ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Informatica

Tesi di Laurea Triennale

Tallulah: Gestione Assiomatica di Modelli Reversibili

Relatore
prof. Ivan Lanese

Laureando
William Arnone

Marzo 2022

Una dedica agli amici che mi hanno accompagnato in questi anni regalandomi momenti indelebili e ai miei familiari. Senza i loro sacrifici tutto questo non sarebbe stato possibile.

Abstract

Riuscire ad annullare un'azione in un programma comporta numerosi vantaggi: uno dei quali è la maggiore semplicità nel rilevare problemi in fase di debugging. Mentre nei programmi sequenziali è sufficiente annullare ricorsivamente le ultime istruzioni eseguite, ciò non è possibile in un ambiente concorrente. Tra i diversi approcci proposti per i modelli da utilizzare in questo caso ne esiste uno chiamato *Causal-Consistent Reversibility*. La Causal-Consistent Reversibility sostituisce il concetto di temporalità con quello di causalità. In letteratura è stato studiato un sistema assiomatico per dimostrare le proprietà della Causal-Consistent Reversibility tramite assiomi più semplici da verificare. Nonostante ciò risulta comunque complicato verificare queste proprietà manualmente. In questo documento si espone Tallulah: uno strumento per la gestione di questi assiomi e proprietà.

Indice

Capitolo 1

Introduzione

Al giorno d'oggi quasi la metà del tempo che viene impiegato per la produzione di un software si impegna principalmente nella fase di debugging [12]. Una volta verificatosi un errore è raramente possibile ritornare agli stati precedenti del calcolatore per comprendere dove l'errore si sia generato.

Strumenti come Windows Time Traveler [13] o il debugger GDB [3] consentono già di annullare azioni in fase di esecuzione, ma questo è possibile solo perché operano in un ambiente sequenziale, dove le pratiche di reversibilità sono state comprese a fondo. Esse infatti si basano sul fatto che vengono annullate ricorsivamente le ultime azioni eseguite. Però la maggior parte dei programmi che vengono prodotti adesso sono concorrenti [11] e per essi non è corretto utilizzare lo stesso approccio. Per questo motivo sono nate diverse proposte su quali tecniche si debbano adottare in un ambiente concorrente. Una di queste è la *Causal-Consistent Reversibility* [2].

La Causal-Consistent Reversibility generalizza ciò che succede nella reversibilità dei programmi sequenziali, ma non potendo più basarsi su quando un azione viene eseguita, sostituisce il concetto di temporalità con quello di causalità: si annullano ricorsivamente le azioni, annullando prima le loro conseguenze, se ci sono. Oltre a ciò, vi è anche un lemma fondamentale chiamato Loop Lemma, esso enuncia che eseguire un'azione e annullarla deve far ritornare la computazione allo stato precedente, lo stesso ovviamente deve avvenire anche se prima un'istruzione viene annullata e poi eseguita .

Per soddisfare il Loop Lemma dunque non è possibile salvare in memoria tutte le azioni che sono state annullate, in quanto lo stato della macchina non sarebbe più come il precedente. La reversibilità Causal-Consistent garantisce che un qualsiasi stato raggiungibile dallo stato iniziale tramite l'esecuzione e l'annullamento di istruzioni possa anche essere raggiungibile da solo esecuzioni. Quanto detto viene catturato in definitiva dal modello chiamato *Causal Consistency*.

Cercare di verificare la Causal Consistency caso per caso però è un problema a dir poco

complicato, si è dunque tentato di ridurre il problema al controllo di assiomi di più facile verifica [5], ma che potessero implicare le proprietà della Causal Consistency. Questi assiomi utilizzano come modello un sistema generico di transizioni etichettate, con aggiunte relazioni di indipendenza (Labelled Transition System with Independence, da adesso abbreviato in LTSI), che verranno meglio definite in seguito (Vedi sezione 2.2). Nonostante questi assiomi e proprietà semplifichino un problema complicato, rimane comunque difficile verificare la loro correttezza manualmente, soprattutto quando il numero di transizioni inizia ad essere elevato. Per questo motivo è stato implementato un software che possa effettuare questi controlli in modo automatizzato [9].

Il tool prende in input un file DOT (.gv o .dot) che descrive un grafo diretto con archi etichettati, mentre per le informazioni riguardanti l'indipendenza, esse vengono espresse tramite una sintassi particolare all'interno dei commenti, in modo da rendere il file compatibile anche con tutti i software che utilizzano i file DOT. Una volta letto l'LTSI è possibile selezionare di quali assiomi e proprietà si vuole controllare la correttezza, nel caso non siano verificati viene visualizzato un controesempio. Il programma fornisce anche un'altra possibilità: quella di forzare la correttezza delle proprietà, generando un nuovo grafo.

Capitolo 2

Teoria alla base del tool

2.1 Causal-Consistent Reversibility

Reversibilità La computazione reversibile è un paradigma che permette l'esecuzione dei programmi non solo in una direzione (*forward*) come nella programmazione standard, ma che permette anche una esecuzione inversa (*backward*), permettendo di poter ritornare a stati di computazione precedenti. La reversibilità comporta svariati vantaggi: uno in particolare è quello di rendere più semplice il debugging di un software, una volta presentatosi un errore è infatti possibile ripercorrere gli stati del computer all'indietro per poter individuare cosa lo ha causato. Nei programmi sequenziali sono già presenti strumenti in grado di fare ciò (come [3] e [13]), il loro funzionamento può essere riassunto in: "annulla ricorsivamente le ultime azioni", si basano infatti sull'ordine di esecuzione delle istruzioni: se si esegue A, B ed infine C, si annulleranno nell'ordine C, B e A.

Nella programmazione concorrente però non si può applicare la stessa metodologia perché l'ordine di esecuzione delle istruzioni può variare in base a numerevoli fattori: come ad esempio la presenza di più programmi aperti contemporaneamente e la CPU presente nella macchina.

Causal-Consistent Reversibility e Loop Lemma Tra le diverse proposte su come si dovesse approcciare la reversibilità in ambienti concorrenti ve ne è una chiamata *Causal-Consistent Reversibility* [2]: "annulla ricorsivamente le azioni, annullando prima le loro conseguenze, se ci sono". In questo modo la componente temporale viene sostituita con quella causale, in modo da definire quali azioni debbano essere annullate per prime. Nell'articolo viene anche enunciato un lemma molto importante, il *Loop Lemma*: "eseguire un'azione e annullarla, oppure annullarla ed eseguirla nuovamente, dovrebbe riportare la macchina allo stato precedente".

La Causal-Consistent Reversibility non garantisce solo che ogni stato sia raggiungibile dallo stato iniziale, ma anche che uno stato raggiungibile da transizioni forward e backward può essere raggiunto anche da sole transizioni forward.

Assumiamo di avere delle azioni $A, B..., A^{-1}, B^{-1}, ...$, dove A^{-1} è la transizione che annulla A. Consideriamo la traccia σ come una sequenza di azioni, denotando la traccia vuota con il simbolo ϵ.

Definizione 2.1.1 (Indipendenza) *Indichiamo con $\imath$ una relazione simmetrica binaria tra due azioni per indicare che l'ordine con cui vengono eseguite non è importante.*

Seguendo la Definizione 9 di [2]:

Definizione 2.1.2 (Causal Equivalence) *Definiamo con $\approx$ la relazione di causal equivalence come la più piccola relazione di equivalenza su tracce chiuse rispetto alla concatenazione e che soddisfino i seguenti assiomi:*

$$AA^{-1} \approx \epsilon \qquad A^{-1}A \approx \epsilon \qquad AB \approx BA \Leftrightarrow A \imath B$$

In questo modo si può verificare che è possibile ottenere da ogni traccia causal-consistent una traccia con sole transizioni forward, causal equivalent ad essa. Questo perché se tra un'azione e la propria inversa le conseguenze sono eseguite ed annullate, tramite gli assiomi presentati precedentemente si possono ottenere tracce equivalenti alle precedenti senza le conseguenze, ricorsivamente si ottiene una traccia ϵ o contenente solo azioni forward.

Causal-consistency theorem In un software sono presenti spesso istruzioni non reversibili senza l'aggiunta di altre informazioni: per esempio una volta eseguito l'assegnamento $X = 5$ si ha una perdita del precedente valore di X. Sarà quindi necessario memorizzare lo stato di X e mantenerlo in memoria per poter rendere possibile tornare allo stato precedente, così come sarà necessario memorizzare altre informazioni cronologiche sull'esecuzione per altri casi specifici. Informazioni come "quante volte è stata annullata un'azione" però non possono essere salvati, perché violerebbero il Loop Lemma.

Cosa debba essere memorizzato e cosa no viene specificato nel *causal-consistency theorem*: "due computazioni che iniziano dallo stesso stato terminano nello stesso stato se e solo se esse sono causal equivalent".

Se due computazioni terminano nello stesso stato allora hanno le medesime informazioni memorizzate, quindi l'implicazione $\Leftarrow$ indica che non è possibile salvare quante volte un'azione viene eseguita e annullata, e nemmeno quale delle due istruzioni concorrenti venga eseguita per prima. L'implicazione $\Rightarrow$ invece indica che se uno stato è raggiungibile da due computazioni non causal equivalent è necessario aggiungere delle informazioni per distinguerle.

2.2 Labelled Transition System with Independence

Gli assiomi descritti nella sezione 2.3 sono stati realizzati in modo da essere il più generiche possibile. Per lo stesso motivo è stato utilizzato un modello anch'esso generico: un sistema di transizioni etichettate (vedi definizione 2.2.1). Trattando proprietà relative alla programmazione reversibile però è necessario rappresentare sia le transizioni forward che le transizioni backward. Quindi sarà necessario un LTS combinato (vedi definizione 2.2.4) a cui poi verranno aggiunte le relazioni di indipendenza per renderlo un LTSI (vedi definizione 2.2.2).

Definizione 2.2.1 (LTS) *Possiamo definire un* ***sistema di transizioni etichettate (Labelled Transition System, LTS)*** *come un insieme* $(\mathbf{Proc}, \mathbf{Lab}, \rightarrow)$*, dove* **Proc** *è un insieme di stati (o processi),* **Lab** *un insieme di etichette e* $\rightarrow \subseteq \mathbf{Proc} \times \mathbf{Lab} \times \mathbf{Proc}$ *una relazione di transizione.*

Siano $P, Q \in \mathbf{Proc}$*;* $a \in \mathbf{Lab}$*, una transizione* t *può essere scritta in questo modo:*

$$t : \ P \xrightarrow{a} Q \text{ per indicare } t = (P,\ a,\ Q)$$

Definizione 2.2.2 (LTSI) *Si parla invece di* ***LTSI (Labelled Transition System with Independence)*** *quando abbiamo* $(\mathbf{Proc}, \mathbf{Lab}, \rightarrow, \imath)$*, dove (***Proc***,* **Lab***,* $\rightarrow$*) definisce un LTS e* $\imath$ *una relazione sulle transizioni binaria simmetrica non riflessiva, chiamata relazione di indipendenza.*

Definizione 2.2.3 (LTS Inverso) *Dato un LTS (***Proc***,* **Lab***,* $\rightharpoonup$*) definiamo il suo LTS inverso (***Proc***,* **Lab***,* $\rightsquigarrow$*) in questo modo:*

$$P \overset{a}{\rightsquigarrow} Q \Leftrightarrow Q \overset{a}{\rightharpoonup} P$$

Definizione 2.2.4 (LTS Combinato) *Per comodità combineremo un LTS e il suo inverso in un terzo LTS, definendo le etichette delle transizioni invertite come* $\underline{\mathbf{Lab}} = \{\underline{a} : a \in \mathbf{Lab}\}$ *e l'LTS combinato come*

$$(\mathbf{Proc}, \mathbf{Lab} \cup \underline{\mathbf{Lab}}, \rightarrow)$$

In questo caso $\rightarrow$ *diventa* $\rightarrow \subseteq (\mathbf{Proc} \times (\mathbf{Lab} \cup \underline{\mathbf{Lab}}) \times \mathbf{Proc})$ *dove* $P \xrightarrow{a} Q \Leftrightarrow P \overset{a}{\rightharpoonup} Q$ *e* $P \xrightarrow{\underline{a}} Q \Leftrightarrow P \overset{a}{\rightsquigarrow} Q$*.*

Per ottenere l'etichetta non inversa definiamo la funzione **und** come $\mathbf{und}(\alpha) = \alpha$ e $\mathbf{und}(\underline{\alpha}) = \alpha$. Dato $t : \ P \xrightarrow{\alpha} Q$, $\underline{t} : \ Q \xrightarrow{\underline{\alpha}} P$ è la transizione che inverte t. Definiamo inoltre per ogni $\alpha \in \mathbf{Proc}$, $\underline{\underline{\alpha}} = \alpha$.

Dato un LTS, un path è una sequenza di transizioni forward o backward nella forma: $P_0 \xrightarrow{\alpha_1} P_1 ... \xrightarrow{\alpha_n} P_n$. Si può scrivere anche $r : \ P \xrightarrow{\rho}_* Q$ quando gli stati intermedi sono

sottintesi. L'inverso di $r: P \xrightarrow{\rho}_* Q$ è $\underline{r}: Q \xrightarrow{\underline{\rho}}_* R$ dove $\underline{\epsilon} = \epsilon$ e $\underline{\rho\alpha} = \underline{\alpha}\underline{\rho}$. Due paths $r: P \xrightarrow{\rho}_* Q$ e $s: R \xrightarrow{\sigma}_* S$ si dicono coiniziali se $P = R$ e cofinali se $Q = S$.

Gli LTSI utilizzati nelle definizioni degli assiomi e delle proprietà della sezione 2.3 sono ottenuti combinando un LTS con il suo inverso, a cui viene aggiunta la relazione di indipendenza, chiamiamo gli LTSI ottenuti in questo modo **LTSI combinati**.

Facendo un esempio: Assumendo $P,\ Q,\ R \in$ **Proc**; $a,\ b \in$ **Lab**; $(P,\ a,\ Q)$, $(Q,\ \underline{a},\ P)$, $(P,\ b,\ R)$, $(R,\ \underline{b},\ P) \in\!\!\rightarrow$ e $(P,\ a,\ Q)\ \imath\ (P,\ b,\ R)$. Abbiamo (come visualizzato nella figura 2.1) un LTSI con due transizioni forward e due backward, inoltre le due transizioni $P \xrightarrow{a} Q$ e $P \xrightarrow{b} R$ sono indipendenti tra loro. Graficamente si visualizzeranno solo le transizioni forward, sottintendendo sempre anche la presenza delle transizioni backward.

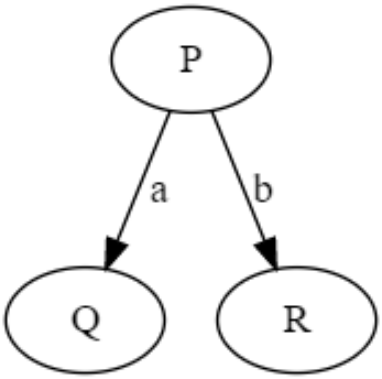

Figura 2.1: Esempio di LTSI

2.3 Assiomi e Proprietà

Determinare se un programma gode di causal-consistency è un problema molto complicato, perché si dovrebbe controllare caso per caso ogni possibile sviluppo dello stato di una computazione, per semplificare questa verifica sono stati studiati degli assiomi [5] che fossero più semplici da verificare rispetto al problema complessivo, ma che insieme potessero dimostrarne le proprietà.

2.3.1 Proprietà di base

Per provare la Causal-Consistency sono necessari pochi e semplici assiomi, mostrati in questa sezione.

Assioma 2.3.1 (SP - Square Property)

Se $t: P \xrightarrow{\alpha} Q,\ u: P \xrightarrow{\beta} R,\ t\ \imath\ u$ *allora esistono* $u': Q \xrightarrow{\beta} S,\ t': R \xrightarrow{\alpha} S$

Con la Square Property si vuole catturare l'idea che due transizioni indipendenti tra loro possono essere eseguite in un qualsiasi ordine, formando in questo modo i commuting diamonds.

Assioma 2.3.2 (BTI - Backward Transitions are Independent)

$$Se\ t:\ P \overset{\alpha}{\rightsquigarrow} Q,\ t':\ P \overset{\beta}{\rightsquigarrow} Q',\ t \neq t'\ allora\ t\ \iota\ t'$$

BTI consiste nel fatto che due transizioni backward coiniziali coincidono. Generalizzando: "due transizioni backward coiniziali sono indipendenti".

Assioma 2.3.3 (WF - Well-Foundedness)

$$Non\ si\ hanno\ P_i\ tale\ che\ P_{i+1} \xrightarrow{\alpha_i} P_i\ per\ ogni\ i = 0,1,...$$

WF può essere visto così: "non esistono cammini all'indietro di lunghezza infinita". Questo perché si inizia sempre da uno stato iniziale per poi eseguire o annullare azioni.

Nonostante PL e CC non siano proprietà presenti in Tallulah, in seguito viene riportato quanto scritto in [5] riguardo alla loro derivabilità dagli assiomi precedentemente descritti.

Proprietà 2.3.1 (PL - Parabolic Lemma)

$$Per\ ogni\ path\ r\ esistono\ dei\ cammini\ in\ avanti\ s, s\ tali\ che\ r \approx \underline{s}s'\ e\ |s| + |s'| \leq |r|$$

Si utilizza per $\approx$ la definizione 2.1.2 e dato un path r, $|r|$ rappresenta il numero di transizioni presenti in r. È possibile provare che se un LSTI soddisfa BTI e SP allora soddisfa anche PL.

Proprietà 2.3.2 (CC - Causal Consistency)

$$Se\ r\ ed\ s\ sono\ coiniziali\ e\ cofinali\ allora\ r \approx s$$

Con CC si vuole distinguere computazioni che non sono causal equivalent, infatti se sono cofinali raggiungono lo stesso stato (con le stesse informazioni sulle azioni eseguite), allora devono essere necessariamente causal equivalent. Spesso vale anche l'implicazione da destra a sinistra: indicando che se due computazioni sono causal equivalent non è possibile distinguere quali delle due direzioni si sia percorsa. È possibile dimostrare che se un LTSI soddisfa WF e PL allora soddisfa anche CC.

2.3.2 Eventi

Definizione 2.3.1 (Evento) *Dato un LTSI* $(Proc,\ Lab,\ \rightarrow,\ \imath)$ *definiamo con* $\sim$ *la più piccola relazione di equivalenza che soddisfa: se* $t:\ P \xrightarrow{\alpha} Q,\ u:\ P \xrightarrow{\beta} R,\ u':\ Q \xrightarrow{\beta} S,\ t':\ R \xrightarrow{\alpha} S\ e\ t\ \imath\ u,\ \underline{u}\ \imath\ t',\ \underline{t'}\ \imath\ \underline{u'},\ u'\ \imath\ \underline{t}\ e$

$$\begin{cases} Q \neq R, & \text{se } \alpha \text{ e } \beta \text{ sono entrambi forward o entrambi backward.} \\ P \neq S, & \text{altrimenti.} \end{cases}$$

allora $t \sim t'$*. Le classi di equivalenza delle transizioni forward, scritte* $[P, \alpha, Q]$ *sono eventi, quelle delle transizioni backward* $[P, \underline{\alpha}, Q]$ *sono eventi inversi.*

Assioma 2.3.4 (CPI - Coinitial Propagation of Independence)

$$Se\ t:\ P \xrightarrow{\alpha} Q,\ u:\ P \xrightarrow{\beta} R,\ u':\ Q \xrightarrow{\beta} S,\ t':\ R \xrightarrow{\alpha} S\ e\ t\ \imath\ u\ allora\ u'\ \imath\ \underline{t}$$

CPI estende l'indipendenza da due transizioni all'intero commuting diamond di cui fanno parte, propagando l'indipendenza lungo gli angoli del commuting diamond.

Definizione 2.3.2 (LTSI Pre-reversibili) *Se un LTSI soddisfa contemporaneamente SP, WF, BTI e CPI esso viene chiamato* ***pre-reversibile****.*

Proprietà 2.3.3 (ID - Independence of Diamonds) *Se abbiamo un commuting diamond:* $t:\ P \xrightarrow{\alpha} Q,\ u:\ P \xrightarrow{\beta} R,\ u':\ Q \xrightarrow{\beta} S,\ t':\ R \xrightarrow{\alpha} S\ e$

$$\begin{cases} Q \neq R, & \text{se } \alpha \text{ e } \beta \text{ sono entrambi forward o entrambi backward.} \\ P \neq S, & \text{altrimenti.} \end{cases}$$

allora $t\ \imath\ u$.

Se un LTSI soddisfa CPI e BTI allora soddisfa anche ID.

2.3.3 Causal Safety e Causal Liveness

In [5] vengono introdotte le proprietà: *Causal Safety* e *Causal Liveness*, esse non vengono controllate in Tallulah, ma possono essere implicate dalla presenza di altre proprietà.

Definizione 2.3.3 (CS - Causal Safety) *Un'azione non può essere annullata finché tutte le azioni causate da essa non siano annullate;*

Definizione 2.3.4 (CL - Causal Liveness) *Deve essere possibile invertire l'esecuzione in un qualsiasi ordine compatibile con CS, anche se è un'ordine diverso da quello di esecuzione.*

È possibile notare che queste due nuove proprietà sono ispirate da CC ma non è necessariamente vero che se CS e CL siano entrambe valide lo sia anche CC.

2.3.4 Altre Proprietà

Assioma 2.3.5 (IRE - Independence Respects Events)

Se $t \sim t'$ $\imath$ u *allora* t $\imath$ u.

IRE implica che se due archi sono indipendenti tra loro, lo sono anche gli archi appartenti ai rispettivi eventi. Se un LTSI è pre-reversibile IRE è una condizione sufficiente per dimostrare la validità di CS e CL.

Assioma 2.3.6 (IEC - Independence of Events is Coinitial)

Se t_1 $\imath$ t_2 *allora esistono* $t_1' \sim t_1$, $t_2' \sim t_2$ *tali che* t_1' *e* t_2' *sono coiniziali e* t_1' $\imath$ t_2'

IEC indica che l'indipendenza è completamente determinata dalla propria restrizione alle transizioni coiniziali.

Proprietà 2.3.4 (RPI - Reversing Preserves Independence)

Se t $\imath$ t' *allora* $\underline{t}$ $\imath$ t'

Se un LTSI soddisfa SP, CPI, IRE e IEC allora soddisfa anche RPI.

Assioma 2.3.7 (CIRE - Coinitial Independence Respects Events)

Se $[t]$ **ci** $[u]$ *e* t, u *sono coiniziali, allora* t $\imath$ u

Dati due eventi e, e' si indica con e **ci** e' che i due eventi sono **coinitially independent**, ovvero che esistono due transizioni t, t' tali che [t]=e, [t']=e' e t $\imath$ t'.

Capitolo 3

Librerie utilizzate

Tallulah è un programma che si pone il compito di controllare che in un grafo valgano le proprietà descritte nella sezione 2.3 con la possibilità di modificare il grafo per forzarne la validità dove necessario. I compiti del programma non strettamente legati al controllo delle proprietà sono principalmente due: la visualizzazione a schermo del grafo e la lettura del file. Si tratta di due funzionalità che richiedono diverso tempo per essere implementate e, dato che sono presenti online diverse librerie che svolgono egregiamente questi compiti, si è deciso di utilizzarle nello sviluppo di Tallulah, in modo da concentrare il tempo e gli sforzi sull'implementazione dei controlli delle proprietà.

Ora verranno introdotte le librerie utilizzate, per poi essere approfondite nelle sezioni a seguire.

Visualizzare il grafo Visualizzare a schermo un grafo richiede una buona gestione degli spazi per organizzare nodi e archi in modo ordinato e che ottimizzi lo spazio a disposizione. Graphviz è un software in grado di esportare un grafo descritto in forma testuale tramite il linguaggio DOT [8] in una varietà di formati diversi: immagini, SVG, PDF e PostScript.

Sono disponibili per i linguaggi di programmazione più utilizzati delle librerie che permettono di utilizzare le capacità di Graphviz nei propri programmi, in particolare per Tallulah è stata utilizzata la libreria di Python [6].

Leggere i file Tallulah deve già leggere dei file DOT per la visualizzazione del grafo, quindi si sarebbe potuto usare una libreria che potesse leggere i file dot per restituire le informazioni necessarie.

Questo però non è possibile perché per controllare la presenza delle proprietà illustrate nella sezione 2.3 l'utente deve specificare anche tra quali archi è presente l'indipendenza, cosa che il linguaggio DOT non supporta nativamente, quindi è stata creata una nuova sintassi per indicare l'indipendenza che viene inserita all'interno dei commenti multi-line (che

in linguaggio DOT sono compresi tra "/*" e "*/") in modo da rendere i file completamente compatibili con altri software che utilizzano i file DOT.

Per questo motivo si è preferito implementare un nuovo parser, generato da ANTLR a partire da un file contenente le regole del nuovo linguaggio ottenuto.

ANTLR (acronimo di ANother Tool for Language Recognition) [1] è un software che permette di generare un parser in grado di poter essere utilizzato per leggere, processare, eseguire o tradurre file di testo o binari che siano organizzati in modo strutturato. I linguaggi per i quali si possono ottenere i parser di ANTLR sono: Java, C#, Python, javaScript, Go, C++, Swift, PHP e DART.

3.1 Graphviz

Graphviz permette di esportare un grafo descritto in linguaggio DOT in immagini. Un file DOT (solitamente con estensione ".dot" o ".gv") può presentarsi cosi:

```
1  digraph G {
2    start [shape=Mdiamond];
3    end [shape=Msquare];
4    { k1; k2; };
5    start -> k1
6    subgraph cluster_0 {
7      node [style=filled, color=white];
8      a0 -> a1 -> a2 -> a3 [color=red];
9      style=filled;
10     color=lightgrey;
11     label = "sub #1";
12   }
13   subgraph cluster_1 {
14     node [style=filled];
15     b0 -> b1 -> b2 -> b3;
16     label = "sub #2";
17     color=blue
18   }
19   {a3 b3} -> end;
20   start -> {k1 -> k2 -> {a0 b0} [style=dashed];} ["style"=dotted];
21 }
```

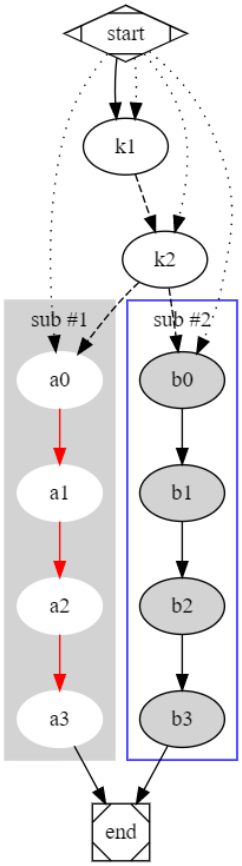

Figura 3.1: Risultato del file DOT di esempio

Grafi e nodi L'esempio si apre con una definizione di un grafo diretto di nome **G** e vengono dichiarati due nodi: **start** ed **end**, entrambi con una forma personalizzata tramite proprietà tra parentesi quadre (*Righe 1-3*).

Sottografi Alla *riga 4* viene definito un sottografo e vengono dichiarati i nodi **k1** e **k2**, in questo caso la definizione del sottografo può anche venire omessa senza cambiare nulla nel risultato finale.

Archi Vi è anche un arco che inizia dal nodo **start** e termina nel nodo **k1** (*Riga 5*), non viene messo il punto e virgola per mostrare che la sua presenza è completamente facoltativa.

Attributi Oltre a poter specificare le proprietà di nodi ed archi singolarmente si possono utilizzare le parole chiave: "graph", "edge" e "node" per influenzare tutti gli elementi del sottografo nel quale vengono utilizzate (*Righe 7 e 14*).

Archi consecutivi Alla *riga 8* vengono definiti 3 archi consecutivamente, con i nodi: **a0**, **a1**, **a2** e **a3**, ognuno collegato al successivo. Da notare che i 4 nodi non sono stati definiti precedentemente, essi vengono definiti nel momento in cui viene dichiarato l'arco che li collega. Infine i 3 archi sono rossi perché la proprietà tra parentesi quadre influenza tutti gli archi della dichiarazione.

Sottografi cluster Se si vogliono applicare delle modifiche estetiche ai sottografi è necessario segnalarlo a Graphviz utilizzando un identificativo che inizi per "cluster". Il sottografo **cluster_0** può specificare le proprietà: stile, colore ed etichetta (*Righe 9-11*) proprio per questo motivo.

Viene poi definito un sottografo in modo simile al precedente (*Righe 13-18*). I sottografi vengono utilizzati sopratutto in casi simili alle *righe 19 e 20*: nella *riga 19* creiamo due archi: uno da **a3** a **end** ed uno da **b3** a **end**: viene creato un arco per ogni nodo presente nel sottografo a sinistra.

Più sottografi nella definizione di archi La *riga 20* presenta un altro caso particolare: viene dichiarato un sottografo all'interno della definizione di un arco, all'interno di un sottografo a sua volta nella definizione di un arco. In questo caso succede questo: per ogni nodo presente all'interno del sottografo più interno (**a0** e **b0**) viene creato un arco tratteggiato da **k2** fino al nodo corrispondente e successivamente, per ogni nodo presente all'interno del primo sottografo (**k1**, **k2**, **a0** e **b0**) viene creato un arco punteggiato da **start** fino ai nodi del sottografo. La figura 3.1 è stata generata da Graphviz con il codice mostrato precedentemente, è possibile distinguere gli archi tratteggiati e punteggiati per comprendere meglio come vengano generati dalla *riga 20*.

Stringhe L'attributo **"style"** (*Riga 20*) qui è tra virgolette per evidenziare una particolarità del linguaggio DOT: scrivere un identificativo con le virgolette o senza non ha alcuna importanza se essa è composta da una parola sola, viceversa nel caso siano presenti più parole (come per il valore attribuito a **label** nelle *righe 11 e 16*) omettere le virgolette comporterebbe un errore o interpretazioni errate.

3.2 ANTLR

Per poter generare il parser di una linguaggio, ANTLR necessita di un file che contiene le regole del linguaggio che deve riconoscere. Un semplice esempio delle regole da utilizzare è questo:

```
1 main: expression ';' main?;
2 expression : assignment | addition | subtraction;
3 assignment : Variable '=' Variable | Number;
4 addition : Variable '=' Variable | Number '+' Variable | Number;
5 subtraction : Variable '=' Variable | Number '-' Variable | Number;
6 Variable: [a-zA-Z] [a-zA-Z0-9]*;
7 Number : '-'? [0-9]+;
8 WS: [ \t\n\r]+ -> skip;
```

In ANTLR le variabili con l'iniziale minuscola servono per definire le regole grammaticali, mentre quelle con l'iniziale maiuscola definiscono regole lessicali. Il carattere "|" indica la presenza di più opzioni e il punto interrogativo indica che un elemento è facoltativo.

Se volessimo utilizzare ANTLR per generare il parser dell'esempio precedente otterremmo un parser che legge dei file che contengono solo assegnamenti, sottrazioni e addizioni; queste però possono essere definite solo una alla volta e devono terminare con il punto e virgola. La regola nella *riga 8* indica al parser di ignorare linee vuote e spazi.

Visualizziamo ora un file con le regole descritte precedentemente:

```
1 a = 5;
2 b = a+2;
```

Quando il parser ottenuto da ANTLR leggerà il file creerà un albero come quello nella figura 3.2.

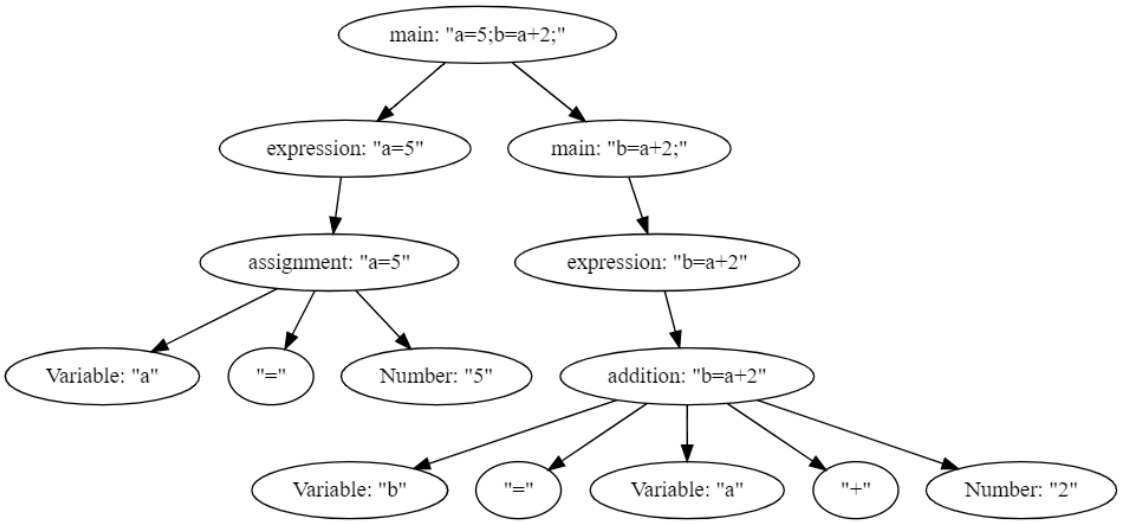

Figura 3.2: Albero ottenuto dal parser

Nel momento in cui viene eseguito ANTLR genererà diversi file, tra essi c'è il Listener, che contiene una funzione di enter e di exit per ogni variabile definita nella grammatica, esse vengono eseguite nel momento che viene incontrato il nodo dell'albero che corrisponde a quella variabile. Queste funzioni però sono vuote, è necessario creare un oggetto che estenda il Listener e che esegua l'override sulle funzioni interessate. Dopo che questo oggetto è stato creato lo si può passare come argomento al Walker (generato da ANTLR), che percorrerà l'albero generato dal parser e eseguirà le funzioni del Listener.

In particolare per ogni nodo verrà eseguita la rispettiva funzione "enter" nel momento in cui il walker entrerà nel nodo, ed eseguirà la sua funzione "exit" una volta che saranno visitati tutti i figli del nodo.

3.3 Tkinter

L'interfaccia di Tallulah è stata sviluppata utilizzando tkinter, la libreria standard per creare un'interfaccia grafica in Python3 [7].

Per creare una finestra con tkinter è necessario istanziare l'oggetto **Tk**, esso sarà l'istanza principale della schermata, si dovranno poi creare dei widget da aggiungere allo schermo.

Tramite il seguente codice è possibile creare una semplice interfaccia con la scritta "Hello World!" e a seguire un bottone per terminare il programma.

```
1  from tkinter import *
2  from tkinter import ttk
3  root = Tk()
4  # Viene creato un frame nella schermata principale
5  frm = ttk.Frame(root, padding=10)
6  frm.grid()
7  # Viene creata un'etichetta nella prima riga e prima colonna del frame
8  ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
9  # Viene creato un bottone nella prima riga e seconda colonna del frame
10 ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
11
12 # Inizializza il loop principale della schermata
13 root.mainloop()
```

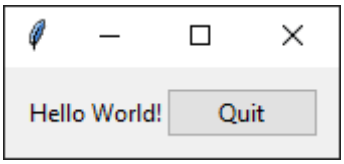

Figura 3.3: Schermata di esempio di Tkinter

Esiste un'ampia varietà di widget per tkinter, a cui è possibile associare una funzione come handler di eventi (come per i bottoni), oppure oggetti che restituiscono diversi valori a seconda dello stato di un widget (è possibile ottenere un valore booleano che indica se una checkbox è contrassegnata).

Leggere e salvare file La libreria di tkinter offre inoltre l'utilizzo di un oggetto chiamato **filedialog**. Tramite filedialog è possibile aprire una finestra per esplorare i file locali per ottenere il path di un file da aprire o per salvare un file. Per ottenere il path del file aprire è sufficiente richiamare la funzione **askopenfilename**, per salvare un file **asksaveasfile** e richiamare il metodo **write** dell'oggetto ottenuto.

Capitolo 4

Il tool

Tallulah è un software sviluppato in Python 3 che prende in input un file DOT dove determinati commenti contengono le relazioni di indipendenza (come descritto nella sezione 4.3).

4.1 Interfaccia utente

Appena si esegue Tallulah viene visualizzata una schermata che contiene solo due bottoni: uno per caricare un file ed uno per chiudere il programma.

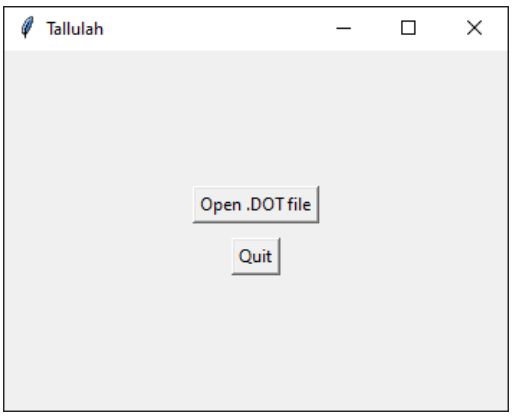

Figura 4.1: Schermata iniziale di Tallulah

Una volta caricato il file desiderato, nella nuova schermata saranno presenti:

- **L'immagine del grafo:** con tutte le impostazioni grafiche definite nel file originale.
- **Le proprietà:** ognuna ha una checkbox che deve essere selezionata se si vuole controllare la sua validità.

- **Il bottone per il controllo:** una volta cliccato aprirà una finestra di log con i risultati del controllo delle proprietà selezionate. Le proprietà però vengono controllate utilizzando lo stesso grafo di partenza: quindi i controlli delle proprietà successive non terranno conto degli errori rivelati precedentemente.

- **Le relazioni di indipendenza:** visibili nella parte inferiore.

È anche possibile aprire un nuovo file tramite la tendina in alto a sinistra.

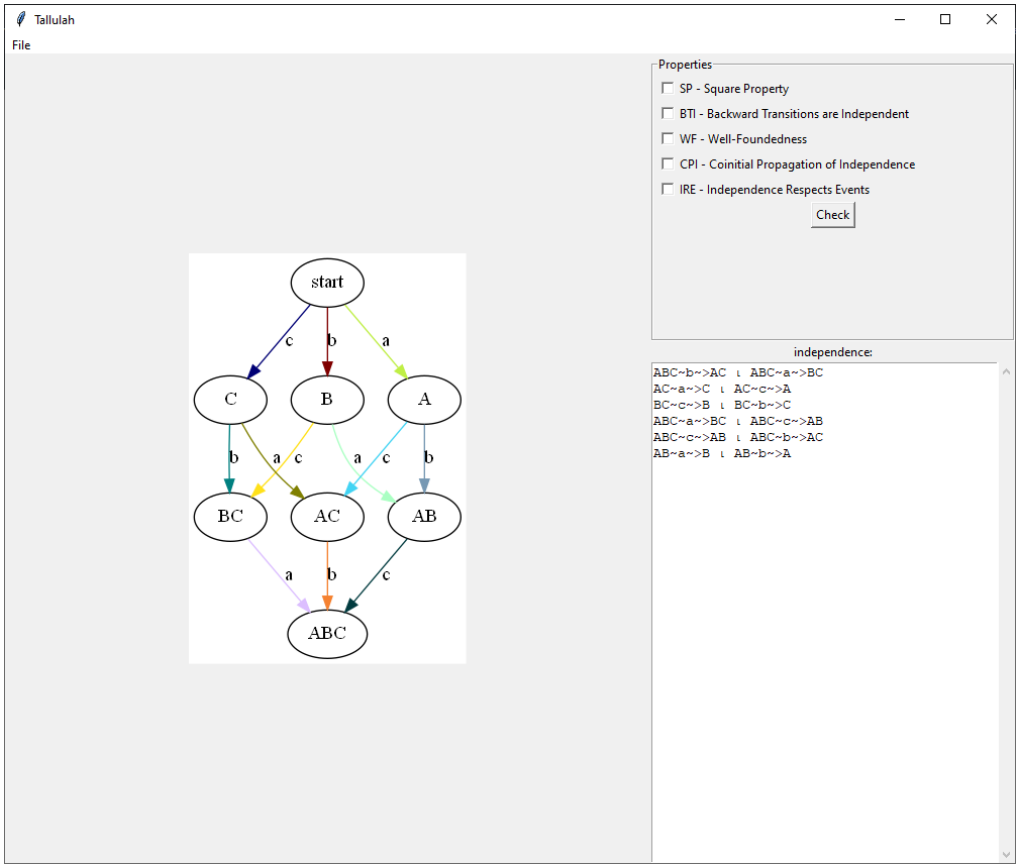

Figura 4.2: Schermata principale di Tallulah

Una volta che si sarà cliccato il bottone per il controllo delle proprietà, verrà creata una nuova finestra che conterrà:

- **Risultati:** per ogni proprietà selezionata verranno visualizzati dei messaggi che indicheranno se è valida e, nel caso non lo sia, si indicherà un controesempio. Ognuno avrà una checkbox, la checkbox sarà disabilitata se il relativo cambiamento non può essere apportato.

- **Bottone di salvataggio:** è possibile salvare in formato testuale il messaggio presente nel log.

- **Bottone per applicare le modifiche:** cliccandolo sarà possibile aggiungere/rimuovere nodi e archi, in base a quali checkbox sono state selezionate. Successivamente si genererà un nuovo file DOT con le modifiche effettuate.

Figura 4.3: Finestra con i risultati del controllo delle proprietà

Quando si applicheranno le modifiche dei file di log gli archi appartenenti allo stesso evento nel nuovo grafo generato saranno colorati con lo stesso colore. Sono disponibili 16 colori predefiniti, se un grafo dovesse possedere più di 16 eventi i successivi avranno un colore casuale.

Nella figura 4.4 è possibile vedere l'immagine del grafo che viene generato applicando SP, WF, BTI e CPI al grafo della figura 4.2.

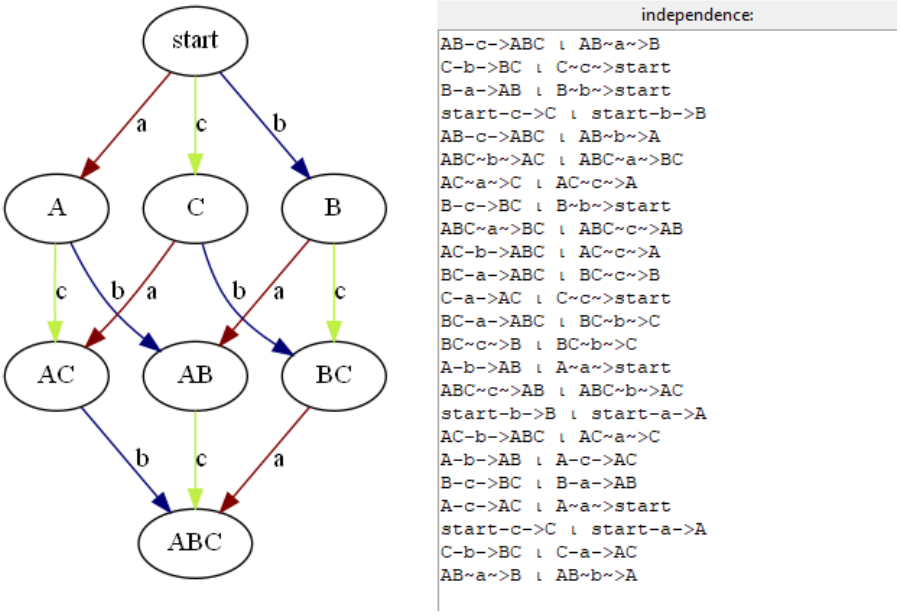

Figura 4.4: Grafo generato applicando le modifiche

4.2 Struttura del tool

Il linguaggio utilizzato per lo sviluppo di Tallulah è Python 3. Si è preferito utilizzare questo linguaggio di programmazione per due motivi principali: la presenza di librerie come Graphviz e ANTLR, che hanno semplificato l'implementazione della parte relativa alla generazione di immagini dei grafi e del parser dei file; e il fatto che Python fosse un linguaggio ad alto livello, questo ha permesso di avere un codice che potesse utilizzare delle definizioni molto fedeli alle regole matematiche di partenza. Utilizzare un linguaggio a basso livello avrebbe complicato l'implementazione perché avrebbe reso più difficile comprendere se un errore fosse causato dalla definizione del grafo o dall'implementazione dei controlli.

Il codice sorgente di Tallulah è composto da circa 2800 righe di codice, circa 1700 di queste generate da ANTLR. Com'è possibile vedere nelle figura 4.5 il programma è stato strutturato seguendo il design pattern del Model-View-Controller [10, Sezione 6.3].

View La view gestisce tutto ciò che riguarda l'interfaccia grafica (finestre, widget di Tkinter e dialog per aprire e salvare i file).

StartUI è la funzione che inizializza la schermata iniziale e assegna la funzione OpenFile come handler del bottone per caricare l'immagine.

OpenFile apre la finestra per selezionare il file da caricare e successivamente chiama MainUI, che crea l'interfaccia principale per il grafo.

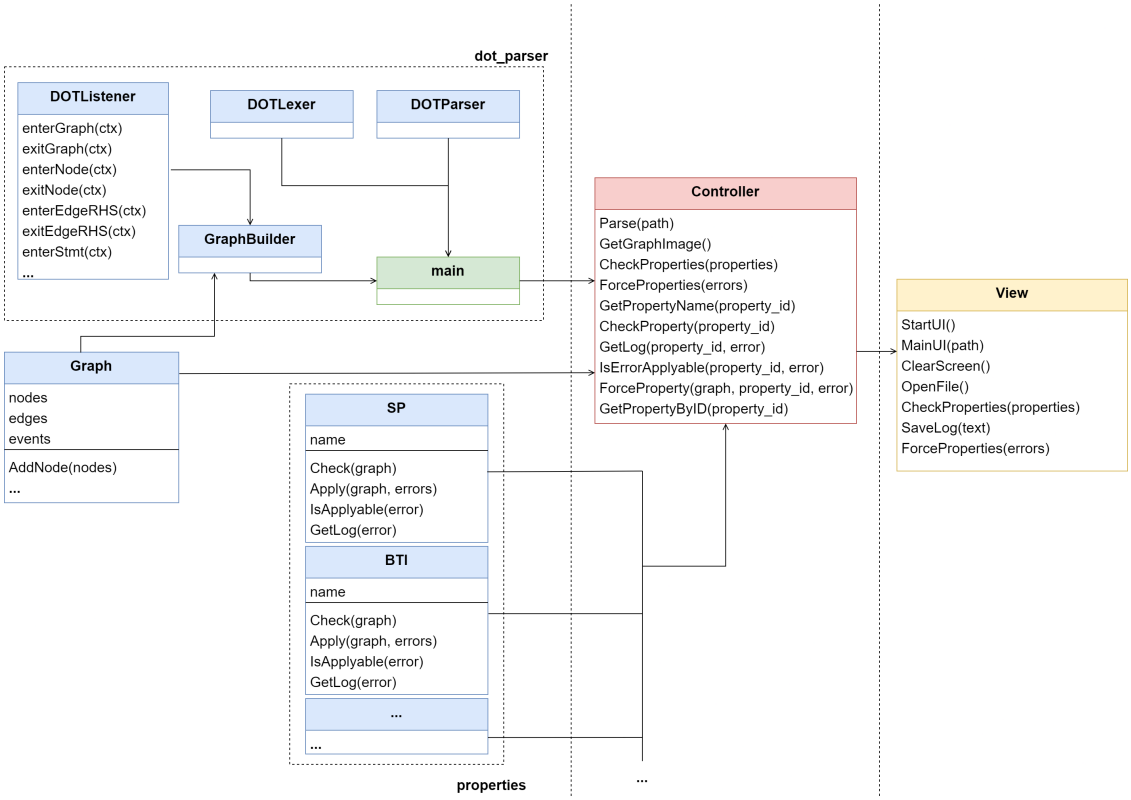

Figura 4.5: Struttura del programma

La prima volta che viene eseguito **MainUI** vengono creati tutti i widget necessari. Al caricamento di un nuovo grafo all'interno di MainUI sono eliminati (all'interno di **ClearScreen**) e ricreati solo l'immagine del grafo e il testo con le relazioni di indipendenza relativi al grafo caricato.

Al pulsante per controllare le proprietà viene assegnato **CheckProperties**, in questa funzione vengono inviati gli id delle proprietà selezionate al Controller e viene ritornato il risultato dei controlli effettuati sul grafo, essi sono visualizzati in una nuova finestra.

All'interno del Log vengono creati i bottoni con assegnati SaveLog e ForceProperties: le due funzioni svolgono rispettivamente i seguenti compiti:

SaveLog: Crea un file di testo contenente le informazioni del Log e mostra un dialog per selezionare dove salvare il file.

ForceProperties: Passa grafo, errori e informazioni riguardanti quali errori sono stati selezionati al controller, esso applica i cambiamenti degli errori selezionati e ritorna il nuovo grafo in forma testuale. Infine viene creato il dialog per salvare in un file DOT il testo così ottenuto.

Controller Il controller si occupa di manipolare le informazioni che riceve dalla view e dai file DOT. Contiene 2 attributi: graph, l'oggetto che rappresenta l'LTSI caricato dal file, e path, contenente il percorso assoluto del file caricato.

- **Parse:** riceve dalla view il parametro path, lo salva nel proprio attributo e richiama la funzione main del dot_parser, successivamente salva l'oggetto ottenuto nell'attributo graph.
- **GetGraphImage:** con la libreria di Graphviz si crea un'immagine in formato .png del file DOT utilizzando il path salvato precedentemente e ritorna il path dell'immagine così generata.
- **CheckProperties:** ha come argomento l'elenco delle proprietà selezionate nella view. Se la proprietà è stata selezionata si esegue la funzione "Check" della proprietà corrispondente, la quale richiede in input il grafo, per ritornare una lista di errori, se non ve ne sono viene ritornata una lista vuota.
- **GetLog:** per ogni errore esegue la funzione "GetLog" della rispettiva proprietà per ottenere dei messaggi da ritornare alla view in una tupla contenente porzioni di testo e il colore con cui il testo deve essere visualizzato.
- **ForceProperties:** data la lista degli errori e le informazioni sui loro checkbox, per ogni errore selezionato esegue la funzione "Apply" della relativa proprietà.
- **Funzioni per interagire con le proprietà:** Le funzioni rimanenti servono per fornire un'interfaccia che verrà utilizzata dalla View per ottenere informazioni sulle proprietà e sugli errori.

L'oggetto properties L'oggetto "properties" viene definito nella view per essere passato come argomento al controller nella funzione "CheckProperties". Questo oggetto è un dizionario che ha come chiavi gli acronimi dei nomi delle proprietà e ognuna ha come valore un oggetto "BooleanVar". I BooleanVar sono classi di tkinter che possono essere assiociate ad una checkbox inserendole come parametro del costruttore, è possibile ottenere lo stato delle checkbox sotto forma di variabile booleana semplicemente richiamando la funzione get dei BooleanVar. Quindi dato l'oggetto properties, per sapere se la checkbox della proprietà "SP" è etichettata, nel controller viene eseguita l'istruzione "properties['SP'].get()".

L'oggetto errors L'oggetto che contiene gli errori trovati nel controllo delle proprietà è chiamato "errors". Errors è un dizionario e, come per properties, ha gli acronimi delle proprietà come chiavi. Tuttavia gli elementi di errors sono degli insiemi di tuple. A seconda della proprietà alla quale si riferiscono contengono un numero diverso di elementi. Questi elementi non contengono solo gli errori, ma anche informazioni utili da segnalare all'utente.

L'oggeto check_values Esso serve per riconoscere quale errore è stato selezionato nella schermata di log. Come gli oggetti appena descritti è un dizionario con chiavi gli identificativi delle proprietà. I valori di check_values sono liste di BooleanVar, contenenti i valori delle checkbox degli errori. Quindi "check_values['SP'][0].get()" sarà True se il primo errore della proprietà Square Property è selezionato.

dot_parser All'interno della directory "dot_parser" sono presenti i file generati da ANTLR per eseguire il parsing dei file. Tramite **DOTLexer** e **DOTParser** sarà possibile ottenere le informazioni del file organizzate in un albero, queste informazioni vengono visitate tramite un oggetto della libreria di ANTLR. Quest'ultimo eseguirà le funzioni di un **DOTListener**. **GraphBuilder** è un oggetto che estende DOTListener, in quanto il DOTListener presenta funzioni vuote. Come avviene il parsing è illustrato in dettaglio nella sezione 4.3.

Proprietà Per ogni proprietà è stato creato un oggetto nella directory "properties". Gli oggetti presenti possiedono un solo attributo contenente il nome completo e le funzioni:

- **Check:** controlla se nel grafo è valida la proprietà e ritorna gli errori trovati
- **Apply:** dato un errore e un grafo esegue le azioni necessarie a rimuovere l'errore.
- **IsApplyable:** dato un errore ritorna True se è possibile rimuoverlo.
- **GetLog:** dato un errore ritorna una lista di tuple contenenti porzioni di testo che descrivono l'errore e il colore con il quale debbano essere visualizzate.

Graph Si tratta della struttura dati principale per poter implementare l'LTSI (sezione 2.2). Contiene informazioni riguardanti i nodi, archi, relazioni di indipendenza e appartenenza a eventi. L'implementazione è analizzata in dettaglio nella sezione 4.4.

4.3 Parsing del file DOT

Per ottenere il parser di Tallulah si è utilizzato ANTLR. Per generare un parser ANTLR necessita di un file contenente la grammatica di riferimento, quindi si è partiti dalle caratteristiche del linguaggio DOT fornite da Graphviz [8]. Però il linguaggio DOT standard non è idoneo per i seguenti motivi:

- **Il linguaggio DOT supporta grafi indiretti.** I grafi descritti nei file DOT che dobbiamo leggere le transizioni devono avere una direzione ben definita, quindi si è ristretto il linguaggio ai soli grafi diretti.

- **Le transizioni possono non essere etichettate.** Nella nostra teoria di riferimento ogni transizione è etichettata, quindi nel nuovo linguaggio è necessario etichettare ogni transizione.

- **Il linguaggio DOT non supporta l'indipendenza.** Per mantenere il nuovo linguaggio compatibile con i software che utilizzano il linguaggio DOT standard (lo stesso Graphviz per esempio), le informazioni riguardanti l'indipendenza sono inserite all'interno di commenti multi-line. Questo però comporta che i essi non debbano essere interpretati come commenti da Tallulah, non è quindi possibile utilizzarli nel metodo tradizionale.

Il linguaggio che otteniamo da questa premessa è il seguente:

```
1  graph  :  'strict'? 'digraph' identifier '{' stmt_list '}' ('/*'
       independence_list? '*/')?;
2
3  stmt_list: (stmt ';'? stmt_list)? ;
4  independence_list: independence (';' | ',')? independence_list?;
5  stmt : node_stmt | edge_stmt | attr_stmt | assignment | subgraph_stmt ;
6  attr_stmt : ('graph' | 'node' | 'edge') attr_list ;
7  attr_list : '[' a_list? ']' attr_list?;
8  attr_label : '[' a_label ']' attr_list?;
9  a_list : assignment (';'|',')? a_list?;
10 a_label : ('label' | '"label"') '=' identifier (';'|',')? a_list?;
11 edge_stmt : (node_id | subgraph_stmt) edgeRHS attr_label;
12 edgeRHS : '->' (node_id | subgraph_stmt) edgeRHS?;
13 node_stmt : node_id attr_list?;
14 node_id : identifier port?;
15 port : ':' identifier (':' identifier)?;
16 subgraph_stmt : ('subgraph' identifier?)? '{'stmt_list'}';
17 assignment : identifier '=' identifier;
18 independence: independence_edge'/'independence_edge;
19 independence_edge: Direction identifier '-' identifier '->' identifier ;
20 identifier : Variable | Quote | Number ;
21
22 Quote: '"' (~["\\] | '\\' .)* '"';
23 Variable: [a-zA-Z] [a-zA-Z0-9]*;
24 Number : '-'? (('.'[0-9]+) | [0-9]+('.'[0-9]*)?);
25 Direction : '<' | '>';
26 WS: [ \t\n\r]+ -> skip;
27 LINE_COMMENT: '//' ~[\r\n]* -> skip;
```

Commenti I commenti multi-line non sono più interpretati dal parser come commenti ma come parte del codice. I commenti inline tuttavia rimangono ancora possibili, anche dentro la definizione dell'indipendenza, in quanto gli altri software ignoreranno il commento inline trattandolo come parte del commento multi-line.

Indipendenza Le relazioni di indipendenza devono essere descritte solo una volta che si è finito di descrivere il grafo. Due archi indipendenti sono separati dal simbolo '/'. Ogni arco viene preceduto da un simbolo che indica se si tratta di una transizione forward o backward (rispettivamente '>' o '<').

A seguire un esempio di definizione di indipendenza:

```
1 /*
2 //BTI
3 < G-b->E / < G-a->F
4 */
```

Nell'esempio si dichiara una sola relazione di indipendenza tra due transizioni backward. Come spiegato precedentemente è possibile anche inserire un commento all'interno della definizione dell'indipendenza, si potrebbe usare come nell'esempio per segnalare che l'indipendenza descritta serve a soddisfare BTI.

Implementazione del Listener Una volta generati i file di ANTLR si è dovuto implementare il Listener. Il Listener funziona in questo modo: una volta che dal file si è ottenuto un albero che segue le definizioni della grammatica, si inizia ad esplorare in profondità l'albero. Quando si entrerà in un nodo verrà eseguita la funzione "enter" relativa al tipo di nodo e quando si saranno finiti di esplorare i figli si uscirà, richiamando la funzione "exit".

Per implementare il Listener quindi si è dovuto decidere di quali funzioni eseguire l'override, e come queste avrebbero dovuto salvare le informazioni.

Lo schema della figura 4.6 visualizza graficamente l'ordine con cui vengano richiamate le funzioni "enter" ed "exit". Lo schema comporta leggere modifiche volte a semplificare la comprensione di come le funzioni vengano richiamate, infatti nodi come "assignment" e "port" complicherebbero il grafo senza fornire informazioni essenziali, per questo motivo non sono stati rappresentati.

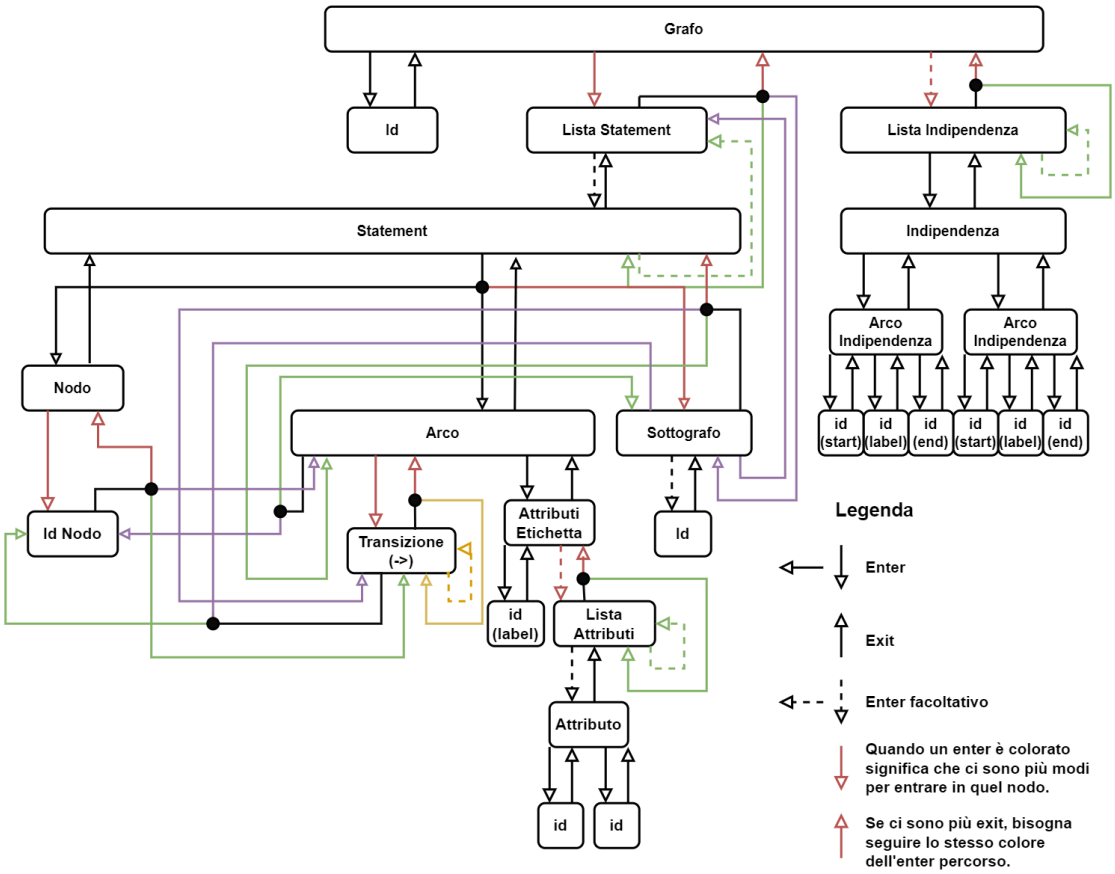

Figura 4.6: Ordine di chiamata delle funzioni del Listener

Per riuscire a ricostruire l'LTSI i nodi dell'albero risultante dal parsing che risultano rilevanti sono:

- **Arco (edge_stmt):** Contiene le informazioni principali sulle varie transizioni presenti nel grafo.
- **Sottografo (subgraph_stmt):** È necessario tenere in memoria informazioni riguardo l'inizio e la fine di un sottografo, perché potrebbe essere stato definito all'interno della definizione di un arco.
- **Transizione (edgeRHS):** Le informazioni di questo nodo sono essenziali perché contiene la destinazione di una transizione.
- **Id del Nodo (node_id):** Contenendo gli identificativi dei nodi è ovvio che le informazioni contenute in questo nodo sono importanti, anche se non fanno parte della definizione di un arco (questo perché potrebbero fare parte di un sottografo definito all'interno di un arco).

- **Attributo Etichetta (a_label):** Da questo nodo è necessario salvare l'etichetta della transizione (o delle transizioni, se ne sono presenti più di una).

Dopo aver individuato i nodi dei quali eseguire l'override del Listener è necessario chiarire come salvare le loro informazioni tenendo conto delle seguenti complicazioni:

- **Definizione di più archi.** Dalla grammatica è possibile notare come *edgeRHS* sia ricorsiva. Questo implica che è possibile definire più di un arco all'interno del nodo *edge_stmt*, però l'etichetta di tutti gli archi si trova dopo l'ultimo *edgeRHS*.

- **Sottografi nella definizione di archi.** Nella definizione di un arco potrebbe essere presente un sottografo invece di un nodo, questo sottografo a sua volta può contenere nodi o altre definizioni di archi.

Per memorizzare il livello di profondità dei nodi è stata creata la lista **nodes_depth**. Per potere salvare archi definiti consecutivamente invece si sarebbe potuto optare per salvarli di volta in volta, ma nel caso dovesse presentarsi un sottografo si sarebbe dovuto interrompere la costruzione dell'arco per poi analizzare il sottografo. Per contenere l'ordine dei nodi è stato creato **nodes_order**.

GraphBuilder oltre ad effettuare gli override delle funzioni di DotListener possiede 4 attributi utilizzati durante la visita dell'albero di parsing:

- **nodes_order:** lista di insiemi per tenere traccia dell'ordine dei nodi

- **nodes_depth:** lista di insiemi che contengono i nodi definiti nello stesso livello di incapsulamento di sottografi o nei livelli inferiori.

- **edges:** lista di insiemi per salvare temporaneamente gli archi definiti dei quali ancora non è nota l'etichetta.

- **label:** attributo che contiene il valore dell'ultima etichetta incontrata.

A seguire lo pseudo-codice dei metodi dell'oggetto GraphBuilder (ctx contiene le informazioni presenti nel nodo dell'albero).

Algorithm 1 enterEdge_stmt(ctx)

Aggiungi un *insieme vuoto* in coda a $self.nodes\_order$
Aggiungi un *insieme vuoto* in coda a $self.nodes\_depth$
Aggiungi un *insieme vuoto* in coda a $self.edges$

Entrando nella definizione di un arco si devono ignorare tutte le informazioni precedentemente salvate, quindi le informazioni presenti all'interno del nodo saranno salvate in

nuovi elementi delle tre liste, questi nuovi elementi saranno rimossi una volta usciti dal nodo "edge_stmt".

Algorithm 2 exitEdge_stmt(ctx)

```
Rimuovi ultimo elemento di self.nodes_order
nodes ← Rimuovi ultimo elemento di self.nodes_depth
edges ← Rimuovi ultimo elemento di self.nodes_edges
if self.nodes_depth ha almeno un elemento then
    for node ∈ nodes do
        self.nodes_depth[−1].add(node)        ▷ L'indice "-1" indica l'ultimo elemento
    end for
end if
for edge ∈ edges do
    Crea un arco da edge.start a edge.end con etichetta self.label
end for
```

Una volta terminata la definizione si creeranno nel grafo gli archi salvati temporaneamente in **edges** aggiungendo l'etichetta. Gli elementi delle liste aggiunti precedentemente vengono rimossi, ma i nodi vengono salvati nel livello superiore (il nuovo ultimo elemento di **nodes_depth**), questo perché sono comunque nodi definiti all'interno del grafo (o sottografo).

Algorithm 3 enterSubgraph_stmt(ctx)

```
Aggiungi un insieme vuoto in coda a self.nodes_order
Aggiungi un insieme vuoto in coda a self.nodes_depth
Aggiungi un insieme vuoto in coda a self.edges
```

Come per enterEdge_stmt una volta entrati in un sottografo bisogna ignorare le informazioni definite precedentemente aggiungendo dei nuovi insiemi alle tre liste.

Algorithm 4 exitSubgraph_stmt(ctx)

```
nodes ← Rimuovi ultimo elemento di self.nodes_depth
Rimuovi ultimo elemento di self.nodes_order
if self.nodes_order e self.nodes_depth non sono vuoti then
    for node ∈ nodes do
        self.nodes_depth[−1].add(node)
        self.nodes_order[−1].add(node)
    end for
end if
Rimuovi ultimo elemento di self.nodes_edges
```

Si rimuovono gli elementi creati nella funzione di enter e si salvano i nodi di quel livello nell'ultimo insieme di **nodes_depth**. Vengono salvati anche in **nodes_order** in modo che i nodi definiti nel grafo possano essere usati per creare gli archi.

Algorithm 5 enterNode_id(ctx)

if *self.nodes_order* e *self.nodes_depth* non sono vuoti **then**
  self.nodes_depth[−1].*add*(nodo contenuto in ctx)
  self.nodes_order[−1].*add*(nodo contenuto in ctx)
end if

Viene salvato l'identificativo del nodo nell'ultimo insieme di **nodes_depth** e **nodes_order**.

Algorithm 6 enterEdgeRHS(ctx)

Aggiungi un *insieme vuoto* in coda a *self.nodes_order*

EdgeRHS non crea un nuovo livello sottostante, ma solo un nuovo arco, quindi si deve creare un nuovo elemento in **nodes_order** per la destinazione di questo arco.

Algorithm 7 exitEdgeRHS(ctx)

end_nodes ← Rimuovi ultimo elemento di *self.nodes_order*
start_nodes ← *self.nodes_order*[−1]
for *start* ∈ *start_nodes* **do**
  for *end* ∈ *end_nodes* **do**
    self.edges[−1].*add*((*start*, *end*))
  end for
end for

Si rimuove l'ultimo elemento di **nodes_order** (quello che è stato aggiunto nella funzione "enter") e si creeranno gli archi dai nodi contenuti nell'ultimo elemento di nodes_order a quelli contenuti nell'insieme rimosso. Dato che ancora non è nota l'etichetta questi archi vengono salvati temporaneamente nell'ultimo elemento di edges.

Algorithm 8 enterA_label(ctx)

self.label ← etichetta contenuta in ctx

In questo momento è sufficiente salvare l'etichetta in una variabile, dopo la definizione dell'etichetta inizieranno i vari **exitEdgeRHS** e **exitEdge_stmt** che utilizzeranno quella variabile per creare gli archi etichettati nel grafo.

Grazie alle definizioni di queste funzioni quando un nodo viene definito e deve essere utilizzato per la creazione di un arco si troverà in un insieme di **nodes_order** che sarà utilizzato per quello scopo. Se il nodo dovesse essere definito nel grafo principale o in un sottografo senza fare parte di alcun arco l'insieme di **nodes_order** in cui sarà salvato viene scartato e il nodo di conseguenza ignorato.

Leggere e salvare l'indipendenza dal file è molto più semplice, questo perché gli archi devono essere scritti due alla volta, è sufficiente salvare l'arco quando si entra nel nodo **Independence_edge** e aggiungere l'indipendenza tra i due archi uscendo dal nodo **Independence**.

4.4 Implementazione LTSI

Per implementare l'LTSI è stata creata la classe Graph, in modo che potesse gestire la presenza di nodi, archi, relazioni di indipendenza ed eventi.

Archi All'interno di Tallulah gli archi vengono rappresentati tramite delle tuple nella forma:

(**start**, **label**, **end**, **is_forward**)

Start, label ed end sono rappresentati da una stringa e indicano rispettivamente: il nodo iniziale, l'etichetta dell'arco ed il nodo finale. Is_forward è rappresentato da un valore booleano: True se si tratta di una transizione forward, False se di una backward.

Nonostante Graph prenda in input e restituisca gli archi nella forma appena descritta, all'interno della classe gli archi non sono salvati in una lista di tuple. Per migliorare l'efficienza durante la ricerca degli archi si è preferito salvarli tramite un dizionario.

L'attributo **edges** è un dizionario che prende in input una tupla contenente il nodo di partenza e quello di fine, restituendo una lista di stringhe. Le stringhe così ottenute sono le etichette delle transizioni dal primo al secondo nodo, nel caso non ci siano transizioni la tupla non sarà presente tra le chiavi oppure restituirà una lista vuota.

Per occupare meno spazio gli archi che vengono salvati sono esclusivamente forward, ottenendo nel caso servissero, le transizioni backward invertendo inizio e fine di un arco.

Per ottenere gli archi dell'LTSI è possibile utilizzare le funzioni:

- **GetEdgesFrom:** prende in input un nodo di partenza e due valori booleani che, in caso non vengano esplicitati saranno impostati a True. Il primo è "all", se "all" è True ritornerà sia le transizioni forward sia quelle backward che iniziano dal nodo, se è False il secondo valore ("only_forward") è la discriminante per decidere se ottenere transizioni forward o backward.

- **GetEdgesBetween:** ritorna le transizioni da un nodo iniziale ed uno finale presi in input. Possiede anche gli argomenti "all" e "only_forward", come per GetEdgesFrom.
- **EdgeExists:** prende in input un arco, ritorna True se l'arco è presente nel grafo, False altrimenti.

La seconda funzione potrebbe risultare ridondante dato che dato lo stesso nodo di partenza le transizioni di **GetEdgesBetween** sono contenute anche in **GetEdgesFrom**, ma se sono già noti nodo di partenza e di arrivo **GetEdgesBetween** è più efficiente.

Nodi Ogni volta che viene aggiunto un arco i nodi di partenza e di arrivo vengono salvati in un insieme di stringhe che rappresenta l'insieme dei nodi del grafo.

Indipendenza Se due archi hanno una relazione di indipendenza viene salvata una tupla che li contiene, questa tupla viene poi aggiunta ad un insieme chiamato **independence**.

Dato che l'indipendenza è una relazione simmetrica nel momento in cui si aggiunge l'indipendenza tra un arco t e un arco u, prima di inserire la tupla (t,u) si controlla che non sia presente (u,t). Per lo stesso motivo quando si vuole controllare che t ed u siano indipendenti si deve ricercare sia la tupla (t,u) che la tupla (u,t).

Non è possibile salvare un arco indipendente con sè stesso perché l'indipendenza è irriflessiva.

Eventi Una volta letto il file e costruito il grafo verrà richiamata la funzione **InitEvents**. Questa funzione effettuerà i controlli della definizione dell'evento (definizione 2.3.1). Gli eventi sono salvati come insiemi di archi e sono contenuti nella lista **events**. Se durante la ricerca due archi fanno parte dello stesso evento: se i loro eventi sono già stati creati si rimuovono e si salva la loro unione, se ne è stato creato solo uno si aggiunge l'arco mancante all'evento creato, altrimenti si crea un nuovo evento contenente i due archi nella lista.

4.5 Controllo delle proprietà

Per ogni proprietà implementata nel tool è stato creato un oggetto contenente un attributo contenente il nome completo della proprietà e 4 funzioni statiche:

- **Check:** Prende in input il grafo e controlla che nel grafo la proprietà sia valida, nel caso sia valida verrà ritornato un insieme vuoto, altrimenti l'insieme conterrà le informazioni necessarie per fornire un controesempio.
- **IsApplyable:** Prende in input un errore e ritorna True se è possibile modificare il grafo per rimuovere l'errore.

- **Apply:** Prende in input un grafo e un errore trovato utilizzando "Check", la funzione esegue dovute modifiche al grafo in modo da rimuovere l'errore.
- **GetLog:** Prende in input un errore trovato utilizzando "Check" e ritorna una lista di tuple contenenti porzioni di testo che descrivono l'errore e il colore con cui questi testi debbano essere colorati.

In seguito si analizzeranno in dettaglio le tipologie di errori e le implementazioni di Check e Apply delle varie proprietà.

4.5.1 SP - Square Property

Controllo proprietà Per effettuare il controllo di SP (assioma 2.3.1) all'interno del grafo viene svolto il seguente procedimento:

1. Si crea la lista vuota di errori
2. Si itera per ogni nodo "node" del grafo
3. Si ottiene la lista delle transizioni che iniziano da "node"
4. Se due transizioni u, t sono indipendenti:
   (a) Si cicla per ogni nodo "end" del grafo
   (b) Si controlla la presenza delle transizioni u' e t' (seguendo la regola dell'assioma) che terminano in "end".
   (c) Se u', t' esistono si procede a controllare altre coppie di transizioni, SP è valido per la coppia u, t.
   (d) Se è stato trovato almeno un candidato tra t' e u' si aggiunge alla lista di errori l'errore 1 (descritto in seguito) per ogni transizione mancante. In questo modo è possibile la chiusura di SP rispetto a questi archi.
   (e) Indipendentemente dal fatto che si siano trovati dei possibili t' o u' si aggiunge anche l'errore 2 (descritto in seguito) per u' e t' alla lista di errori. In questo modo è possibile chiudere SP creando un nuovo nodo.

Tipologie di errori Gli errori possibili per SP sono due:

- **Errore 1: (u, t, edge, False)** Se durante la ricerca di u' e t' solo uno dei due è presente allora la transizione mancante viene salvata come **edge**. Le transizioni u e t servono solo per fornire informazioni all'utente riguardo la presenza dell'errore.
- **Errore 2: (u, t, edge, True)** L'arco **edge** ha come destinazione un nodo non ancora presente nel grafo.

Risoluzione degli errori Per poter risolvere entrambi gli errori si aggiunge la transizione **edge** al grafo. Per rendere SP valido è comunque necessario che per l'errore 2 si selezionino sia l'errore per u' sia per t', altrimenti mancherà una transizione per rendere valida la proprietà.

4.5.2 BTI - Backward Transitions are Independent

Controllo proprietà BTI (assioma 2.3.2) viene controllato ciclando su tutti i nodi del grafo e controllando che tutte le transizioni backward di quel nodo siano indipendenti tra loro, nel caso non lo siano viene memorizzato l'errore.

Tipologie di errori

- **Errore: (edge1, edge2)** Le due transizioni sono due transizioni backward coiniziali non indipendenti.

Risoluzione di errori Per forzare la validità di BTI si aggiunge la relazione di indipendenza tra le due transizioni.

4.5.3 WF - Well-Foundedness

WF (assioma 2.3.3) implica che dato un nodo non esistano percorsi all'indietro di lunghezza infinita. I file che descrivono l'LTSI sono per natura finiti, quindi l'unico modo in cui è possibile ottenere una mancata validità di WF è che nel grafo sia presente un ciclo.

Controllo proprietà L'algoritmo del controllo di WF nel grafo inizia creando una lista di nodi da visitare, inizialmente uguale alla totalità dei nodi del grafo. Si sceglie un nodo casuale e si effettua una visita in profondità (algoritmo DFS). Nel caso si dovesse trovare un ciclo nel grafo si salveranno i nodi che fanno parte del ciclo. Dopo che i nodi sono stati visitati vengono eliminati dalla lista dei nodi da visitare. Il procedimento si ripete finché la lista non diventa vuota.

Dopo aver trovato un ciclo si ottengono gli archi che lo formano e vengono restituiti come errore.

Tipologie di errori

- **Errore: edge** Gli errori che otteniamo sono costituiti da sole transizioni: quelle che formano il ciclo.

Risoluzione di errori Per forzare WF è sufficiente rimuovere gli archi presenti negli errori (l'utente può selezionare dalla schermata di log quali di questi archi rimuovere). Da notare che nel momento in cui si richiama la funzione **RemoveEdge** della classe **Graph** si rimuoveranno le transizioni forward, backward e le relazioni di indipendenza in cui sono presenti.

4.5.4 CPI - Coinitial Propagation of Independence

Controllo proprietà CPI (assioma 2.3.4) viene controllato ciclando su ogni nodo del grafo e controllando che se due nodi di quel grafo dovessero essere indipendenti e comprendere un angolo di un commuting diamond allora ci sia una relazione di indipendenza anche tra le transizioni che comprendono i due angoli adiacenti. Se non dovessero possedere una relazione di indipendenza ciò viene segnalato nell'insieme di errori.

I commuting diamond hanno 4 angoli, dato anche un solo angolo compreso tra transizioni indipendenti CPI comporta una propagazione dell'indipendenza anche al quarto angolo. Per individuare anche questi errori dovuti alla propagazione è necessario effettuare il controllo finché si hanno iterazioni in cui vengono aggiunti errori nell'insieme e trattare le mancate relazioni di indipendenza come se fossero presenti nel grafo.

Tipologie di errori

- **Errore: (u, t, $\underline{u}$, t')** Gli archi $\underline{u}$ e t' comprendono l'angolo adiacente a quello compreso da u e t. Mentre questi ultimi servono solo per chiarire all'utente quale sia la causa dell'errore, $\underline{u}$ e t' invece sono le transizioni che non hanno la relazione di indipendenza.

Risoluzione di errori Un errore di CPI viene risolto aggiungendo una relazione di indipendenza tra $\underline{u}$ e t'.

Vi è però un caso particolare in cui non è possibile forzare la validità di CPI senza apportare cambiamenti alle transizioni del grafo: se u e t hanno rispettivamente un'etichetta α e $\underline{\alpha}$ è possibile ottenere $\underline{u} = t'$. Questo richiederebbe avere $\underline{u}\ \imath\ \underline{u}$ ma per definizione la relazione di indipendenza non è riflessiva.

Quando è presente questo errore Tallulah segnala la sua presenza con una scritta rossa, inoltre in questo caso la checkbox dell'errore non sarà selezionabile.

4.5.5 IRE - Independence Respects Events

Controllo proprietà Per controllare IRE (assioma 2.3.5) è sufficiente ottenere dalla classe Graph l'insieme delle indipendenze, per ognuna si ha u, t. Si controlla che tutte le transizioni t' ~ t siano indipendenti da u, e si controlla che tutte le transizioni u' ~ u

siano indipendenti da t. Se non dovesse essere valida una relazione di indipendenza allora si aggiunge come errore.

Tipologie di errori

- **Errore: (t, t', u)** L'errore indica che t è indipendente con u. t' ~ t ma non vale la relazione t' $\imath$ u.

Risoluzione di errori Per forzare IRE è sufficiente aggiungere l'indipendenza tra le transizioni t' e u.

Nel caso in cui vale t $\imath$ u e u ~ t, questo implica u $\imath$ u. Ma questo non è possibile perché la relazione di indipendenza è irriflessiva, quindi in questo caso l'errore non può essere risolto.

Quando è presente questo errore Tallulah segnala la sua presenza con una scritta rossa, inoltre in questo caso la checkbox dell'errore non sarà selezionabile.

4.5.6 CIRE - Coinitial Independence Respects Events

Controllo proprietà Il controllo di CIRE (assioma 2.3.7) si effettua controllando tutte le possibili coppie di eventi. Per ogni transizione t appartenente al primo evento e ogni transizione u appartenente al secondo si controlla se ricadono in uno dei seguenti casi:

- t e u sono coiniziali e indipendenti
- t e u sono coiniziali e non indipendenti

L'errore si presenta solo nel secondo caso, salvando le transizioni in una lista temporanea. Se ci sono state delle coppie di transizioni coiniziali e indipendenti gli errori presenti nella lista temporanea vengono inseriti nella lista definitiva di errori. Se non vi sono state transizioni che hanno soddisfatto il primo caso allora i due eventi non sono coinitially independent, la premessa è falsa quindi gli errori presenti nella lista temporanea vengono scartati.

Tipologie di errori

- **Errore: (t, u)** L'errore indica che t e u fanno parte di due eventi coinitially independent, sono coiniziali ma non sono indipendenti.

Risoluzione di errori Per forzare la validità di CIRE si aggiunge la relazione di indipendenza tra le due transizioni.

4.5.7 IEC - Independence of Events is Coinitial

Controllo proprietà IEC (assioma 2.3.6) viene controllato ciclando su tutte le relazioni di indipendenza. Successivamente si controllano tutte le coppie di transizioni possibili appartenenti agli eventi. Il controllo può terminare con i seguenti risultati:

- Sono state trovate due transizioni coiniziali e indipendenti.
- Sono state trovate delle transizioni coiniziali ma non indipendenti. Si aggiunge l'errore 1 (vedi dopo) per ogni coppia di transizioni coiniziali alla lista di errori.
- Non sono state trovate transizioni coiniziali. Si aggiunge l'errore 2 (vedi dopo) alla lista di errori.

Per alleggerire la computazione vengono salvati i due eventi, in questo modo quando si dovranno controllare le transizioni delle prossime relazioni di indipendenza si potrà evitare di effettuare i controlli, dato che i due eventi sono già stati controllati precedentemente.

Tipologie di errori

- **Errore 1: (t_1, t_2, t_1', t_2')** L'errore indica che $t_1 \ \imath \ t_2$, $t_1' \ \sim \ t_1$, $t_2' \ \sim \ t_2$, t_1' e t_2' sono coiniziali ma non indipendenti.
- **Errore 2: (t_1, t_2, None, None)** L'errore indica che $t_1 \ \imath \ t_2$, ma i loro eventi non possiedono transizioni coiniziali.

Risoluzione di errori Tallulah risolve solo l'errore 1, aggiungendo la relazione di indipendenza tra t_1' e t_2'.

4.5.8 ID - Independence of Diamonds

Controllo proprietà Per effettuare il controllo di ID (proprietà 2.3.3) all'interno del grafo viene svolto il seguente procedimento:

1. Si crea la lista vuota di errori
2. Si itera per ogni nodo "node" del grafo
3. Si ottiene la lista delle transizioni che iniziano da "node"
4. Si cicla per ogni nodo "end" del grafo
5. Si controlla la presenza delle transizioni u' e t' (come definiti dalla proprietà) che terminano in "end".
6. Se u', t' esistono allora u e t devono essere indipendenti, se non lo sono si aggiungono alla lista di errori

Tipologie di errori

- **Errore: (t, u)** L'errore indica che t e u formano un commuting diamond ma non sono indipendenti.

Risoluzione di errori L'errore si risolve aggiungendo una relazione di indipendenza tra t e u. È impossibile che t e u siano uguali, in quanto da definizione di ID hanno almeno un nodo diverso.

4.5.9 RPI - Reversing Preserves Independence

Controllo proprietà Per controllare RPI (proprietà 2.3.4) si cicla su tutte le relazioni di indipendenza. Per ogni t, u presenti nelle tuple:

- se $\underline{t}$ == u allora si aggiunge l'errore due volte, una volta assegnando a "first_reverse" il valore "True" ed una il valore "False".
- se $\underline{t}$!=u
  - e $\underline{t}$ non indipendente con u si aggiunge l'errore assegnando a "first_reverse" il valore "True"
  - e t non indipendente con $\underline{u}$ si aggiunge l'errore assegnando a "first_reverse" il valore "False"

Tipologie di errori

- **Errore: (t, u, first_reverse)** L'errore indica che t e u sono indipendenti, ma se "first_reverse" è "True" allora $\underline{t}$ non è indipendente con u, altrimenti t non è indipendente con $\underline{u}$.

Risoluzione di errori L'errore si rimuove aggiungendo l'indipendenza tra $\underline{t}$ e u se first_reverse è "True" e aggiungendo l'indipendenza tra t e $\underline{u}$ altrimenti.

Nel caso in cui $\underline{t}$ e u siano uguali non è possibile aggiungere l'indipendenza, perché vorrebbe significare u $\imath$ u. Ma per definizione l'indipendenza è irriflessiva.

4.5.10 Aggiungere nuove proprietà

Il codice è stato scritto con lo scopo di fornire una struttura modulare, in modo che sia semplice implementare nuove proprietà per Tallulah. Questo perché la teoria a cui fa riferimento Tallulah è tutt'ora studiata, ed è possibile che si creino nuovi assiomi o si modifichino quelli già esistenti.

Per poter aggiungere una nuova proprietà è necessario (e sufficiente):

Implementare la classe della proprietà Per implementare la proprietà bisogna creare una classe nella directory "properties". La classe deve possedere un attributo "name", questo conterrà una stringa che verrà utilizzata sia nella schermata principale, sia in quella di log. Successivamente sono necessarie: Check, Apply, IsApplyable e GetLog, come descritte all'inizio della sezione 4.5.

Aggiungere la proprietà nella schermata iniziale Nella funzione "MainUI" dell'oggetto "view" viene creata la schermata principale, nel caso si volesse aggiungere una nuova proprietà è necessario creare una nuova checkbox. La checkbox deve avere come variabile una BooleanVar all'interno del dizionario "properties", la chiave che si utilizzerà rappresenterà l'identificativo della proprietà. Per le proprietà già implementate è stato utilizzato come identificativo una stringa contenente l'acronimo del nome della proprietà.

Aggiungere identificativo nel controller Infine è necessario associare l'identificativo della proprietà con la classe corrispondente. Per fare ciò è necessario aggiungere il controllo dell'id nella funzione "GetPropertyByID" e, nel caso l'id ricevuto sia uguale a quello inserito come chiave in "properties" la funzione dovrà ritornare la classe.

Capitolo 5

Conclusioni

Sono fiducioso che Tallulah possa contribuire nella ricerca e nello studio assiomatico di modelli reversibili. Nonostante sia in grado di gestire solo 9 delle proprietà illustrate nell'articolo [5] sono assiomi fondamentali per lo studio della Causal-Consinstent Reversibility e il fatto che ora possano essere controllati in modo automatizzato riduce i tempi necessari a controllare la loro validità e il rischio di compiere errori di calcolo. Inoltre la possibilità di eseguire il parsing di file DOT complessi e di ottenere le informazioni riguardanti le relazioni di indipendenza può risultare utile anche in contesti che non utilizzino necessariamente gli assiomi presi in considerazione per questo progetto.

Sicuramente in futuro Tallulah sarà migliorato: potranno essere implementati i controlli delle proprietà al momento non presenti. Potrebbero anche cambiare alcune definizioni degli assiomi già gestiti da Tallulah, questo principalmente perché la teoria a cui si fa riferimento potrebbe essere modificata alla luce di nuovi studi, magari proprio grazie a Tallulah.

Al fine di portare avanti la ricerca è stata adottata una licenza MIT per mantenere il software libero, e il codice del software descritto in questo documento è stato strutturato in modo da poter implementare nuove proprietà in maniera modulare.

Non mi era mai capitato di sviluppare un programma così complesso. Gli sforzi richiesti mi hanno portato ad attingere a piene mani da ciò che ho imparato in questi anni di studio all'università, mettendo in pratica e adattando alle mie esigenze gli argomenti trattati in gran parte dei corsi.

Proprio grazie a questa necessità di doversi adattare in base ai problemi che si ponevano lungo il tragitto ritengo che l'esperienza acquisita con questo progetto mi abbia soddisfatto e arricchito culturalmente.

Bibliografia

[1] *ANTLR*. https://www.antlr.org/. Accessed: 2022-01-10.

[2] V. Danos e J. Krivine. «Reversible Communicating Systems». In: *CONCUR. Lecture Notes in Computer Science* 3170 (2004), pp. 292–307.

[3] *GDB: The GNU Project Debugger*. https://www.sourceware.org/gdb/. Accessed: 2021-12-18.

[4] *Graphviz*. https://graphviz.org/. Accessed: 2022-01-10.

[5] I. Lanese, I. Phillips e Ulidowski I. «An Axiomatic Approach to Reversible Computation». In: *Foundations of Software Science and Computation Structures* 12077 (2020), pp. 442–461.

[6] *Libreria Graphviz per Python*. https://github.com/xflr6/graphviz/. Accessed: 2022-01-31.

[7] *Libreria Tkinter*. https://docs.python.org/3/library/tkinter.html. Accessed: 2022-02-06.

[8] *Linguaggio DOT: Specifiche di Graphviz*. https://www.graphviz.org/doc/info/lang.html. Accessed: 2022-01-10.

[9] *Link GitHub di Tallulah*. https://github.com/WilliamArnone/Tallulah. Accessed: 2022-02-23.

[10] Ian Sommerville. *Software Engineering, Tenth Edition*. Pearson Education, 2016.

[11] H. Sutter. «The free lunch is over: A fundamental turn toward concurrency in software». In: *Dr. Dobbs Journal* 30(3) (2005).

[12] T. Britton, L. Jeng, G. Carver and P. Cheak. *Reversible Debugging Software "Quantify the time and cost saved using reversible debuggers"*. http://www.roguewave.com. 2012.

[13] *Windows Time Travel Debugging Site*. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview. Accessed: 2021-12-18.

Elenco delle figure