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** [1] 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 [3] – 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 [2] 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 **Sezione 3** viene introdotto un **set di trasformazioni del codice** alla stregua di parte di quello che viene presentato in [5]. 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 5** fornisce un'analisi comparativa del codice non

irrobustito rispetto a quello che utilizza le variabili di tipo generico `Hardened<T>`. Nella **Sezione 6** viene definita la struttura dell'ambiente di fault injection, mentre le **Sezioni 7, 8, 9** si occupano di analizzarne i componenti.

2 Struttura del codice sorgente

Part I
Irrobustimento del codice

3 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 restringiamo 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.

3.1 Tre regole per la trasformazione del codice

Le regole di trasformazione del codice citate sono quelle proposte in [4]. 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)**.

3.2 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'**asseegnazione** 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).

4 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)`.

4.1 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!("Found an incoherence!");
        }
        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!(...)`.

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.

4.1.1 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. Inoltre, come presentiamo, non ci sono differenze sostanziali di sintassi tra la versione che utilizza i tipi `Hardened` e quella che utilizza i tipi standard, in quanto la ridondanza dei dati e le operazioni che questa porta con sè sono tutte **integrate** nell'implementazione del tipo argomento di discussione del presente paragrafo.

5 Casi di studio

5.1 Selection Sort

5.2 Bubble Sort

5.3 Moltiplicazioni tra matrici

Part II
Ambiente di fault injection

6 Ambiente di Fault Injection

7 Fault List Manager

7.1 Analisi statica automatica del codice

7.2 Generazione della fault list

7.3 Stage pipeline

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 utilizzata per creare il TestResult relativo alla singola fault list entry, passato al successivo stadio di 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 una 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 le 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 *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.

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 il vettore usato durante l'analisi.

```
pub fn injector_manager(rx_chan_fm_inj: Receiver<FaultListEntry>,
                tx_chan_inj_anl: Sender<TestResult>,
                target: String,
                vec: Vec<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),
    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, c'era la necessita' di 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/RwLock*. Questa soluzione presentava pero' delle criticita'. Per effettuare il controllo condizionale per i cicli while era richiesto di acquisire il lock e poi successivamente effettuare il controllo ma in questo modo

```
struct SelectionSortVariables {
    i: RwLock<Hardened<usize>>,
    j: RwLock<Hardened<usize>>,
    N: RwLock<Hardened<usize>>,
    min: RwLock<Hardened<usize>>,
    vec: RwLock<Vec<Hardened<i32>>>,
}
```

Le variabili vengono definite grazie alla struct *Variables*, contenente tutte le variabili utilizzate dall'algoritmo testato. Dovendo condividere questa struttura tra i due thread eseguiti, c'era la necessita' di renderla accessibile in modo sicuro (dato che viene sia letta che scritta) e per questo motivo una possibile soluzione era quella di racchiudere la struttura in un *Mutex/RwLock*. Questa soluzione presentava pero' una criticita': se acquisivo il lock all'inizio del codice, dovevo rilasciarlo e riacquisirlo prima e dopo ogni istruzione, per permettere all'injector di modificare la variabile di interesse quando necessario, andando inoltre a bloccare l'intera struttura. Per questa ragione ho pensato di utilizzare un *RwLock* per ciascuna variabile, in modo da rendere il codice piu' leggero e bloccando solamente la singola istruzione necessaria. La scelta di utilizzare gli *RwLock* e' stata presa in quanto, considerando diversi tipi di algoritmi, la quantita' di letture e scritture potrebbe essere sbilanciata e di conseguenza *RwLock* potrebbe risultare piu' efficiente.

Ad esempio, la struct per il selection sort e':

```
struct Variables {
    i: RwLock<Hardened<usize>>,
    j: RwLock<Hardened<usize>>,
    N: RwLock<Hardened<usize>>,
    min: RwLock<Hardened<usize>>,
    vec: RwLock<Vec<Hardened<i32>>>
}
```

8.2.2

9 Analizzatore

References

[1] *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).

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

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

[4] 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.

[5] 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.