` ha la funzione di descrivere tutte le informazioni legate alla singola variabile. Si fa notare che per restituire una dimensione parametrizzata di una certa variabile (vettori/matrici), il campo `size` è di tipo `String`. Per completezza si riporta di seguito un frammento del file risultato dell'analisi statica.

```
"num_inst": 19,
"vars": [{  "name": "a","ty": "Vec < Vec < i32 > >",
            "size": "4*nR*nC","start": 1},
         {  "name": "b","ty": "Vec < Vec < i32 > >",
```

```
            "size": "4*nR*nC","start": 1},
        {   "name": "size", "ty": "usize","size": "4",
            "start": 1 }, ...]
```

### 6.1.1 Metodi e descrizione

Si mostra di seguito la gerarchia delle funzioni presenti in `mod static_analysis`, si riporta poi una tabella in cui per ognuna delle funzioni si fornisce una *breve descrizione*.

```
fn generate_analysis_file()
└─fn analyze_function()
  ├─fn count_statements()
  │ └─fn infer_type_from_expr()
  └─fn extract_variables()
    └─fn type_size()
```

| Funzione | Descrizione |
|---|---|
| `fn generate_analysis_file()` | Funzione wrapper che legge tutto il contenuto del file e lo memorizza in una stringa. Questa dopo essere stata trasformata in un formato compatibile con il parser, diventa l'input della funzione successiva. Per generalizzare quanto più possibile vengono cercate all'interno del codice le funzioni (`Item::Fn`). Tuttavia all'interno di ogni file c'è il codice di una sola funzione. |
| `fn analyze_function()` | Chiama le funzioni `count_statements()` e `extract_variables()` per il conteggio delle istruzioni ed estrazione delle informazioni di variabili associate ai parametri formali della funzione o al corpo della funzione stessa. |
| `fn count_statements()` | [Funzione ricorsiva] Conta le istruzioni associati agli statement del *blocco di codice* passato come parametro. Ogni statement può essere di tipo `Stmt::Local` o `Stmt::Expr`. Nel primo caso si tratta di un istruzione semplice di cui si possono già estrarre le informazioni sulle variabili usando le funzioni che seguono in questa descrizione. Nel secondo caso si tratta di un blocco di codice composto (`Expr::If, Expr::While, Expr::ForLoop`) in questo caso viene chiamata **in modo ricorsivo** la stessa `fn count_statements()` e viene passato come input il blocco 'figlio' di questo tipo di espressione. |
| `fn infer_type_from_expr()` | [Funzione ricorsiva] Funzione utilizzata per inferire il tipo di una certa espressione. Questa a sua volta può essere di tipo `Expr::Lit` (literal) o `Expr::Assign` (operazione di assegnazione), nel secondo caso viene chiamata ricorsivamente la stessa funzione per inferire il tipo del *Right Hand Side*(RHS). L'output di questa funzione costituisce l'input della funzione `type_size()`. |
| `fn extract_variables()` | Si prende cura di analizzare le informazioni corrispondenti ai **parametri della funzione**. |
| `fn type_size()` | Effettua il binding tipo-dimensione. Utilizzata da `count_statement()` per popolare il campo `size` della struttura di tipo `ResultAnalysis`. |

Table 2: `mod static_analysis()`

### 6.1.2 Un piccolo CAVEAT per la scrittura del codice

Poiché il modulo di analisi statica prende in pasto un file con il codice dell'algoritmo (non irrobustito), questo deve essere scritto prestando attenzione ad un piccolo particolare che se non rispettato potrebbe far fallire l'inferenza del tipo. Più nello specifico, nella dichiarazione di una variabile, se questa viene ricavata tramite assegnazione di un'altra variabile già esistente, bisogna utilizzare un'**annotazione esplicita di tipo**. Si mostra di seguito un esempio:

```
let mut a=13;
let mut b=15;
let mut c:i32 =a+c;
```

Mentre nei primi due casi il tipo viene inferito direttamente[2], nella terza istruzione bisogna aggiungerlo esplicitamente perché questo dipende da un'altra variabile. Questo dettaglio non è stato implementato per non aggiungere complessità ad un codice già non banale.

# 7 Fault List Manager

Dopo un percorso abbastanza articolato, abbiamo ottenuto *in modo automatico* le informazioni sulle variabili facendo passare il codice sorgente del caso di studio attraverso i metodi dell'analisi statica.
Il *glossario* che viene dall'output di questa fase è di cruciale importanza per la generazione della fault list descritta in questo paragrafo.
Prima di entrare in merito della discussione è doveroso aggiungere un dettaglio non trascurabile. Gli algoritmi (ordinamento/moltiplicazione di matrici) possono lavorare su **dati diversi** ad ogni esecuzione. Per cui per sapere effettivamente il numero di istruzioni che quel set di dati genera bisognerebbe eseguire quel codice! A questo scopo sono state implementate delle funzioni denominate `run_for_count_<casodistudio>()` a cui è associata un'esecuzione fittizia del codice in analisi. Il loro output è un intero associato al numero di istruzioni che un certo algoritmo applicato ad un certo set di dati (matrice o vettore) produce.

Abbiamo accennato nell'introduzione che il modello di guasto che vogliamo adottare è quello del **single bit-flip**. Durante la vita dell'applicativo si possono verificare un certo numero di fault. Per simularne l'occorrenza ed eseguire gli esperimenti sul codice modificato, viene generata randomicamente una lista di guasti.

## 7.1 Fault list entry

La struttura dati preposta a contenere le informazioni sul singolo fault è la seguente:

```
#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct FaultListEntry{
    pub var: String,
    pub time: usize,
    pub flipped_bit: usize,
}
```

dove `var` è il **nome della variabile**, `time` è il **tempo di iniezione**, mentre `flipped_bit` è il bit della variabile che è stato modificato dal guasto. Sul tipo `FaultListEntry` sono derivati, tra gli altri, i tratti `Serialize` e `Deserialize` utili per il *marshalling* della fault list su file.

## 7.2 Generazione della fault list

L'algoritmo di generazione della fault list è riportato di seguito.

Alla fine della sua creazione, la fault list viene serializzata in formato `json` alla stregua di quanto fatto per il glossario di analisi statica. La costante NUM_FAULT se non settata dall'utente da linea di comando viene impostata a 5000, mentre la costante N_INST è l'output delle funzioni `run_for_count()`. Il tempo di iniezione viene scelto in modo compatibile, nel senso che non posso iniettare in una variabile se prima questa non è stata dichiarata, ecco perché si è scelto di ricavare durante l'analisi dell'albero di sintassi anche l'informazione memorizzata in `Variable::start`. Riportiamo uno stralcio di fault list nello snipppet che segue:

```
    {   "var": "swapped",   "time": 218,    "flipped_bit": 6    },
    {   "var": "j",         "time": 15,     "flipped_bit":      },
    {   "var": "swapped",   "time": 102,    "flipped_bit": 4    },
    {   "var": "n",         "time": 20,     "flipped_bit": 24   },
```

## 7.3 Stage pipeline

Una volta creata, la fault list può essere utilizzata e possiamo far partire l'**esperimento di fault injection**. Lo stage della pipeline associata al Fault list manager è espletato dalla seguente funzione:

[2] Nella funzione `infer_type_from_expr()` mi accorgo che ho un `Expr::Assign`, quindi chiamo ricorsivamente la stessa funzione che a questo punto trova la `Expr::Literal` associata all'intero e viene ritornato il tipo corrispondente.

**Algorithm 1** Algoritmo di *Generazione della fault list*

**for** i in [0, NUM_FAULT] **do**
  *var* ← rand(1,NUM_VAR) ▷ Scelgo una variabile tra quelle disponibili
  **if** var.type == 'matrix' **then**
    *r* ← rand(0,var.nR) ▷ Seleziono la riga
    *c* ← rand(0,var.nC) ▷ Seleziono la colonna
    *var* ← mat[r][c]
    *flipped_bit* ← rand(0, var.size-1) ▷ Scelgo il bit da flippare
    *time* ← rand(var.start, N_INST) ▷ Scelgo il tempo di iniezione
  **else if** var.type == 'vector' **then**
    *index* ← rand(0, var.len)
    *var* ← vec[index]
    *flipped_bit* ← rand(0, var.size-1) ▷ Scelgo il bit da flippare
    *time* ← rand(var.start, N_INST) ▷ Scelgo il tempo di iniezione
  **else**
    *flipped_bit* ← rand(0, var.size-1) ▷ Scelgo il bit da flippare
    *time* ← rand(var.start, N_INST) ▷ Scelgo il tempo di iniezione
  **end if**
  FaultList.insert(*var.name*, *time*, *flipped_bit*) ▷ Creo la fault entry e la inserisco nella fault list
**end for**

```rust
pub fn fault_manager(tx_chan_fm_inj: Sender<FaultListEntry>, fault_list:String){
    let flist_string = fs::read_to_string(fault_list).unwrap();
    let flist:Vec<FaultListEntry>=serde_json::from_str(&flist_string.trim()).unwrap();
    flist.into_iter().for_each(|el|tx_chan_fm_inj.send(el).unwrap());
    drop(tx_chan_fm_inj);
}
```

Gli input di questo stage sono il path del file in cui è contenuta la fault list (data source) e l'estremità del canale (pipe) verso l'iniettore. Sono eseguite in ordine le seguenti operazioni:

1. Lettura del file di testo in una stringa (`flist_string`);
2. Deserializzazione della stringa in una collezione `Vec<FaultListEntry>` usando la funzione `serde_json::from_str()`
3. La collezione viene trasformata in un **iteratore** di `FaultListEntry`. Ogni elemento estratto da questo iteratore tramite il metodo `for_each()` viene mandato nel canale verso l'iniettore che lo utilizzerà in modo opportuno.

# 8 Injector

## 8.1 Aspetti Generali

L'iniettore e' stato pensato come un componente della pipeline che riceve le fault list entry dal fault list manager, utilizzandole poi per iniettare gli errori nel momento corretto durante l'esecuzione dell'algoritmo tesato. Il risultato dell'esecuzione viene poi utilizzato per creare il TestResult relativo alla singola fault list entry e passato al successivo stadio della pipeline.

Per l'implementazione dell'iniettore vengono utilizzati 2 thread, uno per l'esecuzione dell'algoritmo che chiameremo *runner*, e uno per l'esecuzione dell'iniettore che chiameremo *injector*. I due thread condividono le variabili in uso che, durante un'istanza dell'esecuzione dell'algoritmo sotto esame (un'istanza per ciascuna fault list entry), verranno lette e modificate da entrambi i thread: il thread runner leggera' e modifichera' le variabili seguendo l'ordine delle istruzioni dell'algoritmo, il thread injector leggera' la variabile su cui iniettare l'errore per poter calcolare il nuovo valore (ovvero quello contenente l'errore) e modificandola di conseguenza. Affinche' i due thread si sincronizzino correttamente e l'iniezione dell'errore avvenga nell'istante specificato nella fault list entry, i due thread utilizzano 2 canali monodirezionali *mpsc* in modo che dopo ogni istruzione dell'algoritmo eseguita dal runner venga mandato un messaggio all'injector su un canale e ne venga attesa la risposta sull'altro. L'*injector*

## 8.2 Aspetti tecnici

### 8.2.1 Injector Manager

La funzione chiamata *injector_manager* ha la funzione di coordinare la ricezione delle fault list entry provenienti dallo stato precedente della pipeline tramite un canale dedicato, ricevendo anche il canale per trasmettere i risultati, l'algoritmo target e i dati da usare durante l'analisi.

```
pub fn injector_manager(rx_chan_fm_inj: Receiver<FaultListEntry>,
                tx_chan_inj_anl: Sender<TestResult>,
                target: String,
                data: Data<i32>);
```

Al suo interno la funzione tramite un ciclo while attende la ricezione sul canale delle fault list entry e, per ciascuna, crea il set di variabili utilizzate (in base al tipo di algoritmo in esecuzione), i 2 canali con cui i thread gestiranno la sincronizzazione e i 2 thread *runner* e *injector*.

Affinche' siano testabili piu' algoritmi, ciascuno avente il proprio set di variabili che utilizza, e' stata usata un'enum chiamata *AlgorithmVariables* contenente per ciascun algoritmo una struct contenente le variabili.

```
enum AlgorithmVariables {
    SelectionSort(SelectionSortVariables),
    BubbleSort(BubbleSortVariables),
    MatrixMultiplication(MatrixMultiplicationVariables),
}
```

Le struct relative ai singoli algoritmi contengono, per ogni variabile, un *RwLock* contenente a sua volta il tipo *Hardened* corrispondente. Dovendo condividere questa struttura tra piu' thread eseguiti, era necessario renderla accessibile in modo sicuro (dovendo essere sia letta che scritta) e per questo motivo una possibile soluzione era quella di racchiudere la struttura per intero all'interno di un *Mutex* o *RwLock*. Questa soluzione presentava pero' delle criticita'. Per effettuare il controllo condizionale per i cicli while era richiesto di acquisire il lock prima del check sulla condizione del ciclo, ma una volta acquisito il lock fuori dal ciclo questo veniva mantenuto per l'intera durata del ciclo, impedendo all'*injector* di iniettare l'errore su una delle variabili. Di conseguenza l'opzione migliore e che richiedesse meno overhead a livello di codice era racchiudere ciascuna singola variabile della struct in un RwLock anziche' la struttura per intero. La scelta di utilizzare RwLock e' stata motivata principalmente da una possibile migliore gestione delle read e write, dovuta a numero di letture e scrittura sbilanciato in base all'algoritmo eseguito.

```
struct SelectionSortVariables {
    i: RwLock<Hardened<usize>>,
    j: RwLock<Hardened<usize>>,
    N: RwLock<Hardened<usize>>,
    min: RwLock<Hardened<usize>>,
```

```
    vec: RwLock<Vec<Hardened<i32>>>,
}
```

Una volta creata la struct contenente le variabili della fault list entry corrente, vengono aperti i canali di comunicazione tra *runner* e *injector* ed eseguiti i rispettivi due thread. Quando il thread *runner* termina invia all'analizzatore (stadio di pipeline successivo) i risultati ottenuti.

### 8.2.2 Runner

Il thread *runner* esegue una funzione wrapper chiamata *runner* la quale si occupa di lanciare l'esecuzione dell'algoritmo irrobustito corretto per il tipo di analisi che si sta facendo e gestendo il risultato prodotto da questo.

```
fn runner(variables: Arc<AlgorithmVariables>,
          fault_list_entry: FaultListEntry,
          tx_runner: Sender<&str>,
          rx_runner: Receiver<&str>) -> TestResult
```

In base al tipo di algoritmo *target* sono stati creati degli algoritmi ad-hoc per poter interagire correttamente con l'iniettore. Questi sono delle versioni rivisitate delle versioni irrobustite originali, le quali non sarebbero state in grado di sincronizzarsi con l'iniettore per subire i fault. Di seguito viene descritta la struttura di questi algoritmi, facendo esempi relativi al Selection Sort, in quanto gli altri seguono tutti la stessa logica.

**Algoritmo Testato** Ciascun algoritmo riceve le variabili da utilizzare, il canale su cui trasmettere il completamento di un'istruzione e quello su cui attendere l'eventuale inserimento del fault.

```
pub fn runner_matrix_multiplication(variables: &MatrixMultiplicationVariables,
            tx_runner: Sender<&str>,
            rx_runner: Receiver<&str>) -> Result<(), IncoherenceError>
```

La procedura per l'esecuzione di una qualsiasi istruzione e':

- Accesso al lock con conseguente lettura/scrittura della variabile
- Scrittura sul canale *tx_runner* per comunicare all'*injector* che un'istruzione e' stata eseguita
- Attesa sul canale *rx_runner* che l'*injector* termini le sue operazioni, necessario affinche' *runner* e *injector* rimangano sincronizzati

Riportiamo di seguito un esempio di un'istruzione equivalente all'istruzione $j.assign((i+1)?)?$:

```
// j = i + 1   -- versione non irrobustita
// j.assign((i+1)?)?    -- versione irrobustita
variables.j.write().unwrap().assign((*variables.i.read().unwrap() + 1)?)?;
tx_runner.send("").unwrap();
rx_runner.recv().unwrap();
```

L'algoritmo ritorna un Result, contenente:

- *Ok(())*: successo ed esecuzione portata a termine correttamente'
- *Err<IncoherenceError>*: variante dell'enum *IncoherenceError* che descrive il tipo di errore riscontrato

**Terminazione Runner** Il runner termina eseguendo un pattern match sul risultato dell'algoritmo eseguito, producendo il TestResult che verra' utilizzato dall'analizzatore per ottenere statistiche utili.

## 8.3 Injector

L'*injector* e' una funzione che si occupa di iniettare nel momento corretto il fault contenuto nella fault list entry sulla variabile indicata. Per fare cio', riceve le variabili usate dall'algoritmo e condivise con il *runner*, la fault list entry e i canali necessari per la sincronizzazione con il *runner*.

```
fn injector(variables: Arc<AlgorithmVariables>,
            fault_list_entry: FaultListEntry,
            tx_injector: Sender<&str>,
            rx_runner: Receiver<&str>)
```

L'informazione sul tipo di algoritmo in esecuzione e' ricavata dal tipo di variabili ricevute, essendo queste un'istanza dell'enum *AlgorithmVariables*. Viene poi manutenuto un *counter* necessario a contare il numero di istruzioni eseguite per poi al momento indicato nella fault list entry iniettare l'errore. Tramite un ciclo while, che termina quando il canale condiviso con il *runner* viene chiuso, vengono ricevuti gli impulsi che indicano la terminazione di un'istruzione. Il flusso di operazioni eseguite e':

1. Calcola la maschera in base al bit indicato nella fault list entry
2. Per ogni segnale ricevuto dal *runner*:
   - 2.1 Incrementa il counter
   - 2.2 Se $counter == fault\_list\_entry.time$
     - i. Ricava la variabile su cui iniettare contenuta nella fault list entry
     - ii. Tramite match inietta sulla variabile la maschera calcolata
   - 2.3 Manda sul canale verso il *runner* il segnale per la continuazione della sua esecuzione

Le maschere vengono calcolate come $mask = 2^{fault\_mask}$. Le maschere vengono applicate alle variabili tramite XOR. Prendendo un esempio per quanto riguarda il Selection Sort:

```
"i" => {
    let val = var.i.read().unwrap().inner().unwrap().clone();   // leggo il valore
    della variabile
    let new_val = val \wedge mask;                                    // nuovo valore
    da salvare (XOR per il bitflip)
    var.i.write().unwrap()["cp1"] = new_val;                   // inietto l'errore
}
```

# 9 Analizzatore

L'analizzatore si colloca come elemento conclusivo della pipeline di *fault injection*. La sua funzione principale è raccogliere e organizzare i risultati generati durante l'iniezione di fault negli algoritmi sottoposti a test al fine di fornire poi una visione dettagliata del comportamento degli algoritmi irrobustiti e non, in presenza di fault.
Tali dettagli sono riassunti in un report in formato pdf generato dinamicamente in base ai risultati ottenuti.

## 9.1 Struct Analyzer e Faults

Per memorizzare e gestire i dati rilevanti, è stata progettata una struttura dati denominata **Analyzer** che viene presentata di seguito:

```
#[derive(Serialize,Deserialize,Debug,Clone)]
pub struct Analyzer{
    pub(crate) n_esecuzione: i8,
    pub(crate) faults: Faults,
    pub(crate) input: Data<i32>,
    pub(crate) output: Data<i32>,
    pub(crate) time_experiment: f64,
    pub(crate) time_alg_hardened: f64,
    pub(crate) time_alg_not_hardened: f64,
    pub(crate) byte_hardened: f64,
    pub(crate) byte_not_hardened: f64,
    pub(crate) target_program: String,
}
```

Al suo interno, il campo *faults* è di tipo **Faults**, una struttura che include una serie di contatori dedicati. Questi contatori vengono incrementati ogni volta che un fault specifico viene rilevato sul canale di comunicazione tra l'iniettore e l'analizzatore. Di seguito sono riportati i contatori presenti nella struttura **Faults**:

```
#[derive(Serialize,Deserialize,Debug,Clone)]
pub struct Faults{
    pub(crate) n_silent_fault: usize,
    pub(crate) n_assign_fault: usize,
    pub(crate) n_inner_fault: usize,
    pub(crate) n_sub_fault: usize,
    pub(crate) n_mul_fault: usize,
    pub(crate) n_add_fault: usize,
    pub(crate) n_indexmut_fault: usize,
    pub(crate) n_index_fault: usize,
    pub(crate) n_ord_fault: usize,
    pub(crate) n_partialord_fault: usize,
    pub(crate) n_partialeq_fault: usize,
    pub(crate) n_fatal_fault: usize,
    pub(crate) total_fault: usize,
}
```

Un'attenzione particolare è dedicata ai fault silent i quali rappresentano errori iniettati durante l'esecuzione di un algoritmo ma non intercettati dal sistema irrobustito. Sebbene la maggior parte di questi fault non abbia un impatto diretto sull'output del sistema, una piccola percentuale (circa il 10%) può generare risultati errati, evidenziando casi critici in cui il sistema irrobustito fallisce nel mantenere l'integrità dell'elaborazione.

## 9.2 Funzionalità dell'analizzatore

Per semplicità possiamo affermare che per ogni fault iniettato, in generale l'analizzatore distingue le due seguenti macrocategorie:

- **Fault silent**: rappresentano gli errori non intercettati dal sistema irrobustito.

- **Fault identificati**: corrispondono agli errori rilevati dal sistema irrobustito, categorizzati in base all'operazione specifica che li ha generati (ad esempio, operazioni di assegnazione, somma o moltiplicazione).

In aggiunta alla rilevazione e categorizzazione degli errori, l'analizzatore tiene traccia di metriche chiave legate all'overhead introdotto dall'irrobustimento del codice, come:

- **Tempi di esecuzione**: confronto tra i tempi necessari per eseguire il codice irrobustito e quello non irrobustito.
- **Dimensione del file**: differenza nella dimensione dei file contenenti il codice irrobustito e non irrobustito.

## 9.3 Tipologie di analisi

L'analizzatore supporta tre modalità principali di analisi, ognuna delle quali genera un report in formato PDF, salvato nella cartella results. Di seguito una panoramica:

1. Analisi singola:
   - Analizza un singolo algoritmo su cui vengono iniettati un numero prefissato di fault.
   - Produce un file PDF denominato <*nome_file*>.pdf.
2. Analisi su più algoritmi:
   - Valuta il comportamento di tre algoritmi diversi: selection sort, bubble sort e matrix multiplication.
   - Produce un file PDF denominato <*nome_file*>_all.pdf.
3. Analisi su diverse cardinalità:
   - Analizza un singolo algoritmo utilizzando tre diverse cardinalità della lista di fault (1000, 2000 e 3000 fault).
   - Produce un file PDF denominato <*nome_file*>_diffcard.pdf.

Ogni report include informazioni dettagliate sui fault rilevati e non rilevati, insieme alle metriche di performance e dimensioni del codice. Questo sistema di analisi offre una visione completa dell'efficacia del processo di irrobustimento e del relativo impatto su risorse e prestazioni.

# 10 Risultati

## 10.1 Aspetti Generali

A valle dell'esperimento di fault injection viene lanciato l'analizzatore il quale si preoccupa di andare a raccogliere i dati

# 11 Struttura del codice sorgente

Al fine di fornire un codice ben organizzato, data la media dimensione in termini di righe di codice del software sviluppato, si è deciso dal principio di strutturarlo in **moduli** e **sottomoduli**. Il seguente schema ne mostra l'organizzazione.

```
src/
├── hardened ..................  Definizione e implementazione del tipo
│                                Hardened<T>; definizione del tipo enumerativo
│                                IncoherenceErorr
│
├── fault_list_manager .........  Generazione della fault list
│   └── static_analysis .........  Analisi statica automatica del codice
├── injector ...................  Iniettore dei fault
│   └── algorithms .............    Adattamento del codice dei tre casi di studio
│                                  alla fault injection software
└── analizer ...................  Calcoli di aggregati e statistiche, analisi
                                 dei risultati, generazione di report in pdf
```

Per ogni modulo, sono stati scritti dei **test d'unità** conformi al **paradigma AAA** (Arrange, Act, Assert), questo al fine di verificare che tutte le funzioni implementate dessero i risultati desiderati. Al fine di essere quanto più allineati con le convenzioni adottate dalla comunità di sviluppatori Rust, ognuno dei moduli ha un sottomodulo `tests` definito come segue:

```rust
#[cfg(test)]
mod tests{
    #[test]
    fn test_add_ok(){
        //Arrange
        let a = Hardened::from(3);
        let b = Hardened::from(2);
        //Act
        let c = (a+b);
        //Assert
        assert_eq!(c.is_ok(), true);
        assert_eq!(c.unwrap().inner().unwrap(), 5);
    }

    #[test]
    fn test_other(){
        unimplemented!()
    }
}
```

Dove `#[cfg(test)]` permette di eseguire tutti i test relativi ad un modulo quando viene lanciato da terminale il comando `cargo test`.

# Riferimenti


[1] *Benchmark Dataset for Sorting Algorithms*. en. URL: https://www.kaggle.com/datasets/bekiremirhanakay/benchmark-dataset-for-sorting-algorithms (visited on 11/21/2024).

[2] A. Benso et al. "A fault injection environment for microprocessor-based boards". In: *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*. ISSN: 1089-3539. Oct. 1998, pp. 768–773. DOI: 10.1109/TEST.1998.743259. URL: https://ieeexplore.ieee.org/abstract/document/743259 (visited on 11/18/2024).

[3] *Dependability*. en. Page Version ID: 1237650888. July 2024. URL: https://en.wikipedia.org/w/index.php?title=Dependability&oldid=1237650888#cite_note-1 (visited on 11/18/2024).

[4] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (1997), pp. 75–82.

[5] D.K. Pradhan J.A. Clark. "Fault Injection: a method for validating Computing-System Dependability". In: *Computer* pp.47-56 (June 1995).

[6] M. Rebaudengo et al. "Soft-error detection through software fault-tolerance techniques". In: *Proceedings 1999 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (EFT'99)*. 1999, pp. 210–218. DOI: 10.1109/DFTVS.1999.802887.

[7] Maurizio Rebaudengo et al. "Soft-error detection through software fault-tolerance techniques". In: *Proceedings 1999 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (EFT'99)*. IEEE. 1999, pp. 210–218.

[8] Douglas C Schmidt et al. *Pattern-oriented software architecture, patterns for concurrent and networked objects*. John Wiley & Sons, 2013.

[9] *serde - Rust*. URL: https://docs.rs/serde/latest/serde/.

[10] *syn - Rust*. URL: https://docs.rs/syn/latest/syn/.