UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria
Corso di Laurea Triennale in
Ingegneria Informatica
Dipartimento di
Ingegneria dell'Informazione

Modellatore e Analizzatore di Stochastic Markovian Fault Tree

Mistretta Marco
Matricola 7005065
Casciaro Emanuele
Matricola 7012347

Anno accademico 2020-2021

SOMMARIO

INTRODUZIONE

1.1 Brevi Accenni Teorici

Uno *Stochastic Markovian Fault Tree* (*SMFT*) è un albero che rappresenta le condizioni sotto cui i fallimenti di un insieme di componenti si propagano, dalle foglie alla radice, realizzando le possibili cause di fallimento del sistema. Tale modellizzazione ci permette dunque di valutare il funzionamento dell'intera macchina osservando il solo stato di funzionamento del nodo radice. Ciascun nodo prende il nome generico di *Event*; ciascuna foglia, detta *Basic Event*, rappresenta lo stato di fallimento di una singola componente del sistema; i nodi intermedi, detti *Gates*, rappresentano la condizione di propagazione di guasto all'interno dell'albero. Questi ultimi si classificano in due sottocategorie: gates statici e dinamici, a seconda che la funzione di propagazione dipenda o meno dall'ordine in cui le componenti figlie abbiamo propagato il guasto. In ultimo il nodo radice prende il nome di *Top Event*.

Un *Event* (*Basic Event* o *Intermediate Event*) può presentarsi in ingresso a più Intermediate Event simultaneamente, questa condizione prende il nome di *repeated Event.*

1.2 Motivazioni e Contenuti

L'obbiettivo di questo progetto era quello di modellizzare uno *SMFT* generico nel quale:

- i *Basic Event* alternano stati di funzionamento e malfunzionamento secondo un semplice modello di *Gilber Elliot* (i.e. ad intertempo esponenziale),
- possono essere presenti *repeated Event*
- i Gate possono essere di natura sia statica che dinamica.

Tutto ciò con l'obbiettivo di:

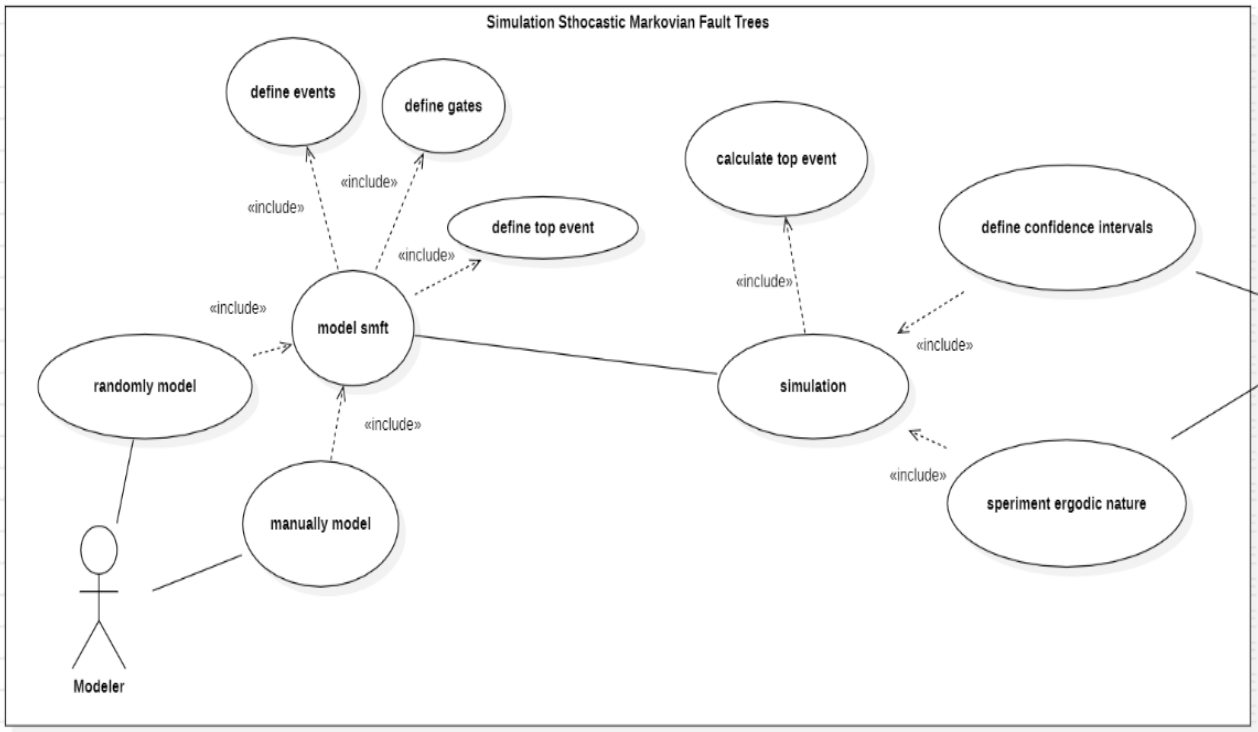
- definire gli indici di confidenza sul valore medio dello stato di funzionamento del sistema (e.g. tramite test di *Student*)
- sperimentare la natura ergodica del modello.

Tutto ciò attraverso la possibilità di simulare l'esecuzione per un dato stato di inizializzazione più e più volte e in condizioni atte a raggiungere lo scopo.

Nota Bene: in fase di progettazione abbiamo deciso di utilizzare la convenzione nella quale ad un nodo funzionante è assegnato un valore di stato di funzionamento booleano vero, ad un nodo malfunzionante invece un valore booleano falso (i.e. un *Gate* AND propagherà lo stato di funzionamento se e solo se tutto le sue sub-componenti funzionano).

1. ANALISI DEI REQUISITI

La prima fase di progettazione si è basata sulla stesura dello Use Case Diagram (figura 2.1)



2.1 Use Case Diagram

Sono stati identificati due Attori fondamentali: *Modeler* e *Analyst*.

Nell'ottica di come è stato ideato il programma, il *Modeler* rappresenta l'utente con esperienza di modellizzazione di *SMFT* che si occuperà appunto, tramite il programma, della fase di modellizzazione dell'intera struttura. Fornire un'interfaccia che permetteva tale libertà di modellazione non era però tra gli obbiettivi del programma, per questo motivo Modeler appartiene al dominio logico del programma stesso. I modelli da esso generati vengono forniti ad Analyst.

Analyst, infatti, rappresenta un utente con sufficienti conoscenze di analisi e studio di *SMFT* il quale, tramite il programma, sarà in grado di valutare la natura ergodica del sistema e calcolare gli indici di confidenza sul valore medio, istante per istante, della *reliability* del sistema.

Per maggiori informazioni, ed ai fini di semplificare la spiegazione e motivazione successiva dei metodi utilizzati, analizziamo il diagramma più nel dettaglio.

2.1.1 Modeler

Come abbiamo detto, il *Modeler* ha il principale caso d'uso di modellizzare l'intera struttura. Per fare questo può quindi definire nuovi Basic Event, e sulla base degli Event già definiti può dunque andare a definire nuovi Intermediate Event (notare che non esistono Intermediate Event senza figli, un Event senza figli è necessariamente una foglia, dunque è un Basic Event per mera definizione).

La definizione di un nuovo *Basic Event*, come quella di un Intermediate Event, può essere effettuata in modo, per così dire, manuale oppure randomico.

- In modo manuale: scegliendo dunque quali valori dare rispettivamente alle due caratteristiche delle due esponenziali che caratterizzano l'intertempo di un Basic Event, e scegliendo da quale dei due stati (funzionamento o malfunzionamento) questo debba partire.
- In modo randomico: lasciando ad un generatore pseudo-casuale l'onore di scegliere tali valori. Avviso di notazione: si è scelto di chiamare $\boldsymbol{\lambda}$ la caratteristica dell'esponenziale che rappresenta la probabilità di passare dallo stato di funzionamento a quello di malfunzionamento, e $\boldsymbol{\mu}$ la rispettiva controparte.

Per quanto riguarda un Intermediate Event:

- In modo manuale: scegliendo quale porta logica, statica o dinamica si desidera realizzare
- In modo randomico: lasciando ad un generatore pseudo-casuale l'onore di tale scelta.

2.1.2 Analyst

The *Analyst* ha principalmente bisogno di simulare più e più volte l'esecuzione del *SMFT* a partire da inizializzazioni atte a raggiungere i vari scopi rappresentati dai suoi casi d'uso. Si serve per questo motivo dei modelli fornitigli dal *Modeler*.

Per valutare la natura ergodica ha bisogno di simulare più volte il sistema a partire da stati iniziali differenti e verificare, in via d'approssimazione, che dopo un tempo sufficientemente lungo, tutte le simulazioni si aggirino intorno allo stesso valore percentuale di reliability. Per il calcolo degli indici di confidenza deve invece simulare più volte un sistema ma a partire dal medesimo stato di inizializzazione (funzionamento del sistema o malfunzionamento del sistema) per fornire in modo sperimentale indici che aiutino nella previsione e stima del comportamento di macchine che presentino la stessa struttura e stesso stato di partenza.

2. PROGETTAZIONE

La prima fase della progettazione ha coinvolto la stesura del Class Diagram. Sotto viene riportata la versione più aggiornata di questo ultimo (figura3.1).

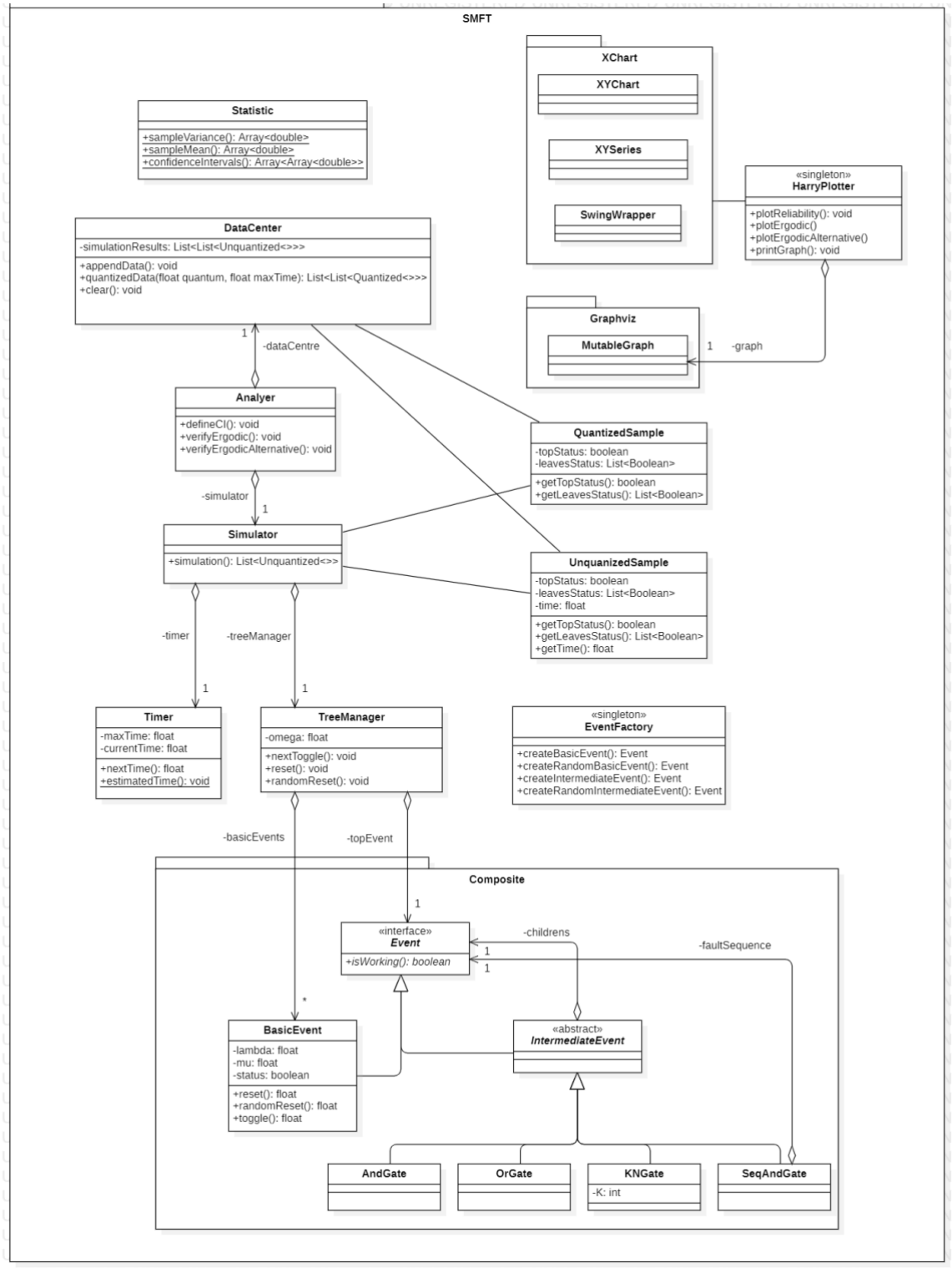
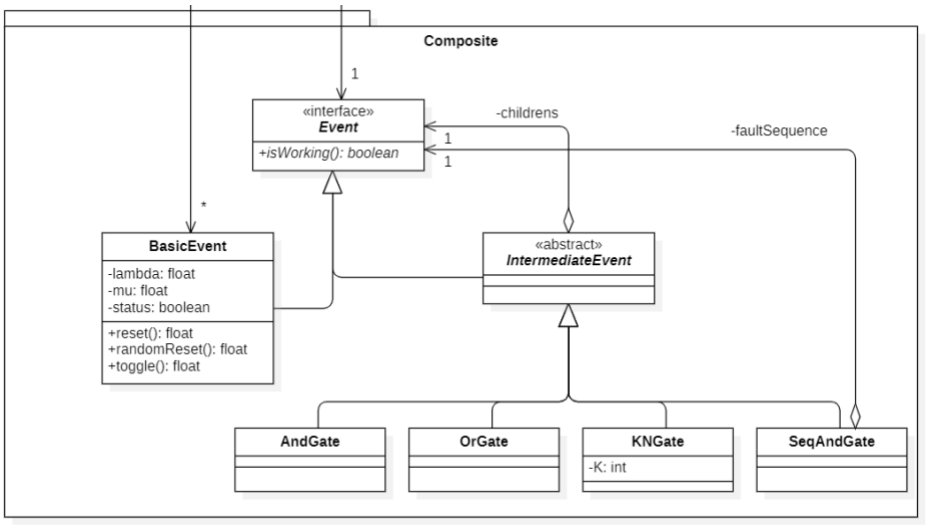


Figura3.1

3.1 Event



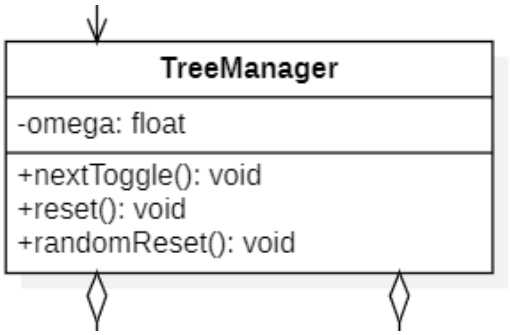
In prima battuta ci siamo concentrati sulle scelte implementative che coinvolgevano la modellazione dell'albero stesso, ossia la modellizzazione di un *Directed Acyclic Graph* (*DAG*). Data la natura del sistema, dove l'utente finale non è tenuto a sapere com'è formato l'albero, ma deve interfacciarsi con esso tramite il Top Event, una soluzione adatta a raggiungere tale obbiettivo era senza dubbio l'implementazione tramite design Pattern Composite. Abbiamo dunque optato per la sua implementazione nella versione in assenza di trasparenza: l'utente medio non ha infatti necessità con questo programma di conoscere il tipo di Event che sta maneggiando (fase di modellazione e inizializzazione esclusa). Ogni Event fornisce un servizio all'Event immediatamente superiore. Per questo motivo ci siamo mossi nella direzione di rappresentare la classe base Event (la nostra *ComponentClass* del pattern) come interfaccia. Essa non poteva senza ombra di dubbio essere una classe astratta per raggiungere tale scopo. Abbiamo dunque deciso di definire al suo interno i soli metodi comuni alle sue due classi figlie: *BasicEvent* e *IntermediateEvent*, rispettivamente la nostra *LeafClass* e *CompositeClass* del nostro Pattern, al semplice scopo di non permettere all'utente di notare differenza di interfaccia nell'utilizzo. Non ci siamo per questo motivo serviti della possibilità di implementare tali metodi nella classe base tramite l'ausilio del modificatore default.

Come si nota dal diagramma l'idea principale era che Event fornisse la dichiarazione del metodo `isWorking()`, del quale sia *IntermediateEvent* che *BasicEvent* avrebbero dovuto fornire l'effettiva e specifica implementazione. Tale metodo se chiamato su Top Event (il nodo radice dell'albero, nonché un particolare *IntermediateEvent* ma senza genitori) avrebbe restituito un booleano (*true* = il Top Event e dunque il sistema funzionano; *false* = il sistema non funziona), risolvendo in modo bottom up la propagazione dell'albero, invocando per prima cosa tale metodo in modo top down su ogni singolo nodo, per ogni livello di profondità.

In *BasicEvent*, aggiunti gli attributi che caratterizzano il modello *Gilbertiano*, abbiamo progettato fondamentalmente di aggiungere altri 3 metodi: `reset()` e `randomReset()` per facilitare la fase di studio dell'albero, setter particolari che servono a ricreare le condizioni necessarie a definire gli intervalli di confidenza ed a valutare la natura ergodica dell'albero; `toggle()` metodo pilastro portante della fase di simulazione e sperimentazione, anch'esso poco più di un setter che avrebbe invertito lo stato di funzionamento di un *BasicEvent* nelle dovute occasioni.

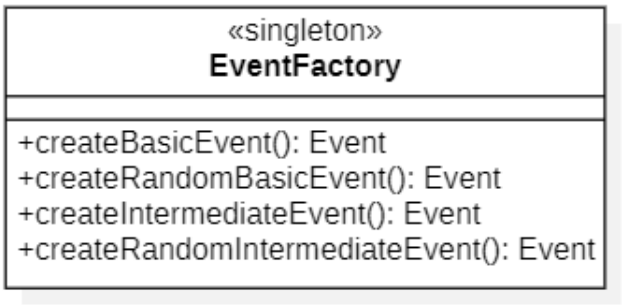
Avevamo tra le mani la struttura di classi che pensavamo meglio avrebbe potuto rappresentare il *SMFT*, il nostro successivo step logico si è mosso in direzione di affrontare la stesura delle classi che avrebbero modellato questo albero e lo avrebbero gestito. La risposta a queste due questioni era semplice: avevamo bisogno di una Classe *EventManager* (nata con lo scopo di gestire l'inizializzazione, operazione ripetuta più volte in fase di analisi e studio del SMFT, e setting dei parametri) e di una classe *EventFactory* (originariamente *EventModeler*) nata con lo scopo di istanziare i Basic Event e gli Intermediate Event.

3.2 TreeManager



Dato che le componenti dell'albero rilevanti all'utente erano i soli Basic Event ed il Top Event, questi dovevano essere i suoi attributi fondamentali, notare la presenta delle due aggregazioni (figura3.1). *TreeManager* doveva consentire di effettuare delle operazioni semplici sul sistema (come ottenere informazioni sullo stesso o manipolare lo stato delle foglie) senza doverci curare della sua composizione e, strutturandola in questo modo, poteva facilmente prendersi carico della selezione del successivo cambiamento di stato di funzionamento di una foglia. La scelta di tenere in memoria, tra i suoi attributi Λ (somma di tutte le caratteristiche di esponenziale attive tra le foglie del modello) ci ha permesso di aggiornare tale valore in tempo costante, evitando così di doverlo ricalcolare ogni volta (complessità lineare rispetto al numero di foglie)

3.3 EventFactory

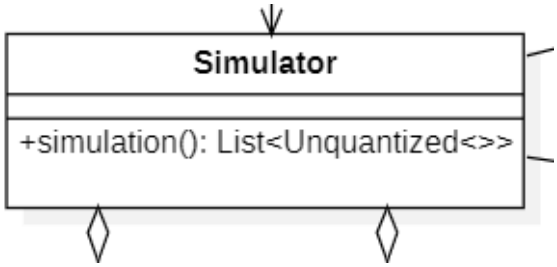


I singoli componenti dell'albero dovevano fornire servizi che non dipendendo dall'ambiente in cui sono creati, pertanto la logica con cui vengono creati poteva essere isolata nella classe *EventFactory*, nata con lo scopo di fornire la possibilità di generare degli Event in modo casuale o deterministico (specificandone i parametri se si desidera creare un *BasicEvent* oppure il tipo se si desidera creare un *IntermediateEvent*). Tuttavia, visto che *l'EventFactory* non doveva essere a conoscenza della struttura dell'albero, i figli di un *IntermediateEvent* non potevano essere scelti in modo casuale, ma dovevano essere completamente determinati al momento della creazione dell'evento. Ciò ha reso logico pensare di costruire l'albero in modo bottom-up.

Considerata l'indipendenza dell'EventFactory da un sistema particolare, è diventato noto come una singola istanza dell'EventFactory possa fornire i propri servizi a tutti i Modeler, con una conseguente implementazione come singleton.

Ogni volta che viene creato un nodo, esso viene associato ad un contatore interno dell'Event *Factory*, che tiene traccia del numero di Event creati fino a quel momento; tale numero agisce da ID per ogni nodo, necessario alla libreria *Graphviz* per stampare l'albero, in quanto potrebbero presentarsi ambiguità, nel caso di BasicEvent con stessi parametri o *IntermediateEvent* dello stesso tipo.

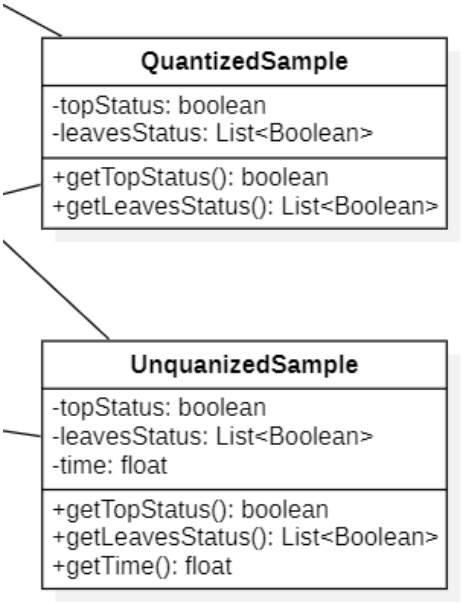
3.4 Simulator



In previsione di progettare un classe che si occupasse della valutazione della natura ergodica, sia del calcolo degli intervalli di confidenza, valutazioni che richiedevano l'esecuzione di una serie di simulazioni del sistema, era necessario ideare una classe che si occupasse di eseguire simulazioni tra loro indipendenti, accomunate tra loro solamente dalla durata, in modo da poter eseguire successivamente la media di tutte le simulazioni. Tali simulazioni sono dunque chiamate singolarmente, portando l'albero in uno stato iniziale opportuno, per poi registrarne le variazioni nel tempo. Questa evoluzione è poi impacchettata in una lista di *UnquantizedSample* e passata al *DataCenter* che si occupa del loro *storing* e della loro quantizzazione in *QuantizedSample* (vedi paragrafo 3.4).

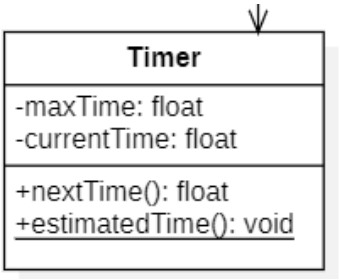
Dato che ogni set di simulazioni viene eseguito sullo stesso albero, e che la loro durata doveva essere identica (pena un'inconsistenza dei dati), a Simulator stata necessaria l'associazione di un *TreeManager* (l'albero) ed un Timer (vedi figura 3.1)

3.5 Unquantized & Quantized Sample



Sono le classi, sopra citate, nato con l'obbiettivo di rappresentare i dati campionati prima e dopo la quantizzazione, e definire una convenzione per conservare i dati. Sono dunque delle classi container, forniscono semplicemente una comoda interfaccia, senza la possibilità di modificare i dati in un istante successivo al campionamento.

3.6 Timer

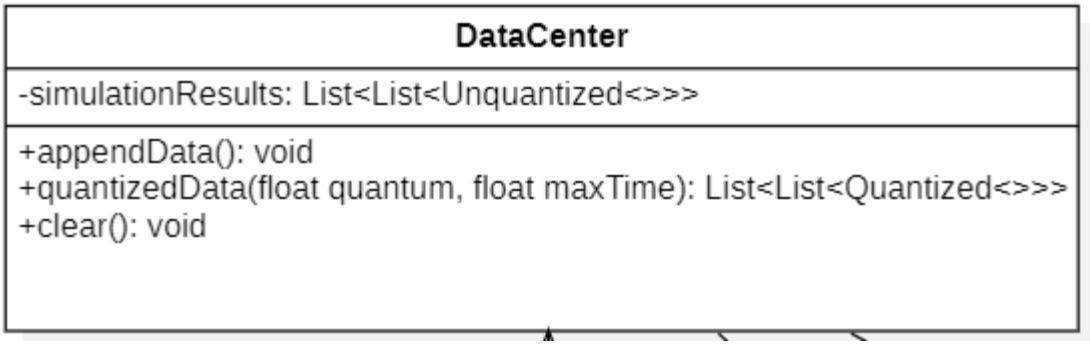


Come già annunciato, avevamo bisogno di una classe Timer progettata per astrarre il concetto di tempo, permettendoci di effettuare una simulazione a durata nominale $\mathbf{t_{max}}$ in tempi trascurabili. Tale compito è svolto conservando informazioni circa l'istante attuale, fornendo un metodo per calcolare l'istante successivo (estrazione da un'esponenziale a caratteristica $\boldsymbol{\Lambda}$), implementato con la formula:

$$t_{n+1} = t_n - \frac{\ln(1 - rand(0,1))}{L}$$

La condizione di fine simulazione è codificata nel tempo stesso: dato che si assume che un tempo non possa mai essere negativo (le simulazioni iniziano convenzionalmente al tempo $t_0 = 0$) al verificarsi di $t_n > t_{max}$ si pone $t_n = -1$. Il simulatore si servirà di tale assunzione come criterio di terminazione.

3.7 DataCenter



I dati restituiti dalle singole simulazioni presi singolarmente non sono significativi, per questo è stata necessaria la definizione di un'entità che si occupi di immagazzinare i risultati di tutte le simulazioni e quantizzarli, a prescindere dal numero di campionamenti delle simulazioni.

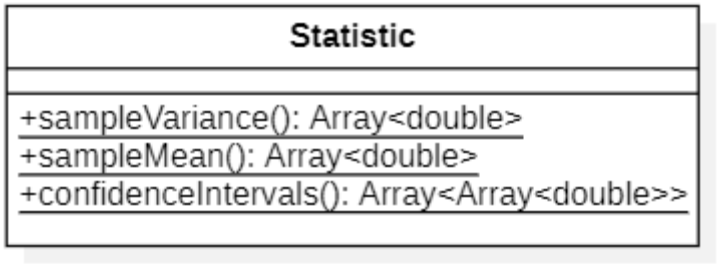
Il *DataCenter* fornisce delle semplici interfacce di gestione dei dati: *appendData*() per inserire un singolo dato, ossia una lista di *UnquantizedData*, che rappresenta una simulazione completa, clear() per cancellare tutti i dati finora sottomessi, *quantizeData*() la più importante, che restituisce un insieme di liste di dati quantizzati.

Dato che la quantizzazione è definita come una funzione continua da destra, i dati sono quantizzati fino all'ultimo istante delle simulazioni, che essendo campionate in tempi casuali, potrebbero avere un numero di quanti differenti, ed i quanti mancanti vengono completati con i dati dell'ultimo campionamento; il numero di quanti da aggiungere è determinato tramite $\mathbf{t_{max}}$, che grazie all'ipotesi di proporzionalità tra il quanto e $\mathbf{t_{max}}$, ci permette di conoscere a priori il numero di quanti previsti.

Era possibile, inoltre, rendere il quanto un parametro del *DataCenter*, permettendo di quantizzare i dati direttamente al momento dell'immissione, scelta che però avrebbe reso l'implementazione troppo dipendente dal modello adottato, in quanto $\mathbf{t_{max}}$, e di conseguenza il quanto, devono essere scelti in base alle caratteristiche dell'albero. Allora l'abbiamo reso un argomento del metodo di quantizzazione, rendendolo allora indipendente dal sistema stesso.

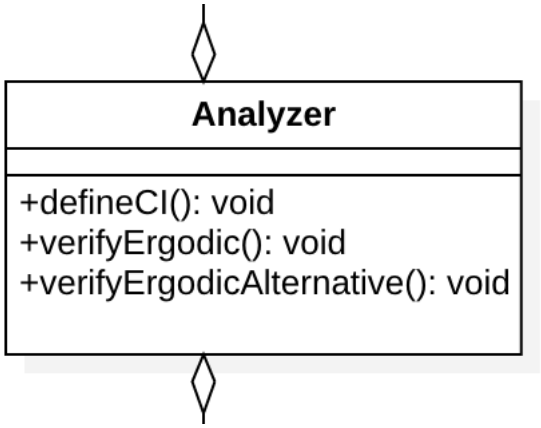
L'ipotesi di quantizzazione ci consente di avere in uscita una struttura dati più compatta, eliminando le informazioni temporali; ciò è possibile in quanto il tempo può essere ricavato dalla sua posizione nella lista, che, moltiplicato per il quanto scelto, fornsice l'istante temporale desiderato.

3.7 Statistic



La definizione degli indici di confidenza, impacchettati e quantizzati i dati, si riduce ad un semplice calcolo di natura matematica, per questo abbiamo definito *Statistic*, una classe con metodi esclusivamente statici, che dati in ingresso campione e significatività restituisce l'intervallo di confidenza desiderato, istante per istante, dell'intera esecuzione.

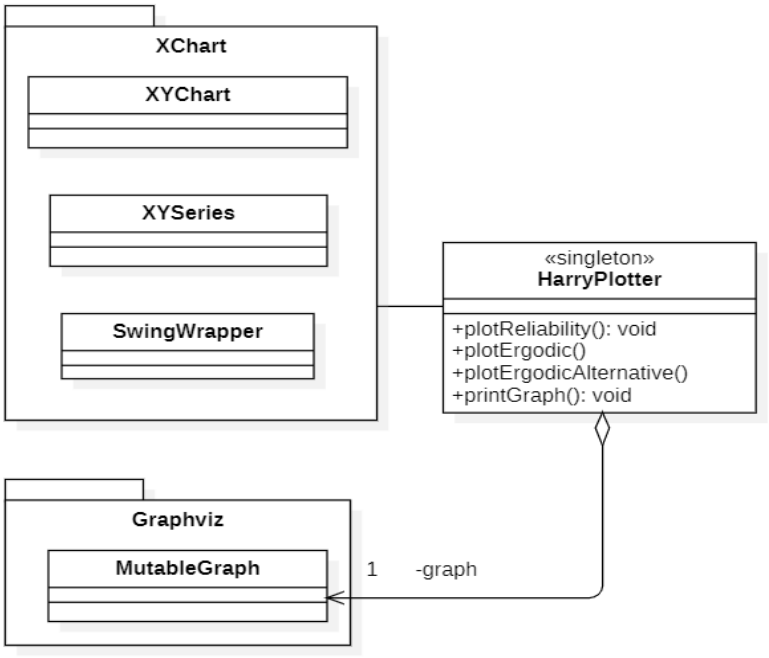
3.8 Analyzer



Siamo così arrivati alla progettazione della classe pilastro dell'analisi dello *SMFT*. Analyzer è stata infatti progettata infatti con l'intento di fornire una semplice interfaccia di utilizzo per i due principali casi d'uso di un Analyst: definire gli indici di confidenza tramite *defineCI*(), e valutare la natura ergodica del sistema tramite *verifyErgodic*(). A tale scopo Analyzer si serve di funzioni ausiliarie interne a lei stessa e dei metodi statici di *Statistic* (paragrafo 3.5).

Per maggiori dettagli rimandiamo al Capitolo 4 di Documentazione del Codice.

3.9 HarryPlotter



Dopo aver modellato un albero ed eseguito le analisi desiderate, è utile avere un modo di visualizzare l'output in modo più intuitivo e poter visualizzare l'albero stesso, per avere un'idea del sistema che sta venendo simulato. Abbiamo progettato con lo scopo di rendere possibile la stampa la classe HarryPlotter la quale fornisce dei metodi per eseguire il rendering dell'albero e per stampare i grafici, con possibilità di eseguire lo zoom nelle varie aree di interesse.

Essendo un'entità che fornisce un servizio di stampa ad altri elementi del programma, doveva essere slegata dall'elemento chiamante, operando soltanto in funzione dei dati ricevuti in ingresso. Ciò unito alla necessità, da parte di componenti diversi di accedere a tali funzionalità ha rese l'implementazione come singleton un opzione fruibile.

HarryPlotter si appoggia su due librerie esterne: *Graphviz* per il rendering dell'albero e *XChart* per il plot dei grafici.

Esaminiamo, dunque, i vari possibili plot:

- **printGraph**: La costruzione dell'albero avviene gradualmente con la definizione dei nodi, che vengono inseriti nel grafo associato in concomitanza con la loro definizione, usando come nome il loro ID ed un colore per ogni tipo di nodo. Se si inserisce nel modello, e di conseguenza nel grafo da stampare, un IntermediateNode, dato che i figli sono già stati definiti, si aggiungono direttamente anche i suoi archi associati
- **plotReliability:** stampa gli indici confidenza del valore atteso dello stato di funzionamento del sistema a significatività α. Opzionalmente si ha la possibilità di stampare anche la media campionaria dello stato di funzionamento (che sarà logicamente e metà di tale intervallo) e la media campionaria dello stato di malfunzionamento (logicamente pari a 1 meno il valore della media campionaria dello stato di funzionamento).
- **plotErgodic:** stampa semplicemente media campionaria e deviazione standard campionaria dello stato di funzionamento
- **plotErgodicAlternative:** stampa la media campionaria dello stato di funzionamento dei vari (tipicamente 10) modelli studiati.

4 STRUTTURA DEL CODICE

Analizziamo adesso alcune componenti fondamentali del codice, motivando eventualmente alcune scelte funzionali.

4.1 Event

Implementazione di `isWorking()`

Fulcro del pattern composite, ci permette di propagare un eventuale cambio di stato di funzionamento delle foglie fino alla radice. Trascurando l'implementazione di BasicEvent, che corrisponde ad un getter sullo stato, analizziamo l'implementazione dei vari gate.

```
//AND Gate
@Override
public boolean isWorking() {
    boolean result = true;
    for (Event e : getChildren()) {
        if (!e.isWorking()) {
            result = false;
            break;
        }
    }
    return result;
}
```

- Il *gate AND* rappresenta una sotto parte del sistema che funziona se e solo se ogni sua componente è attiva, altrimenti propaga l'errore; allora basterà scorrere tutti i suoi componenti, e propagare l'errore se almeno uno di loro non è attivo.

```
//OR Gate
@Override
public boolean isWorking() {
    boolean result = false;
    for (Event e : getChildren()) {
        if (e.isWorking()) {
            result = true;
            break;
        }
    }
    return result;
}
```

- Il *gate OR*, al contrario del precedente, è operativo se lo è almeno uno dei suoi componenti, quindi si scorrono tutti i suoi componenti, e se almeno uno di loro è attivo l'errore non viene propagato.

```java
//K/N Gate
@Override
public boolean isWorking() {
    boolean result = false;
    int count = 0;
    for (int i = 0; i < getChildren().size(); i++) {
        if (getChildren().get(i).isWorking())
            count++;
        if (count ≥ K) {
            result = true;
            break;
        }
    }
    return result;
}
```

- Il *gate K/N* è un compromesso tra i due precedenti: il componente funzione se almeno K dei suoi N componenti sono attivi. È evidente che questo gate generalizzi i primi due, in quanto con K = N si ha un gate AND e con K = 1 si ha un gate OR, che però hanno un'implementazione dedicata in quanto rende la loro esecuzione più semplice. Si è scelto di usare come attributo del gate solamente K, in quanto N è codificato nella lunghezza dell'array dei sottocomponenti del gate stesso (vale infatti N = *`getChildren().size()`*)

```java
//SEquantialANDGate
@Override
public boolean isWorking() {
    for(Event event: getChildren()) {
        int index = faultSequence.indexOf(event);
        if(event.isWorking() && index ≠ -1)
            faultSequence.remove(index);
        else if(!event.isWorking() && index == -1)
            faultSequence.add(event);
    }

    if(faultSequence.size() ≠ getChildren().size()) return true;
    for(int i = 0; i < getChildren().size(); i++)
        if(getChildren().get(i) ≠ faultSequence.get(i))
            return true;

    return  false;
}
```

- Il *gate Sequential AND* è un particolare tipo di *gate AND*, in cui è rilevante anche l'ordine in cui i componenti falliscono: l'errore, in generale si propaga se i componenti si guastano in un ordine prefissato, e tale ordine può essere posto senza perdere di generalità come l'ordine di definizione dei figli, in quanto l'ordine con cui vengono inseriti è influenzabile. Allora ogni volta che si guasta un componente viene accodato ad una lista, che quando raggiunge la massima lunghezza viene comparata con la lista dei figli, e se l'ordine dei componenti coincide si propaga l'errore. Tale lista viene aggiornata al più una volta per iterazione, in quanto tra un campionamento e l'altro cambia solo lo stato di un componente, che può anche non essere connesso al gate. Viene dunque aggiornata (se necessario) tale lista, per poi paragonarla alla lista dei figli.

4.2 Simulator

Implementazione di `simulation()`

Il processo di simulazione è piuttosto semplice, dato che la logica più complessa è fornita da altre entità; ad ogni istante si eseguire il toggle di una foglia casuale e si registra lo stato del sistema in tale momento, infine si calcola il prossimo istante temporale da campionare, che è in funzione di Λ. Il processo continua fino a che il Timer non restituisce il valore di terminazione, definito come un numero negativo. Infine, si restituisce i vari campionamenti raccolti, che andranno poi al DataCenter.

```
public ArrayList<UnquantizedSample> simulation() {
    ArrayList<UnquantizedSample> simResult = new ArrayList<>();
    timer.reset();
    simResult.add(new UnquantizedSample
            (timer.getTime(),
                    treeManager.getTopEvent().isWorking(),
                    treeManager.getStatus()));
    timer.nextTime(treeManager.getOmega());
    while (timer.getTime() ≥ 0) {
        treeManager.nextToggle();
        simResult.add(new UnquantizedSample(timer.getTime(),
                treeManager.getTopEvent().isWorking(),
                treeManager.getStatus()));
        timer.nextTime(treeManager.getOmega());
    }
    return simResult;
}
```

4.3 DataCenter

Implementazione di `quantizedData()`

```
public ArrayList<ArrayList<QuantizedSample>> quantizedData(float quantum, float maxTime) {

    ArrayList<ArrayList<QuantizedSample>> quantizedResults = new ArrayList<>();
    int N = simulationResults.size();
    int numberOfSamples = (int) (maxTime / quantum) + 1;
    double start = System.currentTimeMillis();
    for(int i = 0; i < N; i++) {
        Timer.estimatedTime(N, start, i,  action: "Quantizzazione dei risultati");

        quantizedResults.add(new ArrayList<>());
        for (float step = 0f; step ≤ maxTime; step += quantum) {
            for (UnquantizedSample data : simulationResults.get(i)) {
                if (data.getTime() ≥ step) {
                    quantizedResults.get(quantizedResults.size() - 1).add(new QuantizedSample(data.getTopStatus(),
                            data.getLeavesStatus()));
                    break;
                }
            }
        }
        int l = numberOfSamples - quantizedResults.get(quantizedResults.size() - 1).size();
        while (l > 0) {
            quantizedResults.get(quantizedResults.size() - 1).add(new QuantizedSample(simulationResults.get(i).get(
                    simulationResults.get(i).size() - 1).getTopStatus(),
                    simulationResults.get(i).get(simulationResults.get(i).size() - 1).getLeavesStatus()));
            l--;
        }
    }
    return quantizedResults;
}
```

Ad ogni simulazione si associa una lista di dati campionati, di lunghezza $\frac{t_{max}}{q}$ dove il valore in ogni quanto corrisponde al campionamento al tempo subito successivo al quanto di riferimento. Gli eventuali quanti rimasti senza valore vengono completati al valore dell'ultimo quanto aggiunto; ciò è possibile grazie al fatto di conoscere già a priori il numero totale di quanti.

4.4 Analyzer

Implementazione di `defineCI()`

```
public void defineCI(int N, float alpha, float quantum, boolean meanPlot, boolean fault) {
    dc.clear();
    double start = System.currentTimeMillis();
    for (int i = 0; i < N; i++) {
        Timer.estimatedTime(N, start, i,  action: "Simulazioni per calcolo Intervalli di Confidenza");
        sim.getTreeManager().reset();
        dc.appendData(sim.simulation());
    }
    ArrayList<ArrayList<QuantizedSample>> quantizedResults =
            dc.quantizedData(quantum, sim.getMaxTime());
    double[] sampleMean = Statistic.sampleMean(quantizedResults);
    double[][] CI = Statistic.confidenceInterval(quantizedResults, alpha, sampleMean);
    int l = CI[0].length;

    double[] times = new double[l];
    for (int i = 0; i < l; i++) {
        times[i] = i * quantum;
    }
    HarryPlotter.getInstance().plotReliability(times, CI, sampleMean, meanPlot, fault);
}
```

Allo scopo di definire gli intervalli di confidenza devono prima essere specificati 2 parametri:

- N: numero di run che devono essere eseguite dal modello
- alpha: significatività voluta

Il metodo si occupa dunque di chiamare N volte `simulation()`, inizializzando ogni folta le foglie dell'albero tramite `reset()` allo stato di totale funzionamento, e passare ripetutamente il pacchetto dati a *DataCenter* che si occuperà della loro quantizzazione. La struttura quantizzata prende il nome di *quantizedResults* e questa ultima viene passata insieme ad alpha a *Statistic* che banalmente si occupa del calcolo degli indici di confidenza. Il metodo si occupa poi di invocare `plotReliability()` di *HarryPlotter* per richiamare la stampa del grafico. I due booleani meanPlot e fault richiedono semplicemente la stampa opzionale rispettivamente di media campionaria di funzionamento e malfunzionamento come già annunciato.

Implementazione di `verifyErgodic()`

```
public void verifyErgodic(int N, float quantum, double meanPrecision, double varPrecision) {
    dc.clear();

    double start = System.currentTimeMillis();
    for (int i = 0; i < N; i++) {
        Timer.estimatedTime(N, start, i,  action: "Simulazioni per verifica Ergodicità");
        sim.getTreeManager().randomReset();
        dc.appendData(sim.simulation());
    }

    ArrayList<ArrayList<QuantizedSample>> quantizedResults =
            dc.quantizedData(quantum, sim.getMaxTime());
    double[] sampleMean;
    double[] sampleStandardDeviation;
    double[] times;

    sampleMean = Statistic.sampleMean(quantizedResults);
    sampleStandardDeviation = Statistic.sampleStandardDeviation(quantizedResults, sampleMean);
    times = new double[sampleStandardDeviation.length];
    for (int i = 0; i < sampleMean.length; i++)
        times[i] = i * quantum;
    HarryPlotter.getInstance().plotErgodic(times, sampleMean, sampleStandardDeviation);
    findConvergency(times, sampleMean, sampleStandardDeviation, meanPrecision, varPrecision);

    verifyErgodicAlternative(N, quantum);

}
```

L'implementazione di questo metodo è stata probabilmente quella che ha richiesto più ragionamento.

Non ci soffermeremo sulla documentazione della prima parte, analoga quella di *defineCI()* fatta eccezione che ogni simulazione avviene con una inizializzazione delle foglie randomica tramite *randomReset()*. Ci soffermiamo invece sul suo metodo logico ausiliario *findConvergency()*.

```
private void findConvergency(double[] times, double[] sampleMean, double[] sampleStandardDeviation,
                             double meanPrecision, double varPrecision) {
    double min = 1.1;
    double max = -0.1;
    double start = times[times.length - 1];
    int count = times.length / 10;
    for (int i = times.length - 1; i ⩾ 0; i--) {
        if (sampleStandardDeviation[i] > varPrecision && count > 0) {
            System.err.println("sistema probabilmente non ergodico, deviazione standard campionaria  maggiore di "
                    + varPrecision);
            return;
        }
        if (sampleMean[i] > max) max = sampleMean[i];
        if (sampleMean[i] < min) min = sampleMean[i];
        if ((max - min) > meanPrecision && count > 0) {
            System.err.println(
                    "sistema probabilmente non ergodico, media campionaria non sufficientemente costante");
            return;
        }
        if (sampleStandardDeviation[i] > varPrecision && count ⩽ 0) {
            System.err.println(
                    "Verifica ergodicità arrestata per valore di deviazione standar campionaria maggiore di "
                            + varPrecision);
            break;
        }
        if ((max - min) > meanPrecision && count ⩽ 0) {
            System.err.println(
                    "Verifica ergodicità arrestata per valore di media campionaria non sufficientemente costante");
            break;
        }
        else {
            start = times[i];
            count--;
        }
    }
    System.err.println("Il sistema mostra la sua natura ergodica a partire dall'istante: " + start);
}
```

Dopo attenti ragionamenti abbiamo infatti pensato di fornire un metodo innovativo alla sperimentazione e verifica della natura ergodica di un sistema. Istante per istante, a partire da t_{max}, verifichiamo due condizioni (la mancata verifica di almeno una delle due blocca la sperimentazione):

- la deviazione standard campionaria deve essere sotto una certa soglia definita in ingresso che prende il nome di *varPrecision*
- la media campionaria deve approssimare un andamento costante, questo è verificato tramite l'ausilio di una soglia definita in ingresso che prende il nome di *meanPrecision. Si immagazzina in due variabili di appoggio il valore di media campionaria minimo e massimo trovato durante l'analisi, e si verifica ad ogni iterazione che la loro differenza non superi il valore soglia.*

La variabile intera *count* contatore a decremento serve, dunque, a far si che la funzione definisca il sistema ergodico solo se le condizioni sopra citate siano state verificate correttamente in almeno l'ultimo 10% degli istanti di simulazione.

Implementazione di `verifyErgodicAlternative()`

```
public void verifyErgodicAlternative(int N, float quantum) {

    ArrayList<ArrayList<QuantizedSample>>[] quantizedResults = new ArrayList[10];
    ArrayList<Boolean> status = new ArrayList<>();

    double start = System.currentTimeMillis();
    for (int i = 0; i < 10; i++) {
        dc.clear();
        Timer.estimatedTime( N: 10, start, i,  action: "Simulazioni per verifica Ergodicità 2");
        sim.getTreeManager().randomReset();
        status = sim.getTreeManager().getBasicEventsStatus();
        for (int count = 0; count < N/5; count++) {
            sim.getTreeManager().setBasicEventsStatus(status);
            dc.appendData(sim.simulation());
        }
        quantizedResults[i] = dc.quantizedData(quantum, sim.getMaxTime());
    }

    double[][] sampleMeans = new double[10][];
    for (int i = 0; i < 10; i++)
        sampleMeans[i] = Statistic.sampleMean(quantizedResults[i]);

    double[] times = new double[sampleMeans[0].length];
    for (int i = 0; i < sampleMeans[0].length; i++)
        times[i] = i * quantum;

    HarryPlotter.getInstance().plotErgodicAlternative(times, sampleMeans);
}
```

In modo parallelo al metodo sopra documentato abbiamo deciso di appoggiare un ulteriore analisi della natura ergodica del sistema. Questa come la precedente ripete svariate simulazioni a partire da inizializzazioni casuali, ma salvando lo stato delle foglie definito randomicamente riesce a studiare 10 sole permutazioni dello stato delle foglie. Sotto questo punto di vista la randomizzazione è pseudo-randomica e consento di verificare che il sistema sia ergodico ad occhio nudo, osservando che il grafico che invoca chiamando `plotErgodicAlternative()`, il quale stampa le 10 medie campionarie dello stato di funzionamento, convergano, dopo un tempo sufficiente, allo stesso valore.

5 TESTING EMA

Per garantire il funzionamento del package basta garantire che funzioni le sue componenti più nevralgiche, ossia quelle relative alla simulazione e l'elaborazione dei dati.

5.1 Simulator

Il testing della simulazione si riconduce a testare il corretto funzionamento della classe timer e dell'albero stesso. Data la natura composite dell'albero, è sufficiente garantire il corretto funzionamento dei singoli componenti di esso. Provato ciò il corretto funzionamento dell'albero, e quindi, della simulazione, è garantito. Allora si esegue il funzionamento di tutti i vari gate dell'albero, testando i casi in cui è garantito il funzionamento (o il malfunzionamento), ed i casi generali. Il test del Timer avviene assicurando che una il tempo incrementi effettivamente ad ogni iterazione e che restituisca un valore negativo una volta superato $\mathbf{t_{max}}$.

5.2 Analyzer

L'Analyzer si occupa di processare i dati delle simulazioni, quindi una volta garantito il funzionamento del simulatore e sufficiente dimostrare che le funzioni matematiche usate restituiscano i risultati corretti.

I calcoli fanno uso delle medie e deviazioni standard campionarie, quindi il loro calcolo deve essere corretto. Si testa allora il calcolo di esse su un dataset di modelli sempre funzionanti (a stato logico 1), aspettandoci allora $\boldsymbol{\mu = 1},\ \boldsymbol{\sigma = 0}$

6. MOCKUPS

Figura 1: avvio applicazione

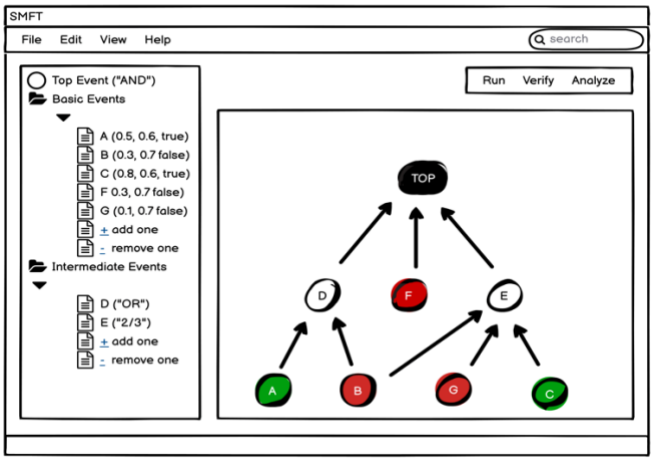


Figura 2: fase di modellazione

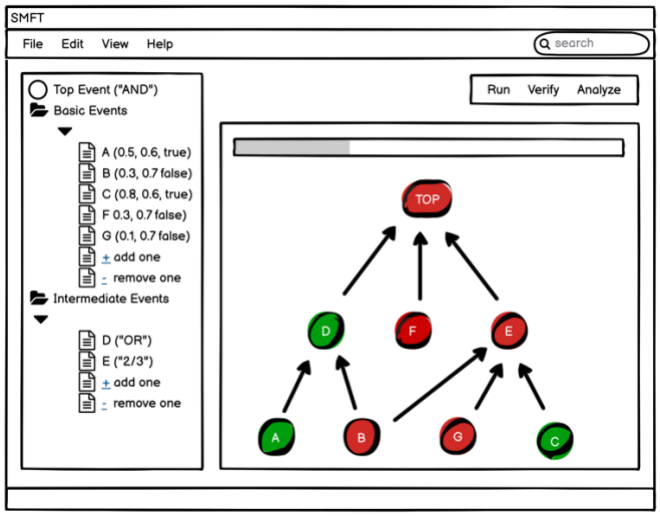


Figura 3: simulazione

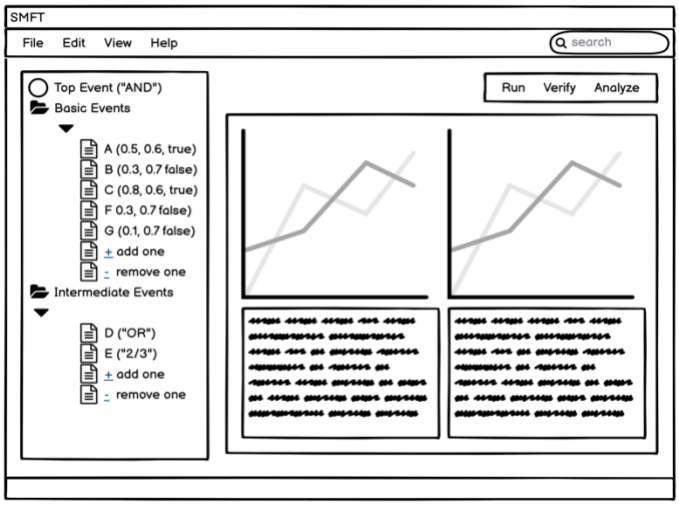


Figura 4: stampa intervalli di confidenza

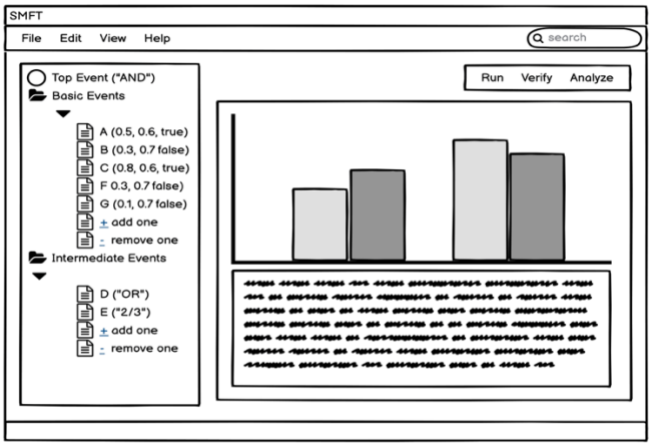


Figura 5: risultati sperimentazione natura ergodica

6 OSSERVAZIONI E CONCLUSIONI

Tra gli aspetti da migliorare c'è senza ombra di dubbio la gestione dei pacchetti dati per alberi di grandi dimensioni ma in conclusione ci possiamo considerare davvero soddisfatti del lavoro svolto. In tempi brevi il nostro package è in grado di fornire risultati relativamente precisi per SMFT di piccole dimensioni. I grafici forniti sono chiari ed evidenti, e la funzione di sperimentazione di natura ergodica sembra restituire risultati davvero attendibili e semplifica il lavoro che sarebbe dovuto svolgere da un occhio vigile umano. La modellazione funziona bene e implementando un'interfaccia grafica (GUI) il tutto sarebbe ancora più immediato e alla portata di tutti (vedi Capitolo 6 Mockups).