POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Gestionale Classe L-8

Tesi di Laurea Triennale

SOFTWARE PER SCHEDULAZIONE E SIMULAZIONE DI NO REGRESSION TEST

Relatore

prof. Fulvio Corno

Candidato

Federico Olivero

Anno 2023

INDICE

PROPOSTA DI PROGETTO

STUDENTE PROPONENTE

s282318 Olivero Federico

TITOLO DELLA PROPOSTA

Software per schedulazione e simulazione di No Regression Test

DESCRIZIONE DEL PROBLEMA PROPOSTO

Il software si propone di risolvere un problema reale riscontrato durante lo svolgimento del tirocinio in azienda. L'attività (chiamata Correttiva) prevede l'esecuzione di No Regression Test da lanciare su diverse macchine virtuali, periodicamente (una volta a settimana) in un arco di tempo ben preciso, che hanno come obiettivo il testing funzionale di un applicativo client-server. Durante le settimane è stata riscontrata un'elevata instabilità nello svolgimento dei test (dovuta a diversi fattori, quali l'instabilità dell'applicativo, degli script, ed altri) che talvolta ha come conseguenza il mancato svolgimento di tutti i test nell'arco di tempo previsto. Il software propone la risoluzione di questi problemi tramite l'ottimizzazione della schedulazione dei test, e una simulazione di eventi che permette di avere una panoramica generale della Correttiva nei giorni precedenti, per permettere agli sviluppatori/tester di apportare le modifiche necessarie per raggiungere la stabilità richiesta.

DESCRIZIONE DELLA RILEVANZA GESTIONALE DEL PROBLEMA

I problemi che il software intende risolvere hanno rilevanza gestionale, in quanto consistono in problemi di ottimizzazione e schedulazione, concetti inerenti alla ricerca operativa e alla progettazione/gestione di sistemi aziendali. Inoltre, la simulazione di eventi che il software andrà a svolgere dipenderà da alcuni parametri in ingresso che un ipotetico Ingegnere Gestionale potrà impostare: al variare dei parametri in ingresso la simulazione produrrà diversi risultati, che potranno essere analizzati per decidere come agire sul progetto per ottimizzarlo.

DESCRIZIONE DEI DATA-SET PER LA VALUTAZIONE

I dati per lo svolgimento del progetto sono stati raccolti nel contesto aziendale, sotto forma di numerosi file csv. Il database utilizzato nel progetto è stato da me costruito, dopo una generale "pulizia" dei file, eliminando eventuali incongruenze e raccogliendo i dati dell'esecuzione dei vari test durante le Correttive degli anni passati.

Il dataset contiene informazioni sui diversi test e le sue esecuzioni nel tempo: ogni test, detto Istanza, appartiene a una specifica compagnia, prevede un determinato scenario assicurativo (ad esempio Emissione o Riscatto di una polizza) ed è eseguito su un determinato prodotto. Per ogni test sono state raccolte tutte le esecuzioni negli anni passati, sulle diverse macchine virtuali.

Il database si suddivide in 2 tabelle:

- **Tabella 1: Istanze**

Questa tabella contiene le informazioni su oltre 50 istanze. Ogni istanza ha Id, Scenario, Compagnia, Prodotto e talvolta un informazione aggiuntiva per distinguere due istanze con stesso scenario, compagnia e

prodotto.

- **Tabella 2: Esecuzioni**

Questa tabella contiene tutte le informazioni sulle varie esecuzioni di ogni istanza negli ultimi anni. Ogni esecuzione di un'istanza è stata svolta su una determinata macchina virtuale in una certa data e in un preciso orario, ha una durata, uno stato (successo o fallimento) e un numero di errori e warnings.

Nota: per motivi di privacy alcuni dati, come nome delle compagnie e dei prodotti, sono stati cifrati. Questo non impatta in alcun modo la realisticità del progetto.

DESCRIZIONE PRELIMINARE DEGLI ALGORITMI COINVOLTI

Il progetto prevede l'implementazione di 2 principali algoritmi:

- Nel **primo caso** si tratta di un algoritmo *ricorsivo*, che ha come obiettivo l'ottimizzazione della schedulazione delle varie istanze. Ricordando che uno dei problemi principali che il software mira a risolvere è l'esecuzione di tutti i test all'interno dell'arco di tempo previsto, l'algoritmo prenderà infatti in input una lista contenente tutte le istanze previste (=test da svolgere) e determinerà su quale macchina virtuale il test in questione va lanciato, in modo da minimizzare la massima durata dei test su singola macchina: se infatti la macchina che ci impiegherà più tempo a svolgere tutti i test che le sono stati "assegnati" riesce a rimanere nell'arco di tempo predeterminato, l'obiettivo risulta essere raggiunto. L'algoritmo terrà in considerazione determinati fattori, quali:
    - conflitto tra determinate istanze se eseguite in parallelo (sovrapposizione temporale delle esecuzioni) in diversi casi (predeterminati), come istanze con stesso prodotto, istanze con stesso scenario e compagnia, ecc..
    - condizioni di stabilità impostate prima di effettuare la simulazione: i parametri impostati dall'utente determineranno una particolare situazione di instabilità, che si può tradurre nell'implementazione di ulteriori specifici vincoli. Esempio: una situazione particolarmente instabile può essere dovuta alle critiche condizioni pre-correttiva; infatti alcune istanze possono essere eseguite soltanto se nella cartella condivisa sono presenti file emessi tramite l'esecuzione di alcune precise istanze (è il caso dei Riscatti che necessitano di file emessi dalle Emissioni).
- Nel **secondo caso** si tratta di un algoritmo di *simulazione di eventi*, che data la schedulazione fornita dall'algoritmo al punto 1, si occupa di simulare lo svolgimento della correttiva, ritornando tutti gli eventi in ordine temporale dall'orario di inizio della correttiva fino all'orario di fine. Gli eventi principali sono 3:
    - Inizio esecuzione: prevede l'inizio dell'esecuzione dell'istanza in questione, determinando in base alle condizioni di stabilità/instabilità impostate dall'utente se il test avrà successo o meno, schedulando l'evento. I parametri impostati dall'utente verranno elaborati e convertiti in probabilità di successo o fallimento del test.
    - Successo esecuzione: prevede il caso in cui l'evento al punto 1 abbia previsto il successo del test. In questo caso l'unica cosa da fare è schedulare un nuovo inizio di esecuzione
    - Fallimento esecuzione: prevede il caso in cui l'evento al punto 1 abbia previsto il fallimento del test. In questo caso il test in questione va rischedulato.

DESCRIZIONE PRELIMINARE DELLE FUNZIONALITÀ PREVISTE PER L'APPLICAZIONE SOFTWARE

L'interfaccia grafica dell'applicazione avrà 2 differenti sezioni:

- La prima dedicata all'impostazione dei parametri per l'utente: qui chi utilizzerà l'applicazione potrà selezionare i parametri che meglio descrivono la situazione in cui si trova. Per ottimizzare l'esperienza utente verranno proposti degli intervalli che permetteranno all'ingegnere gestionale che effettuerà la simulazione di esprimere le condizioni di partenza.
- La seconda dedicata allo svolgimento della simulazione: qui verrà stampata a schermo l'intera simulazione, in modo tale che l'utente possa fare le analisi necessarie e trarre conclusioni, oltre a un resoconto complessivo determinato dall'applicazione stessa (riguardante il raggiungimento dell'obiettivo e alcuni dati importanti come numero di fallimenti e prestazioni di ogni macchina)

PROBLEMA AFFRONTATO

Come anticipato, il problema è sorto durante l'esperienza di tirocinio. La situazione di partenza è la seguente: durante la Correttiva dei test, vanno eseguite oltre 50 diverse istanze , su 5 diverse *virtual machines* ; ogni istanza corrisponde a una navigazione da fare sull'applicativo, che mira a testare che alcune funzionalità siano svolte correttamente (successo del test) e in caso contrario rilevare gli errori (fallimento del test). Il problema riscontrato riguarda la dinamica per cui alcune istanze, se eseguite in parallelo su due macchine diverse, possono andare in conflitto, ovvero potrebbero verificarsi fallimenti di test nonostante l'applicativo funzioni correttamente. Questa dinamica risulta essere problematica per gli sviluppatori, in quanto non solo devono spendere del tempo per capire l'origine dell'errore presentatosi (poichè se il bug riguarda l'applicativo va segnalato, altrimenti no), ma anche perchè comporta una notevole perdita di tempo che può risultare significativa per lo svolgimento dell'intero processo di Correttiva nei tempi prestabiliti. Il software si propone di risolvere questo problema, effettuando una schedulazione dei test sulle varie macchine, evitando i conflitti (predeterminati), e ottimizzando la distribuzione delle istanze per rimanere all'interno dell'intervallo di tempo richiesto. Inotre prevede la possibilità di eseguire una simulazione della Correttiva, per evidenziare eventuali criticità, utili agli sviluppatori per capire dove intervenire a codice nei giorni prima del lancio ufficiale, in modo da correggere gli script delle navigazioni e raggiungere una maggiore stabilità.

DATA-SET

Il data-set su cui si basa il progetto è stato da me costruito, con il seguente processo: i dati sono stati raccolti nel contesto aziendale sotto forma di file *.xlsx*, per poi essere convertiti in formato *.csv.* Successivamente, dopo aver pulito i dati, è stato creato il database *mysql* su cui si basa il progetto.

RACCOLTA DELLE INFORMAZIONI

I dati sono stati raccolti per istanza. Ogni istanza aveva quindi il proprio file *.xlsx* con l'elenco di tutte le esecuzioni negli ultimi anni. Ogni esecuzione presentava diverse informazioni, alcune delle quali non necessarie allo svolgimento del progetto, motivo per cui è stata necessaria una pulizia dei dati.

CONVERSIONE DEI FILE

Per lavorare sui dati risultava più comodo che fossero contenuti in un file con separatori, quindi ho convertito gli *.xlsx* in *csv* tramite il seguente script VBS:

```
' Dichiarazione delle variabili
Dim objFSO, objFolder, objExcel, objFile, objSheet
Dim strFolderPath, strCSVFolder

' Specifica la cartella di origine dei file XLSX
strFolderPath = "pathCartellaXLSX"

' Specifica la cartella di destinazione dei file CSV
strCSVFolder = "pathCartellaCSV"

' Crea oggetti FileSystem e Excel
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objExcel = CreateObject("Excel.Application")

' Crea la cartella di destinazione se non esiste
If Not objFSO.FolderExists(strCSVFolder) Then
    objFSO.CreateFolder strCSVFolder
End If

' Ottieni la lista dei file XLSX nella cartella di origine
Set objFolder = objFSO.GetFolder(strFolderPath)

' Itera attraverso i file nella cartella di origine
For Each objFile In objFolder.Files
    If LCase(objFSO.GetExtensionName(objFile.Path)) = "xlsx" Then
        ' Apri il file XLSX
        Set objSheet = objExcel.Workbooks.Open(objFile.Path).Sheets(1)

        ' Specifica il nome del file CSV di destinazione
        strCSVFile = objFSO.BuildPath(strCSVFolder, objFSO.GetBaseName(objFile) & ".csv")

        ' Salva il file XLSX come CSV
        objSheet.SaveAs strCSVFile, 6 ' 6 corrisponde a xlCSV (valore numerico per CSV)

        ' Chiudi il file XLSX
```

```
        objSheet.Parent.Close False
    End If
Next

' Chiudi l'applicazione Excel
objExcel.Quit

WScript.Echo "Conversione completata."

' Fine dello script
```

PULIZIA DEI DATI

Una volta convertiti i file in *.csv* era necessaria una pulizia per due motivi:

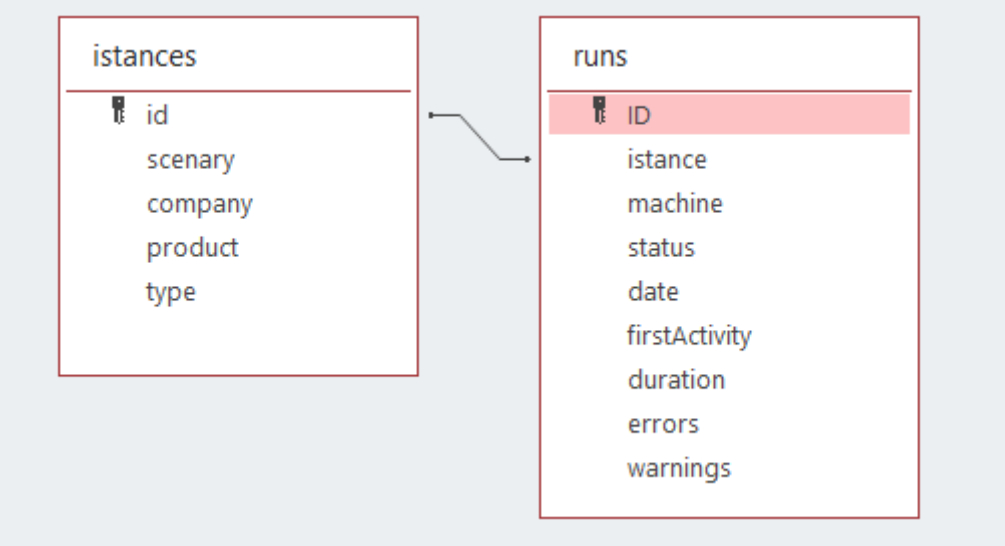
- 1. Gli excel di partenza contenevano informazioni non necessarie allo svolgimento del progetto, che quindi potevano essere non trascritte nel database.
- 2. Alcune run presentavano informazioni mancanti, e sono state quindi escluse nella costruzione del file *.sql*.

La pulizia dei dati è stata fatta tramite un programma in *Java*, che si occupa di creare un oggetto di tipo *Istanza* per ogni file *csv* letto, aggiungendo gli attributi ricavati dal nome del file (il nome di ogni file conteneva informazioni su scenario, compagnia e prodotto), e un oggetto di tipo *Run* per ogni riga letta dai file *csv* (gli attributi in questo caso sono ricavati tramite la separazione dei vari campi all'interno della riga, escludendo quelli non necessari). *Osservazione:* sono stati esclusi tutte le righe con anche solo un attributo necessario mancante.

CREAZIONE DEL DATABASE

Il database è stato creato manualmente tramite il software *HeidiSQL*, ed è costituito da 2 tabelle, una relativa alle istanze e una alle esecuzioni. Memorizzati i dati in liste di oggetti *Istanza* e *Run* , sono stati aggiunti alle tabelle tramite pattern **DAO** (Data Access Object).

Di seguito tabelle e relazioni:



STRUTTURE DATI E ALGORITMI

Il progetto è stato sviluppato utilizzando i pattern **MVC** (Model View Controller) e **DAO** (Data Acces Object), in modo da tenere separate logica applicativa, struttura dati e interfaccia grafica dell'applicazione. La parte grafica è stata realizzata con JavaFX e SceneBuilder, un editor visuale che consente la creazione di file fxml. Il collegamento con la logica applicativa avviene tramite metodi presenti nella classe *FXMLController* (pattern MVC) che generano un oggetto di classe *Model* che contiene variabili e metodi relativi alla logica applicativa. I dati necessari al funzionamento dell'applicativo sono ricavati dal database con funzioni delle classi presenti nel package *.db* (pattern DAO).

PATTERN MVC

I metodi che permettono le due funzioni di avvio dell'applicazione e interazione tra l'utente e la logica applicativa sono contenuti nel package *it.polito.tdp.provaFinale*. Rispettivamente le classi *EntryPoint* e *Main* svolgono la prima funzione, mentre la classe *FXMLController* contiene tutti gli oggetti che rappresentano i vari pulsanti e campi di testo dell'interfaccia grafica. Ogni elemento di tipo <*Button*> è collegato a un metodo che, al premere del pulsante esegue determinate funzioni, come scrivere a schermo eventuali output, rilevare alcuni valori da elementi come le *CheckBox* e gli *Slider* o richiamare metodi della classe Model, che svolgono la logica applicativa e ritornano informazioni da stampare.

PATTERN DAO

Per ottenere dati presenti nel database vengono utilizzate le classi contenute nel package *it.polito.tdp.provaFinale.db*. Quest'ultimo contiene una classe per connettere l'applicazione al database, *DBConnect*, e una classe *DAO* per tabella: *IstanceDAO* si occupa di interrogare il db per quanto riguarda le informazioni presenti nella tabella *Istances*, *RunDAO* per quelle contenute nella tabella *Runs*.

LOGICA APPLICATIVA

Nel package *it.polito.tdp.provaFinale.model* sono contenute le classi che rappresentano i "protagonisti" dell'applicazione e quelle che svolgono la logica applicativa vera e propria, ovvero schedulazione e simulazione. Per quanto riguarda le prime citate, le principali sono:

- **Istance**: rappresenta le istanze e contiene quindi attributi come scenario, compagnia e prodotto, ricavate dalla tabella *Istances*
- **VirtualMachine**: rappresenta le varie macchine virtuali che devono svolgere i relativi test
- **Run**: rappresenta l'esecuzione di un'istanza su una VirtualMachine

Oltre a queste, figurano le classi *Event*, utilizzata per rappresentare un determinato evento della simulazione, e *RowIstances* utilizzata per riportare in tabella le informazioni della schedulazione e della simulazione.

Infine le classi *Scheduling* e *Simulator* contengono variabili e metodi necessari allo svolgimento rispettivamente di schedulazione delle istanze e simulazione di eventi.

ALGORITMO DI SCHEDULAZIONE

L'algoritmo di schedulazione si occupa di determinare la divisione e l'ordine di esecuzione delle varie istanze sulle 5 diverse virtual machines. L'obiettivo, dato un numero fisso di istanze in input ognuna con propria durata media, è quello di determinare (soddisfando i vari vincoli) la miglior divisione in modo da minimizzare il tempo di lavoro della macchina con più carico: essendo lo scopo iniziale del progetto quello di aiutare gli

sviluppatori a rimanere nell'intervallo di tempo concesso per lo svolgimento di tutti i test, più si minimizza il tempo della macchina con più istanze da testare, più aumentano le probabilità di non sforare nei tempi. *Nota:* si osserva infatti che se la durata della peggiore macchina è inferiore della durata totale, non è garantito lo svolgimento di tutti i test in quanto questi possono fallire durante la simulazione, dinamica che non è preventivabile durante la schedulazione. Una prima implementazione dell'algoritmo ricorsivo mirava a ricercare la soluzione ottima di schedulazione. Tuttavia l'elevato numero di istanze da schedulare e di vincoli da soddisfare richiedeva tempi troppo prolungati per un tale contesto Si è quindi optato per una soluzione non ottima: all'inizio della ricorsione viene impostato un tempo massimo, pari a circa la metà del tempo disponibile totale, e l'esecuzione dell'algoritmo termina non appena viene trovata una soluzione con tempo massimo al di sotto di quello impostato. Il metodo di inizio schedulazione prende in input una lista di istanze da schedulare, crea le variabili necessario allo svolgimento della ricorsione e chiama il metodo *ricerca()*. Quest'ultimo, partendo dalla lista di istanze va alla ricerca di una soluzione che soddisfi i requisiti (tempo della macchina peggiore minore di un valore predeterminato), in forma di lista di liste di istanze. Di seguito l'implementazione dei 2 metodi:

```
/*  METODO DI INIZIO SCHEDULAZIONE
    * @PARAM prende in input una lista di istanze da schedulare e un numero di macchine su cui schedulare
    * avvia la ricorsione
    * @RETURN restituisce la schedulazione sotto forma di lista di liste di istanze
    */
   private List<List<Istance>> schedule(List<Istance> toSchedule){

      //creo un parziale, contenente una lista di istanze per macchina (nMacchine passato in input)
      List<List<Istance>> parziale = new ArrayList<List<Istance>>();
      for(int m=0; m<5; m++) {
         List<Istance> listaM = new ArrayList<Istance>();
         parziale.add(listaM);
      }
      List<List<Istance>> ritorno = new ArrayList<List<Istance>>();
      //System.out.println("-------first: "+toSchedule.get(0)+"--------\n");

      livelloMax = toSchedule.size();
      ricerca(toSchedule, parziale, ritorno, 0);
      return ritorno;
   }


//  METODO PER LA RICERCA
   private void ricerca(List<Istance> start, List<List<Istance>> parziale, List<List<Istance>>ritorno,  int livello) {

      int worst = getDurationSlowestMachine(parziale);

      //-------------------------CONDIZIONI DI USCITA--------------------

      if(livello == livelloMax && worst < max) {
         flag = true;
         max = worst;
```

```
            //stampa(parziale);
            ritorno.clear();
            for(int i=0; i<parziale.size();i++) {
                //System.out.println(parziale.get(i));
                List<Istance> vmRitorno = new ArrayList<Istance>(parziale.get(i));
                ritorno.add(vmRitorno);
            }
            return;
            }

        //se tutte le istanze sono state schedulate ma non si ha un tempo migliore del max, torna ai passi precedenti
        else if(livello==livelloMax) {
            return;
        }

        // se tempo attuale della macchina piu lenta peggiore di max, anche se non ancora schedulate tutte
        // le istanze, torna indietro ed esegui ricorsione( non perdere tempo)
        else if(worst>max) {
            return;
        }

        //-----------------------------PARTE RICORSIVA----------------------------

        //prendi la prima istanza in start
        Istance is = start.get(0);

        //per ogni macchina, ordinate
        for(int m : this.getOrdine(parziale)) {

            //se istanza non in conflitto con altre istanze su altre macchine, aggiungi e ricerca
            if(!inConflict(is, m, parziale)) {

                parziale.get(m).add(is);
                start.remove(is);
                if(!flag) {
                    ricerca(start, parziale, ritorno,livello+1);
                    parziale.get(m).remove(is);
                    start.add(is);
                }
            }
        }
    }
```

Nello sviluppo del progetto è stata implementata un'altra versione della classe di schedulazione, *SchedulingV2*, che a differenza della prima versione non utilizza liste di liste di istanze, ma salva direttamente le istanze schedulate sulla virtual machine in questione. Durante il testing delle 2 versioni si è notata una differenza di prestazione tra le due versioni, come mostrato dalla seguente schermata:

```
-----CONFRONTO TRA SCHEDULING V1 E V2-----

TENTATIVO 1: MAX = 38000
SOLUZIONE 1, tempo impiegato: 0.0142477 secondi
SOLUZIONE 2, tempo impiegato: 0.011556 secondi

TENTATIVO 2: MAX = 36000
SOLUZIONE 1, tempo impiegato: 0.1679783 secondi
SOLUZIONE 2, tempo impiegato: 0.0901756 secondi

TENTATIVO 3: MAX = 34000
SOLUZIONE 1, tempo impiegato: 0.0439045 secondi
SOLUZIONE 2, tempo impiegato: 0.0117366 secondi

TENTATIVO 4: MAX = 32000
SOLUZIONE 1, tempo impiegato: 4.7393634 secondi
SOLUZIONE 2, tempo impiegato: 3.3244772 secondi

TENTATIVO 5: MAX = 30000
SOLUZIONE 1, tempo impiegato: 2.6320605 secondi
SOLUZIONE 2, tempo impiegato: 16.6197708 secondi
```

Siccome per lo svolgimento della simulazione si prediligono tempistiche più basse, sempre tenendo in considerazione che la durata di svolgimento dell'algoritmo che non può essere elevata, la prima versione è divenuta quella di utilizzo nell'applicazione.

SIMULAZIONE DI EVENTI

La classe *Simulator* si occupa della simulazione di eventi. Al momento della creazione di un oggetto Simulator vengono passati in input i parametri inseriti dall'utente, riguardanti la stabilità generale, la stabilità singola di alcune istanze e sulla modalità di simulazione. Questi parametri vanno a influenzare la creazione degli eventi, cambiando la probabilità di successo di ogni istanza. Il metodo *setSuccessRate()* determina il tasso di successo di un istanza in base a parametri in ingresso e a dati estrapolati dal database.

```
public void setSuccessRate() {

	RunDAO rdao = new RunDAO();


	for(Istance i: this.istances) {

		double successRate=0.0;

		//recupero dal database numero di successi e di fallimenti, numero massimo di failures in un giorno e media di failures
		Integer successes = rdao.getIstanceSuccesses(i);
		Integer failures = rdao.getFailurDays(i);
		Integer maxDailyFailures = rdao.getMaxFailur(i);
		Double averageFailures = rdao.getAverageFailurs(i);

		//controllo se lo scenario dell'istanza in questione sia tra le checkBox selezionate
		boolean scenaryFlag = false;
		if(this.stabilityMap.get(i.getScenario())!=null)
```

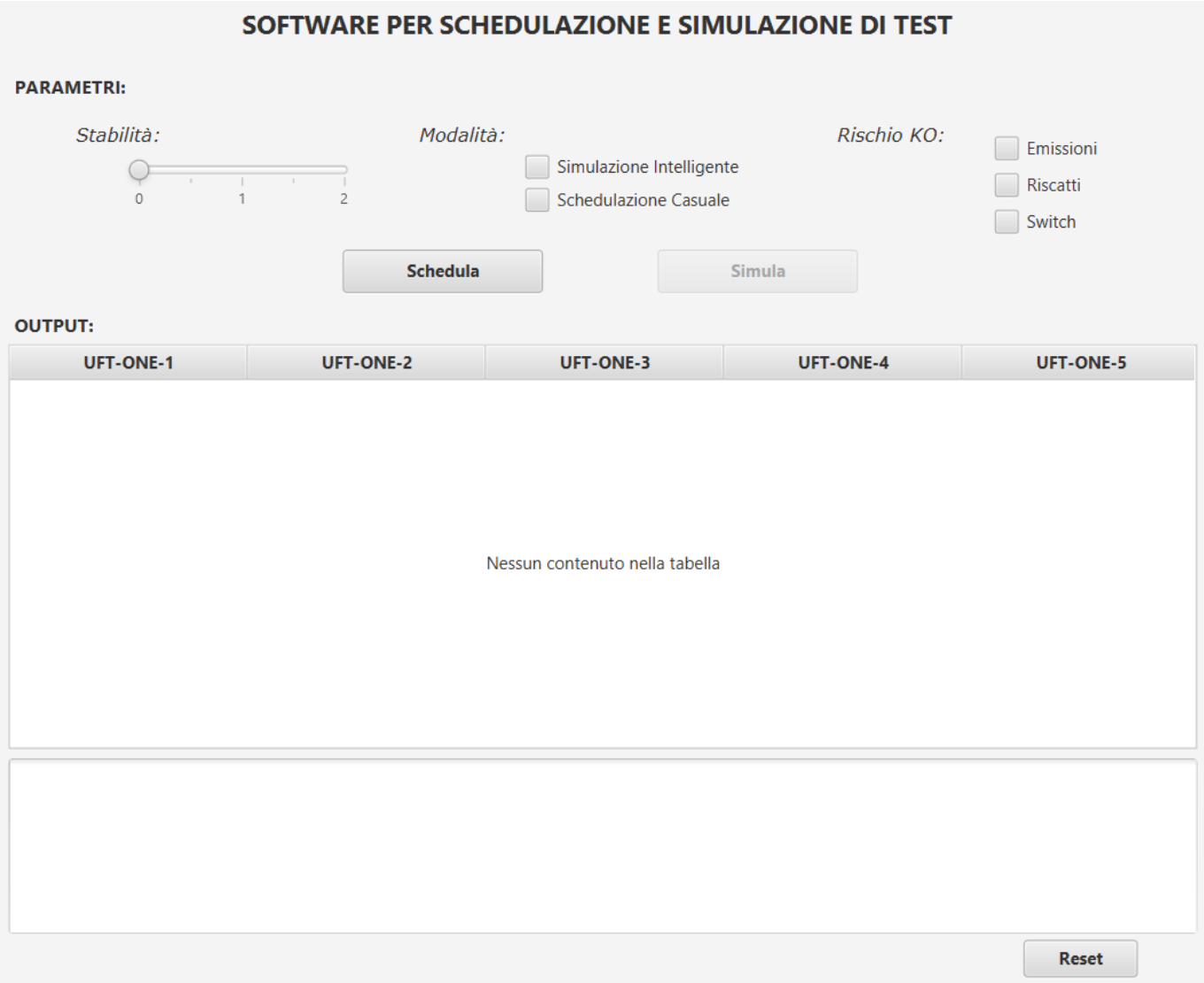
```
            scenaryFlag = this.stabilityMap.get(i.getScenario());

        //----------------------------ATTRIBUISCO UN SUCCESS RATE ALL'ISTANZA-------------------------
-

        // caso migliore: situazione stabile e nessun problema di scenario
        if(this.stability == 2 && !scenaryFlag) {
            successRate = (successes/(double)(successes+failures));
        }

        //caso intermedio: situazione stabile ma problemi di scenario o situazione media e nessun
problema di scenario
        else if ((this.stability==2 && scenaryFlag) ||(this.stability==1 && !scenaryFlag)) {

            if(averageFailures>1.1)
                successRate = 1/averageFailures;
            else
                successRate = 0.9;

        }

        //caso peggiore: situazione media ma problemi di scenario o situazione instabile
        else if( (this.stability==1 && scenaryFlag) || this.stability==0) {

            successRate = 1/ (double)maxDailyFailures;

        }

        i.setSuccessRate(successRate);

    }

}
```

Il risultato di questo metodo, combinato con il controllo di un eventuale conflitto dell'istanza con le altre istanze in esecuzione, determinano il successo o il fallimento del test. Se all'avvio della simulazione l'utente ha selezionato la checkbox *Simulazione Intelligente*, ogni macchina al termine delle esecuzioni previste va alla ricerca di un'istanza da eseguire, controllando che non sia in conflitto con quelle in esecuzione in quel momento. La simulazione termina nel momento in cui tutti i test hanno come stato un successo o una issue, oppure allo scadere del tempo prestabilito.

DIAGRAMMA DELLE CLASSI

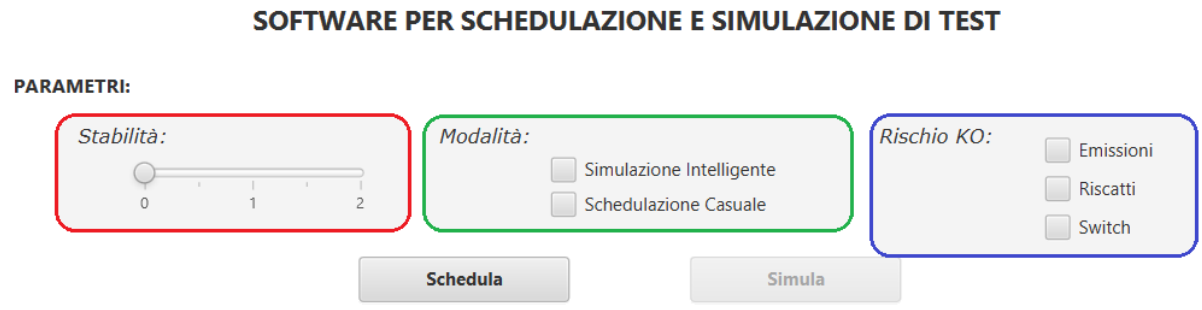
INTERFACCIA GRAFICA

L'interfaccia grafica è divisa in due sezioni: la parte superiore permette all'utente di impostare i parametri, da calibrare in base alla situazione che desidera rappresentare; la parte inferiore, costituita da una tabella e da un area di testo permette di visualizzare i risultati di schedulazione e simulazione. Schermata all'avvio dell'applicazione:



INPUT UTENTE

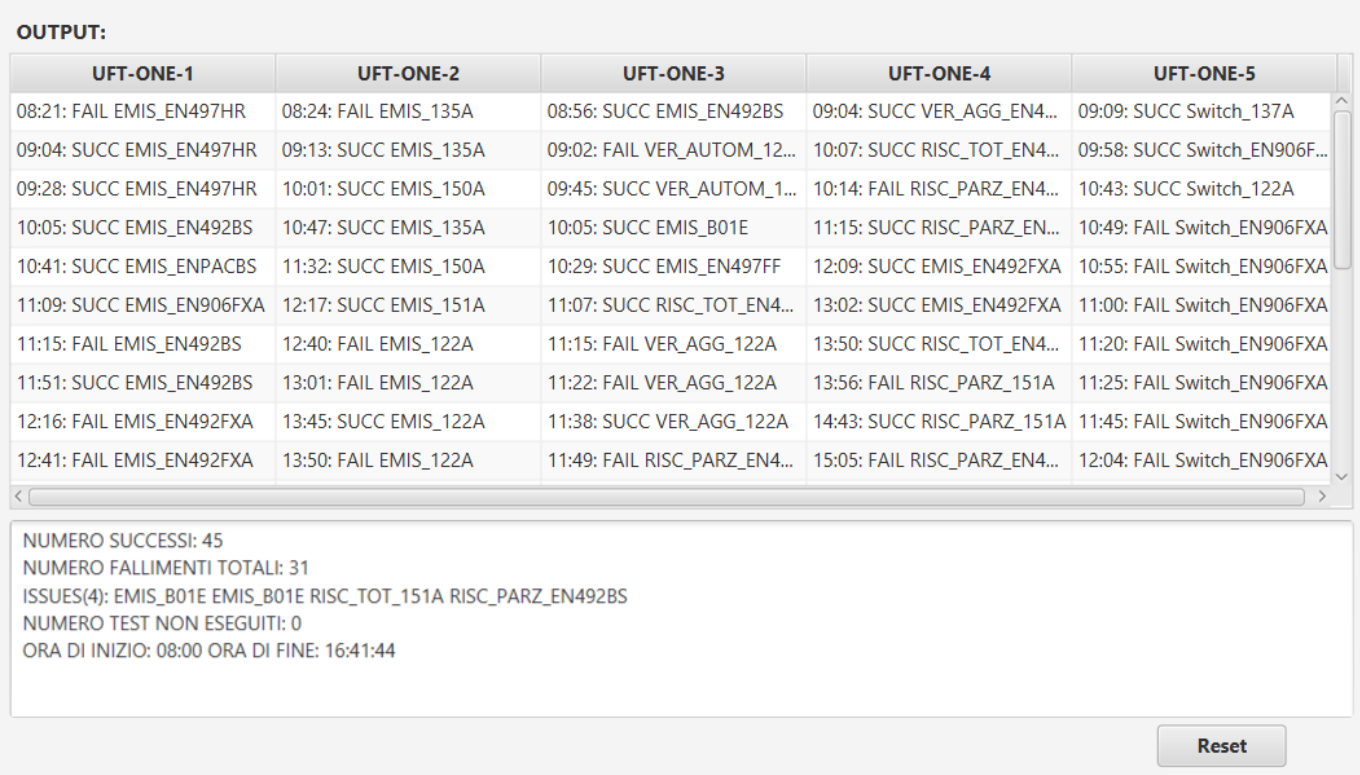
La parte superiore comprende uno slider e alcune checkbox per impostare i parametri e due button, uno per avviare la schedulazione e uno per la simulazione:



Lo slider nel riquadro rosso permette di impostare il livello di stabilità: 0 equivale a un livello di bassa stabilità (tasso di successi minore), 2 a un livello di alta stabilità. Le checkbox nel riquadro verde permettono rispettivamente di svolgere una simulazione intelligente *(vedi 4.5)* e di fare una schedulazione casuale, ovvero di ordinare casualmente la lista di emissioni prima della schedulazione (ciò potrebbe portare a diminuire i tempi di esecuzione della schedulazione oppure ad aumentarli). Le checkbox nel riquadro blu permettono di impostare una maggiore instabilità su determinate istanze: per fare un esempio realistico, spesso capita che nei giorni prima della correttiva vengano apportate modifiche solo ad alcuni script, motivo per cui in certe situazioni si vuole considerare l'instabilità soltanto di alcuni scenari.

RISULTATI APPLICAZIONE

La parte inferiore comprende una tabella per l'output di schedulazione e simulazione, un area di testo per l'analisi dei risultati dopo la simulazione e un button per il reset. La tabella, formata da 5 colonne, una per macchina, rappresenta in ogni cella un'istanza schedulata o un evento simulato. Schermata a fine simulazione:



Link al video dimostrativo: https://www.youtube.com/watch?v=mLdD7e7w0lw

RISULTATI SPERIMENTALI

Si analizzano ora alcune dinamiche osservate.

CONFRONTO ALTA E BASSA STABILITÀ

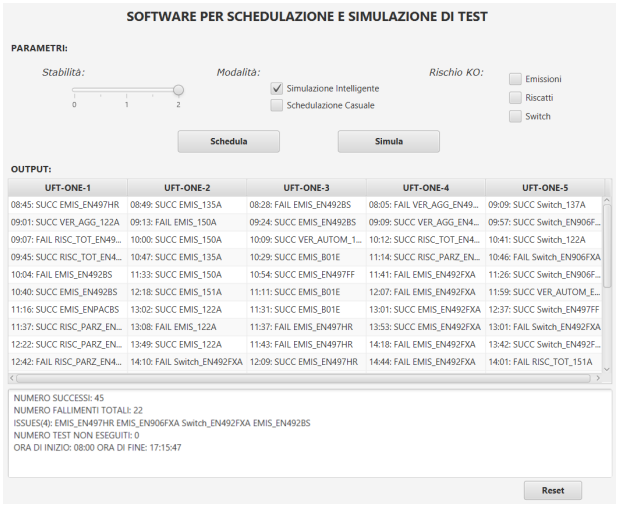
Si osserva il confronto dei risultati ottenuti da una simulazione con alta stabilità e una con bassa stabilità, partendo dalla medesima schedulazione

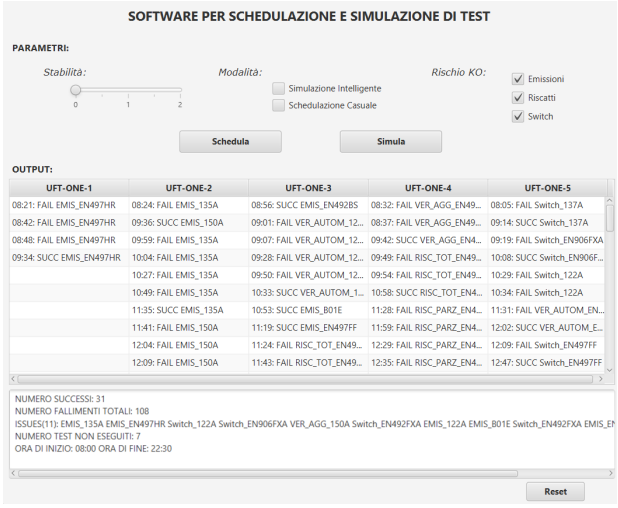
La schedulazione è la seguente:

OUTPUT:

UFT-ONE-1	UFT-ONE-2	UFT-ONE-3	UFT-ONE-4	UFT-ONE-5
EMIS_EN497HR	EMIS_135A	EMIS_EN492BS	VER_AGG_EN492FXA	Switch_137A
EMIS_EN497HR	EMIS_150A	VER_AUTOM_129A	RISC_TOT_EN492FXA	Switch_EN906FXA
	EMIS_135A	EMIS_B01E	RISC_PARZ_EN492FXA	Switch_122A
	EMIS_150A	EMIS_EN497FF	EMIS_EN492FXA	Switch_EN906FXA
	EMIS_151A	RISC_TOT_EN497HR	EMIS_EN492FXA	VER_AUTOM_ENPACBS
	EMIS_122A		EMIS_EN492FXA	Switch_EN497FF
	EMIS_122A		RISC_TOT_EN492BS	EMIS_EN497HR
	Switch_EN492FXA		RISC_PARZ_151A	VER_AGG_150A
	EMIS_122A		RISC_PARZ_EN492BS	EMIS_EN906FXA
	VER_AGG_122A		RISC_PARZ_EN497HR	EMIS_B01E

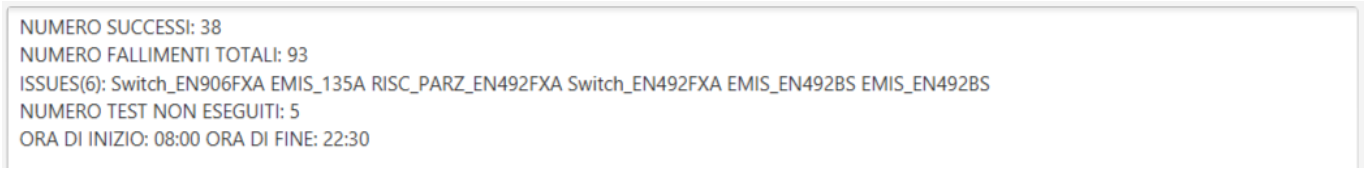
Simulazioni con alta e bassa stabilità:

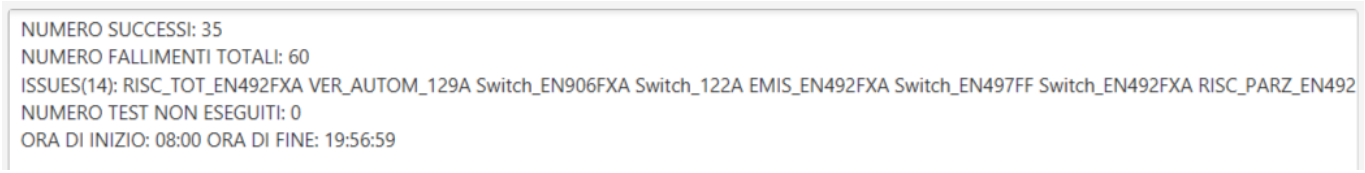


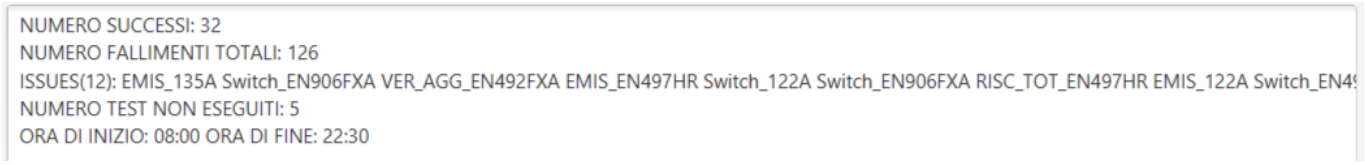


Dai due differenti output nelle rispettive aree di testo si può notare subito la differenza tra le due situazioni di partenza iniziali: nel primo caso infatti su 49 test da eseguire ben 45 hanno registrato uno stato di successo e solo 4 hanno presentato una issue, con un tasso di successo pari al *91.8%*, e si sono verificati soltanto 22 fallimenti momentanei, meno di uno per coppia di istanze; nel secondo caso invece i successi totali sono 31 e le issue 11, con un tasso di successo del *63.3%* e i fallimenti momentanei totali sono ben 108,mediamente più di 2 per istanza. Questi risultati fanno notare come l'alta instabilità della situazione di partenza generi un maggiore numero di fallimenti, che possono andare a compromettere la schedulazione di partenza se mal distribuiti sulle macchine e portare alla mancata esecuzione di tutti i test per insufficienza di tempo. In questo caso entra in gioco la Simulazione Intelligente *(vedi 4.5)* che permette alle macchine di non fermarsi nel momento in cui finiscono di eseguire le istanze assegnategli in partenza. Questa dinamica appena descritta riguarda l'instabilità degli script: all'aumentare di quest'ultima aumenta il numero di errori durante la

simulazione. Ma per definizione l'instabilità riguarda anche la differenza di prestazioni tra una situazione apparentemente identica, ovvero l'alta variabilità dei risultati che si possono ottenere. Ciò è misurabile lanciando diverse volte la simulazione con medesima situazione di partenza e schedulazione:







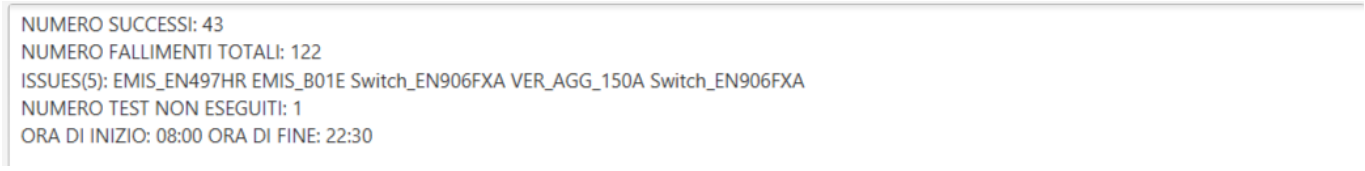
Da questi risultati si può notare come siano estremamente variabili i dati sul numero di successi, di fallimenti totali e di issues. Va osservato che dinamiche di questo tipo sono estremamente problematiche per gli sviluppatori, in quanto non riescono a prevedere l'andamento del processo di correttiva.

CONFRONTO TRA SIMULAZIONE STANDARD E INTELLIGENTE

Si analizzano qui le differenze di output prodotti da una simulazione *Intelligente* e una standard:

- Simulazione standard:

- Simulazione intelligente:



Come si può notare da queste immagini, nel primo caso l'alta instabilità porta la simulazione standard a non solo a non stare nei tempi assegnati, ma a non eseguire ben 7 istanze. Nel secondo caso invece, nonostante il numero di fallimenti maggiore (che si traduce in maggior tempo sprecato per eseguire i test), la simulazione intelligente permette di eseguire più istanze: il numero di quelle non eseguite qui è infatti soltanto 1.

CONCLUSIONI FINALI

Possiamo notare che l'applicativo risponde bene agli input dell'utente, presentando tempi di reazione sufficientemente brevi. Tra i punti di forza emergono l'algoritmo di simulazione, che riesce a rappresentare una situazione reale, ritornando dei parametri che possono essere utilizzati per analizzare la situazione e l'algoritmo ricorsivo, che seppur non trovando la soluzione ottima al problema riesce comunque a produrre una schedulazione che permetta un margine sufficiente allo svoglimento di tutti i test, anche in una condizione di stabilità intermedia. Questi due fattori combinati possono tornare molto utili agli sviluppatori, in quanto grazie all'algoritmo di schedulazione possono dedicare meno tempo all'assegnazione di una macchina virtuale a un test, e grazie all'output della simulazione possono capire con anticipo se è il caso di intervenire a codice prima dell'inizio ufficiale dello svolgimento dei test. Tra le principali criticità figura il raccoglimento dei dati su cui si fonda la simulazione di eventi. Per ottenere un output che sia il più accurato possibile andrebbero analizzate meglio le informazioni in input, cercando la presenza di eventuali trend per riportarla durante la simulazione osservandone le conseguenze che ne derivano. Per concludere, l'applicativo risponde bene agli input e, con una premessa di accuratezza dei dati raccolti, risolve un problema reale.