Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Cauder:
Gestione degli errori tramite link

Relatore:
Chiar.mo Prof.
IVAN LANESE

Presentata da:
LUCA TABANELLI

2020
2016

Indice

1 INTRODUZIONE

Il debugging indica l'attività che consiste nell'individuazione e correzione da parte del programmatore di uno o più errori (bug) rilevati nel software. Essa è una delle attività più costose nell'ambito dello sviluppo software, sia in tempo che in termini economici, tale che, miglioramenti in questo campo, generano un notevole impatto sul piano pratico. Un debugger è uno strumento software specifico utilizzato per eseguire l'attività di debugging, consentendo l'esecuzione del programma a piccoli passi fino a punti scelti dal programmatore (settando un breakpoint per esempio), mostrando nel contempo lo stato del programma: quali variabili sono in gioco, lo stack di esecuzione ecc..

Secondo una ricerca recente [1] (2020) di Undo, un'azienda che si occupa di debugging reversibile, in collaborazione con un progetto MBA della Cambridge Judge Business School, emerge che 620 milioni di ore di sviluppo all'anno, in tutto il mondo, vengano impiegate per il debug di errori software, a un costo di circa 61 miliardi di dollari. Sempre secondo il rapporto emesso, il 41% degli intervistati ha affermato di aver identificato la riproduzione di un bug come la più grande barriera per trovare e correggere i bug più velocemente, seguita dalla correzione effettiva del bug (23%). Ben più della metà (56%) ha affermato che potrebbe rilasciare software uno o due giorni più velocemente se la riproduzione degli errori non fosse un problema. Un secondo studio [2], meno recente(2013), eseguito sempre da MBA Cambridge, evidenzia come l'utilizzo della tecnologia di *debug reversibile* permetta un aumento dell'efficienza sul tempo di debugging del 26%.

Il paradigma di computazione reversibile prevede che i programmi possano essere eseguiti non solo in avanti, bensì anche all'indietro. Ciò permette di recuperare gli stati passati del programma, capacità utile soprattutto nel caso in cui si verifichino errori, in modo tale da procedere successivamente in uno stato di computazione privo di errori. Nel debugging non reversibile, si sfrutta la combinazione di punti di interruzione (BreakPoint), punti di controllo (ControlPoint) e altri elementi generici di debugging, che fungono da indicatori per aiutare a trovare il bug nel software. Con questi debugger risulta così possibile eseguire il programma finché non raggiunge un punto di interruzione e analizzare eventuali problemi che si verificano in quella particolare sezione di codice.In alcuni casi, questo approccio funziona perfettamente, ma il più delle volte, non si sa in quale sezione del codice è avvenuto l'errore e se si sa, non si sanno le conseguenze all'interno di questa sezione che generano il bug. La reversibilità, applicata al debugging, permette la simulazione dell'esecuzione del codice in entrambe le direzioni, *dando la possibilità di soffermarsi in un intorno specifico del codice, al fine di isolare al meglio l'errore, le cause e le conseguenze di questo ultimo*, cosa che migliora l'attività di debugging, come si evince dallo studio [2].

In questa tesi, il debugging reversibile sarà applicato al linguaggio *Erlang*[3]. Erlang è un linguaggio di programmazione ad attori funzionale, concorrente e distribuito a typing dinamico, basato sulla macchina virtuale BEAM. Fu progettato per gestire applicazione real-time, 1)fault-tolerant, 2)distribuite e 3)non-stop. Queste feature, in un'applicazione scritta in Erlang, vengono raggiunte rispettivamente grazie a:

1. Link tra i processi, suddividendoli in processi worker e processi supervisor, in cui un supervisor linkato ad un worker, cattura e gestisce *la fine dell'esecuzione del worker*, sia essa la terminazione normale del codice o un'uscita anomala

2. API per lo spawn di nodi e processi su nodi, su altre macchine in rete

3. Loading del codice a caldo

L'unità di computazione base del linguaggio Erlang è il processo, come per Java o Python risulta essere la classe (più precisamente un oggetto che istanzia una classe). Questi processi vengono schedulati da BEAM, come un normale sistema operativo, tramite uno scheduler di tipo *preemptive* con politica basata sulle *riduzioni*, ovvero ogni operazione fatta da un processo costa un *certo ammontare di riduzioni*, e raggiunta la soglia limite, viene schedulato

un altro processo[1]. I vari processi attivi in un sistema run-time Erlang comunicano attraverso, e solamente così, il *message-passing asincrono*, dato che essi sono totalmente isolati dagli altri processi in esecuzione(ecco perchè si chiamano processi e non thread). Ogni processo possiede una local mailbox che è implementata da una coda in cui i messaggi inviati da altri processi vengono immagazzinati, in attesa di essere elaborati. ***Il motivo per cui è stato scelto Erlang come linguaggio di riferimento risiede nel fatto che la parte sequenziale e la parte concorrente del linguaggio sono ben distinguibili fra di loro***.

Attualmente, il debug reversibile per linguaggi sequenziali è ben compreso e ha trovato applicazioni in ambito industriale. Tuttavia, oggigiorno la maggior parte del software, soprattutto in ambito web, si sta spostando verso il paradigma concorrente, dato l'avvento di cpu multi-core in primis. Sfortunatamente, la nozione di reversibilità in un contesto concorrente è intrinsecamente diversa dalla nozione di reversibilità in un contesto sequenziale. In un contesto sequenziale, l'ultimo stato di un programma è individuabile nello *stato del programma alla penultima riga di codice*, mentre in ambito concorrente, non è mai ben definita l'ultima riga di codice, come si nota nell'esempio 1, dato che l'esecuzione di un codice concorrente, ripetuto più volte con gli stessi input, può variare di volta in volta, quando nell ambito sequenziale è sempre univoco il flusso di codice per gli stessi input. Nello specifico di Erlang, l'esecuzione temporale del codice può sovrapporsi tra i vari processi, dato che la macchina virtuale BEAM[4], *per ogni core della cpu*, crea un *thread scheduler*, che gestisce una coda di processi Erlang, ottendo la *concorrenza reale* della computazione. Sono possibili due soluzioni:

- Sequenzializzazione dell'esecuzione del codice.
- Si trova una nozione differente di reversibilità per la programmazione concorrente.

[1]il costo di un operazione in riduzione dipende da una stima che viene fatta da BEAM, e può variare da esecuzione ad esecuzione

a)Codice sequenziale

```
main()->
  A=3,
  case A of
        3->
      io:fwrite("A"),
      io:fwrite("B");
    _->
      io:fwrite("C")
  end.
```

Il flusso del codice è sempre univocamente definito ed uguale ad ogni riesecuzione per l'input 3.

b)Codice concorrente

```
main()->
  A=3,
  case A of
        3->
      spawn(io,fwrite,["A"]),
      io:fwrite("B");
    _->
      io:fwrite("C")
  end.
```

Il flusso del codice è univocamente definito solo fino al *case* mentre, dopo la *spawn*, può essere eseguita prima la stampa di "A" poi di "B", *o viceversa* in base ad un certo *scheduling*, *cambiando potenzialmente ad ogni riesecuzione.*

Figura 1: Esempio codice sequenziale e concorrente

Si consideri inoltre che, con l'approccio di programmazione concorrente, si aggiungono, nel programma, tipi di bug intrinsechi ad esso, quali *deadlock*, *race conditions* o *starvation*. Seppur i linguaggi di programmazione basati sul paradigma concorrente evitino questi tipi di bug a livello di progettazione, **tramite politiche ben definite e scelte durante la progettazione del linguaggio**[2], si può comunque incorrere in altri tipi di bug, relativi alla concorrenza, durante lo sviluppo software con questi linguaggi, uno in particolare relativo ad Erlang, quale la *violazione dell'ordine di invio dei messaggi*.[3] **Nello specifico per Erlang, la sequenzializzazione fallisce in principio in cpu multi-core, ma in generale essa risulta essere totalmente inefficiente per il paradigma concorrente in quanto, la sequenzializzazione, oltre ad avere un grosso overhead, tralascia le *informazioni relative alla causalità*. Quest'ultime risultano essere fondamentali e centrali in relazione ai bug sopra citati, e in generale ai bug nella concorrenza, in quanto questi bug posso apparire in**

[2]Nello specifico di Erlang, se si verifica un deadlock, BEAM uccide i processi coinvolti in esso, per poi farli ripartire uno alla volta, ed effettuando dei controlli su di essi.

[3]Ipotizziamo di avere due processi X e Y rispettivamente sender e receiver.X invia z al tempo T=1 ed al tempo T=2 invia q.Non è detto che arrivi prima z di q, nonostante fosse stato inviato prima.

una esecuzione ma non apparire in un esecuzione successiva dello stesso codice, dipendendo dalla velocità di esecuzione dei singoli processi e da come questi interagiscano tra di loro. Nella figura di esempio qui sotto, un sender (*sender*) invia due numeri (2,0) a due diversi intermediari (*inter*), che a loro volta invieranno i loro numeri ad un worker (*worker*), che effettuerà una divisione. In base alla sequenza di arrivo di questi due numeri, il worker può restituire 0 o andare in errore, e questa sequenza di arrivo non è mai univoca, anzi varia in base *allo scheduling*.

funcs.erl

```
worker()->
 receive
 X->
  receive
   Y->
   io:fwrite("~p~n",[X/Y])
   end
 end.
inter(W)->
 receive
 X->
  W ! X
 end.
sender(I1,I2)->
 I1 ! 0,
 I2 ! 2.
```

test.erl

```
main()->
 W=spawn(funcs,worker,[]),
 I1=spawn(funzioni,inter,[W]),
 I2=spawn(funcs,inter,[W]),
 spawn(funcs,sender,[I1,I2]).
```

Figura 2: Esempio possibile bug

Tutto ciò porta alla ricerca di una nozione di reversibilità alternativa per la programmazione concorrente. Anticipo che questa nozione di reversibilità sarà la reversibilità *causal-consistent* [5], ovvero una nozione che non si basa sul concetto di tempo, bensì sul concetto di *causalità* tra gli eventi. Questo concetto specifica quali azione possono essere annullate. Questa nozione verrà approfondita nel capitolo relativo al background.

Al momento l'unico debugger reversibile, che sfrutta la nozione sopracitata di reversibilità, per Erlang, è Cauder[6]. Allo stato dell'arte, Cauder supporta solo alcune primitive del linguaggio Erlang, quali, in relazione alla parte concorrente, la *spawn*, per la creazione di nuovi processi, e la *send*, per

l'invio di messaggi, mentre per la parte sequenziale, la *self*, che ritorna il pid del processo che la chiama, e la *sleep*, che sospende il processo chiamante. **Molto importante, Cauder gestisce un solo thread scheduler**. In realtà Cauder non lavora direttamente sul codice sorgente puro di Erlang, bensì su un linguaggio di compilazione intermedio tra Erlang e BEAM, il Core Erlang[7]. La scelta invece di utilizzare Core Erlang invece che Erlang puro, risiede nel fatto che, esistono, nella libreria standard Otp di BEAM, packages che permettono la compilazione di moduli Erlang in questo linguaggio e la manipolazione diretta del codice risultante dopo questa compilazione, tramite strutture dati(tuple/mappe), risultando nel contempo essere un linguaggio abbastanza simile ad Erlang puro.

Cauder permette di eseguire un programma, dopo averlo caricato e compilato dal menù a tendina File, in 3 modalità:

- Manuale: si sceglie un Pid/MessageId e si manda avanti o indietro di un passo il processo/messaggio.
- Automatica: si sceglie un numero N di passi e il sistema avanza o torna indietro, se possibile, di N passi, *secondo uno scheduling ben definito.*
- Rollback: una particolare funzionalità che permette di fare *l'undo di una specifica azione compiuta da un processo o un messaggio, incluse tutte le sue conseguenze.*

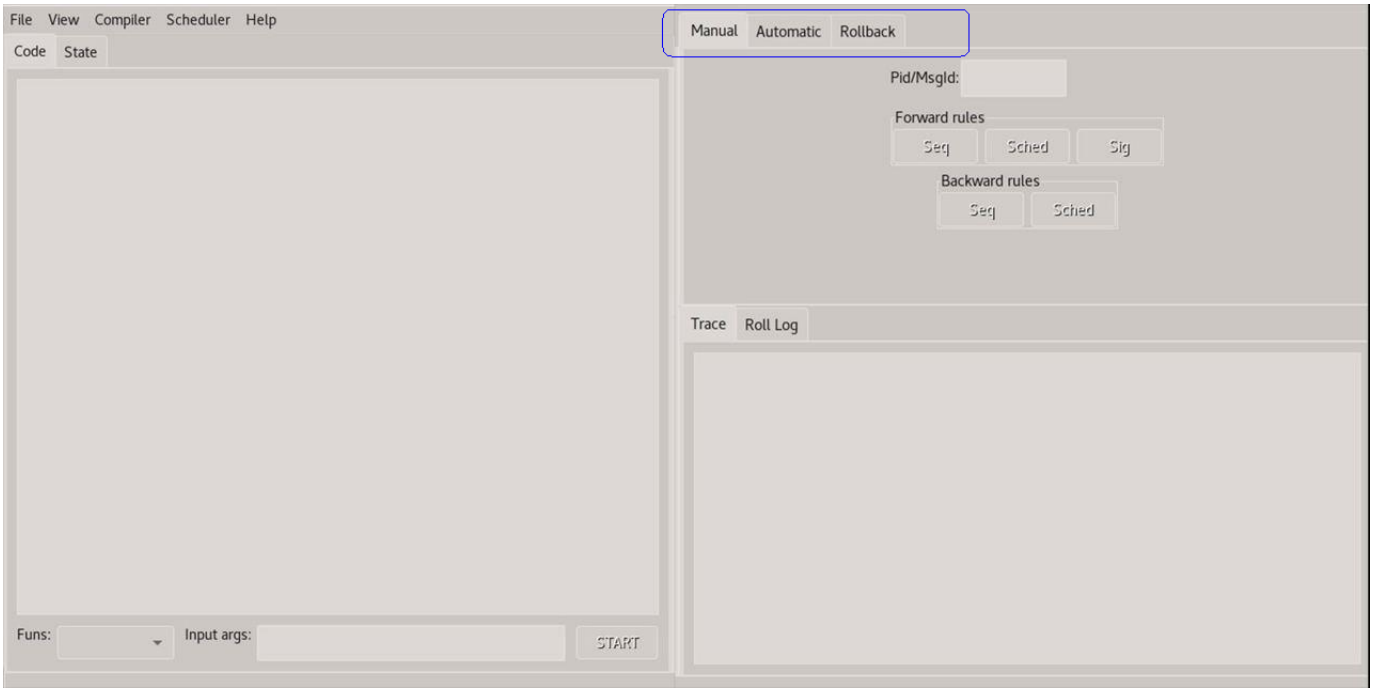

Figura 3: La finestra iniziale di Cauder

Questa tesi verte su *un'estensione* di Cauder, tramite l'implementazione dei *link e la gestione della terminazione del codice*, per la gestione degli errori.
Ciò che introdurrò in primis sarà la nozione di reversibilità in generale, una nozione di reversibilità che si adatta alla nozione di concorrenza e infine l'applicazione di essa al debugging.
Successivamente illustrerò il linguaggio di Cauder, le strutture dati che manipola e le regole informali che implementa e come esse fanno evolvere le strutture dati gestite da Cauder.
Infine passerò ad illustrare l'implementazione dell'estensione del linguaggio supportato, ovvero come esso viene esteso, le strutture dati estese di Cauder e come evolvono, secondo le nuove regole informali derivanti dalla gestione degli errori e i link a run time.

2 BACKGROUND

Come già anticipato nell'introduzione, il calcolo reversibile permette di eseguire computazioni sia in avanti che all'indietro, permettendo il recupero degli stati precedenti. Esso trova origini in fisica, più precisamente nel *Principio di Landauer* [8], affermando che l'eliminazione di bit di informazione

produce una quantità di calore che non può essere diminuita oltre un determinato limite. Il principio di Landauer descrive il *limite di Landauer* W, che fissa il minimo ammontare di energia che serve per cambiare un bit di informazione ed è formulato come segue:

$$W = kTln2$$

dove

- k è la costante di Boltzmann (1,380649x10-23 J K^{-1})
- T è la temperatura assoluta del circuito in kelvin
- ln 2 è il logaritmo naturale di 2

Nella realtà quotidiana la quantità di energia rilasciata da una macchina per un qualsiasi processo di elaborazione dati è decisamente maggiore di molti ordini di grandezza rispetto al limite di Landauer. La reversibilità fisica permetterebbe quindi di ottenere risultati senza rilascio di calore, ma come affermato sempre da Landauer [9], *per poter essere reversibile fisicamente un calcolatore deve essere anche logicamente reversibile*. Da allora il calcolo reversibile ha suscitato interesse in svariati campi, quali il design di hardware convenzionale e quantistico, biologia computazionale, sviluppo software ecc..[10]. Questa tesi si focalizzerà nell'applicazione del calcolo reversibile al debugging. Articolerò il background in 2 step:

1. Quali tecniche servono per applicare la reversibilità al debugging
2. Implementare queste tecniche su uno strumento software reale

2.1 Reversibilità ed applicazione al debugging

In questa sezione fisserò il **tragitto** che parte dalla nozione di reversibilità fino all'applicazione di essa al debugging, in cui faccio riferimento a [11]. Questo tragitto si articola nei passi seguenti, che verranno enfatizzati nelle sezioni successive:

1. Quale nozione di reversibilità serve nei sistemi concorrenti
2. Quali informazione devo memorizzare
3. Come controllare i meccanismi reversibili di base
4. Come sfruttare questi meccanismi per il debugging reversibile

1)Quale nozione di reversibilità serve nei sistemi concorrenti:
Già nell'introduzione ho accennato che la nozione che userò sarà la nozione di *causal-consistent reversibility*. Questa nozione, in termini informali, può essere formulata come: *"annulla ricorsivamente qualsiasi azione a patto che tutte le sue conseguenze siano state annullate in precedenza"*. Più formalmente: Sia *un'azione=A,B,...* per denotare l'effettiva azione o A^{-1}... per denotare l'undo di un'azione.
Sia σ una sequenza di azioni.
Denoto con A_n l'ennesima occorrenza di A in σ.

Lemma causal-consistent trace. *σ è causal-consistent $\iff \forall\ A_n$ e B_m, che sia compresa tra A_n e A_n^{-1}, o B_m non è conseguenza di A_n oppure B_m^{-1} è prima di A_n^{-1}.*

Ecco come la *causalità*, espressa nell'introduzione, prende forma in questa nozione. La consistenza-causale poggia le sue basi sul *Loop Lemma*, che si formalizza nei due assiomi a seguire. Informalmente questo lemma afferma che l'esecuzione di un'azione e poi l'immediato annullamento di essa, dovrebbe ricondurre allo stato di partenza. Formalmente:

Assioma Loop Lemma 1. *$S \xrightarrow{A} S' \xrightarrow{A^{-1}} S$: Se da uno stato S eseguo un'azione A per poi annullarna nel passo successivo, allora devo ritornare allo stato S.*

Assioma Loop Lemma 1. *$S \xrightarrow{A^{-1}} S' \xrightarrow{A} S$: Se da uno stato S eseguo l'undo di A per poi rifare A, allora devo ritornare allo stato S.*

Questo lemma risulta essere importante, in quanto ci assicura di effettuare passi all'indietro in **modo univocamente definito**. Se così non fosse, ad ogni passo all'indietro, accederei ad un'ulteriore sotto-albero di computazione, cosa che si rivelerebbe totalmente inutile per il debugging.
2)Quale informazione devo memorizzare:
Per poter annullare un'azione, devo evitare la perdita di informazioni. Se così non fosse avrei una combinazione elevata di possibili stati predecessori, il chè renderebbe impossibile risalire allo stato precedente. Va quindi mantenuta un'informazione sulla storia della computazione che chiamerò *history*, che sia compatibile con la nozione causal-consistent. **Detto ciò bisogna tenere quindi conto del vincolo posto dal Loop Lemma per costruire la *history* giusta ma significativa allo stesso tempo**. Per esempio tener traccia di quante volte è stata fatta l'undo di un'azione viola chiaramente il Loop Lemma, in quanto un counter globale verrebbe solo incrementato. Fortunatamente grazie al *causal-consistency theorem*[5], si riesce a caratterizzare la *history* da conservare. Il teorema afferma, informalmente, che 2

computazione che iniziano dallo stesso stato S, finiscono nello stesso stato S' $\iff$ sono equivalenti.
Questo è un risultato importante in quanto la *history* fa parte dello stato, ciò vuole dire che finire nello stesso stato significhi avere la stessa *history*. Sebbene questo sia un risultato molto generico e non dipendente da un linguaggio specifico, esso ci da una base astratta per poter applicare su un linguaggio specifico la nozione di consistenza causale.
3)Come controllare i meccanismi reversibili di base:
Nel punto 1 abbiamo fissato il concetto di causal-consistency reversibility dicendo quando una sequenza di azione è causal-consistent, mentre nel punto 2 abbiamo fissato cosa ci serve per poter effettuare passi all'indietro. In questo punto introduco il meccanismo di *roll*. Questo meccanismo permette di fare l'undo di un'azione A, mantenendo la consistenza-causale e algoritmicamente formalizza la nozione di causal-consistency reversibility:

Algorithm 1 Roll(A)

```
for all B ∈ {conseguenze dirette di A} do
  Roll(B)
end for
A^{-1}
```

Poiché le azioni dipendenti devono essere annullate in anticipo, l'annullamento di un'azione arbitraria deve comportare l'annullamento di tutto l'albero delle sue conseguenze causali.
4)Come sfruttare la consistenza causale per il debugging reversibile:
Dato un comportamento scorretto che emerge dal programma, per esempio un output sbagliato sullo schermo, si deve trovare la riga di codice contenente il bug che ha causato tale comportamento. In particolare, la riga di codice che esegue l'output potrebbe essere corretta, semplicemente riceve valori errati da elaborazioni passate. Essendo in ambito concorrente, la catena di causalità risulta essere più complessa rispetto all'ambito sequenziale dato che la riga di codice che esegue l'output e quella contenente il bug possono trovarsi in processi diversi. Ciò può risultare ostico per trovare la fonte del bug.
Strutturalmente, una catena causale in ambito concorrente può essere modellata tramite un grafo, mentre in ambito sequenziale risulta essere una lista. Detto ciò, la seguente tecnica cattura l'idea di debugging reversibile causal-consistent: Dato un comportamente scorretto, possiamo cercare di capire se è sbagliato di per sè (esempio: cercare di fare un secondo binding di una

variabile, nel caso di Erlang) oppure cercarne le cause, per poi iternarne la procedura. Le cause di un comportamento scorretto dipendono dall'errore che emerge (per esempio: un valore errato di una variabile dipende dall'ultimo assegnamento della variabile stessa). Questa tecnica viene supportata dai meccanismi relativi alla *history* e *roll*. La *history* ci definisce lo stato precedente ed inoltre ci permette di visualizzare l'azione che è la causa dell'errore, mentre *roll* ci permette effettivamente di effettuare l'undo dell'azione che causa l'errore.

2.2 Cauder

In questa sottosezione introduco Cauder ovvero:

1. La sintassi che supporta, ponendo enfasi sulle funzioni rilevanti per questa tesi.
2. Le strutture dati che manipola.
3. La semantica indotta dalla sua sintassi e come modifica le strutture dati.

Per tutta questa sottosezione faccio riferimento a [12].
1) La sintassi supportata:
La sintassi del linguaggio può essere trovata nella Figura 2 4. Un modulo è una sequenza di definizioni di funzioni, dove ogni nome di funzione f / n (atom / arity) ha una definizione associata della forma $fun(X_1, ..., X_n) \xrightarrow{body} e$. Il corpo di una funzione è un'espressione, che può includere variabili, letterali, nomi di funzioni, liste, tuple, chiamate a funzioni predefinite, principalmente operatori aritmetici e relazionali, applicazioni di funzioni, espressioni case, associazioni let e espressioni di ricezione; inoltre, consideriamo anche le funzioni *spawn*, *"!"* (per inviare un messaggio) e *self()* (ritorna il pid del processo chiamante) che in realtà sono incorporati nel linguaggio Erlang.

$$
\begin{array}{rcl}
module & ::= & \mathsf{module}\ Atom = fun_1, \ldots, fun_n \\
fun & ::= & fname = \mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow expr \\
fname & ::= & Atom/Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid [\,] \\
expr & ::= & Var \mid lit \mid fname \mid [expr_1|expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \mathsf{call}\ expr\ (expr_1, \ldots, expr_n) \mid \mathsf{apply}\ expr\ (expr_1, \ldots, expr_n) \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clause_1; \ldots; clause_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ Var = expr_1\ \mathsf{in}\ expr_2 \mid \mathsf{receive}\ clause_1; \ldots; clause_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(expr, [expr_1, \ldots, expr_n]) \mid expr_1\,!\,expr_2 \mid \mathsf{self}() \\
clause & ::= & pat\ \mathsf{when}\ expr_1 \rightarrow expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \ldots, pat_n\}
\end{array}
$$

Figura 4: Regole sintattiche del linguaggio

Di seguito spiego con maggior dettaglio solo alcune *expressioni*. Il motivo di ciò risiede nel fatto che l'estensione di Cauder risulta mostrare analogie, più o meno rilevanti, con queste regole sintattiche e le loro regole semantiche.

- $\mathsf{spawn}(expr, [expr_1, ..., expr_n]) \xrightarrow{return} SpawnedPid$: La chiamata alla funzione spawn crea un nuovo processo, che inizia con la valutazione di $\mathsf{apply}(expr, [expr_1, ..., expr_n])$. Ritorna il *pid* del processo appena generato.

- $expr_1$! $expr_2 \xrightarrow{return} expr_2$: La chiamata alla funzione send (!) invia il messaggio *expr2* al processo con *pid* $expr_1$. Ritorna il messaggio inviato ovvero $expr_2$.

- receive $clause_1; ...; clause_n \xrightarrow{return} expr_2$ di clause matching: L'espressione receive attraversa i messaggi nella coda dei messaggi del processo finché uno di essi non corrisponde a un ramo nell'istruzione di ricezione; dovrebbe trovare il primo messaggio v nella coda tale che $\exists$ clause $\in \{clause_1;...;clause_n\}$ tale che v matcha *pat* di clause $\wedge$ $expr_1$ di clause==true; ritorna $expr_2$ di clause, con l'ulteriore effetto collaterale di eliminare il messaggio v dalla coda dei messaggi del processo. Se non sono presenti messaggi corrispondenti nella coda, il processo sospende la sua esecuzione finché non arriva un messaggio corrispondente.

Nel punto relativo alla semantica ometto la spiegazione della receive, in quanto è interessante solo dal punto di vista comportamentale.

2) Le strutture dati di Cauder:

In questa sezione vado a definire le strutture dati gestite da Cauder. Nella

figura sottostante mostro come lo stato di Cauder viene rappresentato graficamente. Ad ogni struttura dati che introduco, associerò come essa viene rappresentata internamente in Cauder e come venga rappresentata graficamente, facendo riferimento alla figura sottostante.

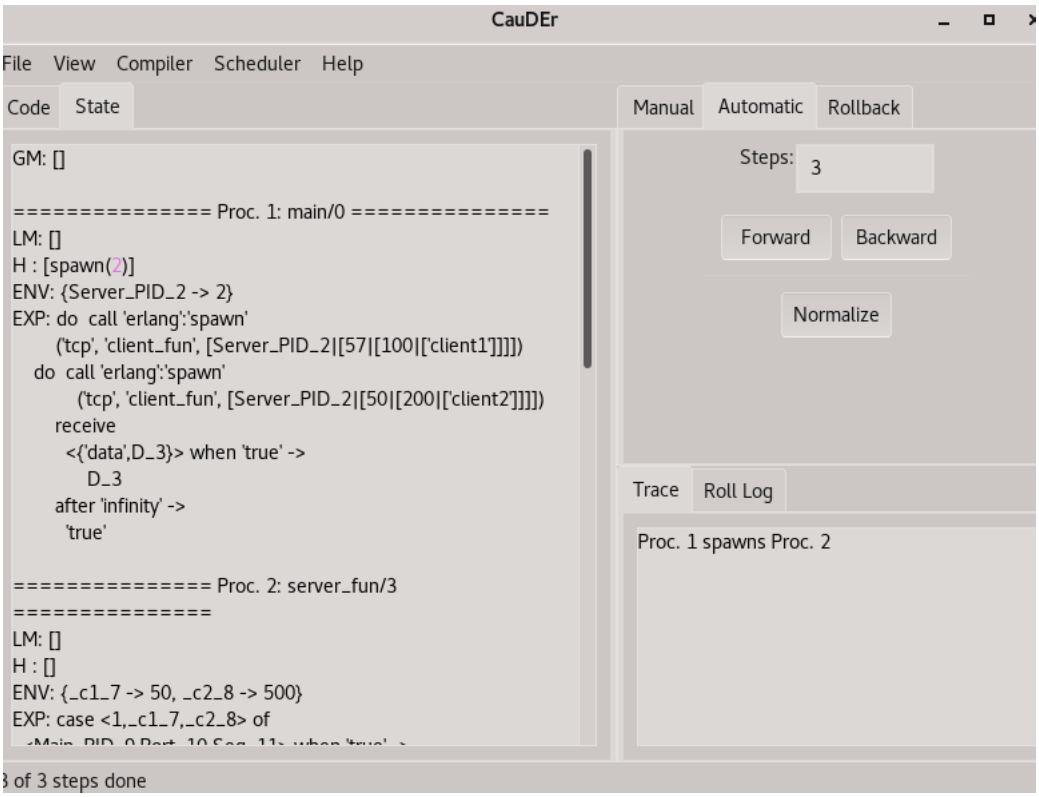

Figura 5: Lo stato di Cauder.

Un *sistema* in esecuzione viene denotato tramite Γ;Π, dove Γ denota la mailbox global ovvero l'insieme dei *messaggi inviati* in attesa di essere consegnati, mentre Π denota l'insieme dei *processi* nel sistema. Il *sistema* viene implementato in Cauder tramite un *record sys* contenente i seguenti campi:

- ***sched***: denota la tipologia di scheduling delle azioni da utilizzare. Cauder utilizza due politiche di scheduling:
  - RANDOM: L'azione viene scelta casualmente *tra tutte le azioni possibili* del sistema.
  - PRIO_RANDOM: L'azione viene scelta casualmente *tra tutte le azioni effettuabili dai processi* e se non ce ne sono viene scelta

casualmente *tra tutte le azioni effettuabili per la schedulazione dei messaggi.*

Nel punto della semantica verrà spiegato il perchè è necessaria una schedulazione dei messaggi.

- ***msgs***: denota la mailbox globale ovvero Γ. Nella figura 5 viene rappresentata da *GM.*
- ***procs***: denota l'insieme dei processi ovvero Π. Nella figura 5 viene rappresentata dalla lista *======Proc.N:fun/M====.*
- ***trace***: denota le azioni effettuate nel sistema. Nella figura 5 vengono visualizzate nel *tab Trace.*
- ***roll***: denota le operazioni di *roll* effettuate nel sistema. Nella figura 5 vengono visualizzare nel *tab Roll log.*

Illustrato la struttura di un *sistema* mi soffermo sui campi *msgs* e *procs*, in quanto, **per motivi diversi e che verrano spiegato nella sezione relativa all'estensione**, risultano essere rilevanti per questa tesi. Il campo *sched* verrà menzionato nella parte dell'estensione, ma non approfondito in quanto non troppo rilevante per questa tesi. *Msgs* risulta essere implementata tramite una *lista* di *msg.*
Formalmente *msg* è una tripla del tipo *(pid_dest,value,time).* Il campo *time* risulta essere necessario in quanto discrimina quel messagio specifico, dato che potre avere più messaggi con lo stesso valore e destinatario. Banalmente, un *msg* viene implementato tramite un *record msg* contenente i campi sopra citati. Analogalmente a *msgs*, *procs* viene implementato tramite una *lista* di *proc.* Formalmente *proc* viene denotato tramite $\langle\ p,\theta,e,h,lm\rangle$, dove:

- p: rappresenta il pid del processo.
- θ: rappresenta l'ambiente del processo, ovvero *l'insieme* dei bindings.
- e: rappresenta l'espressione da valutare.
- h: rappresenta *l'history* del processo, ovvero la *sequenza* di tutte le azioni che il processo ha effettuato.
- lm: rappresenta la mailbox del processo, ovvero la *sequenza* di tutti i messaggi *schedulati* il cui destinatario è il processo.

Proc viene implementata tramite un *record proc* con i seguenti campi e farò riferimento alla figura 5 per le visualizzazioni:

- *pid*: implementa p ed è visualizzato tramite N in =====*Proc.N:fun/M*====.
- *hist*: implementa h ed è visualizzato tramite H.
- *env*: implementa θ ed è visualizzato tramite ENV.
- *exp*: implementa e ed è visualizzato tramite EXP.
- *mail*: implementa lm ed è visualizzato tramite LM.

Il *record proc* possiede campi aggiuntivi ma che non menziono dato la non rilevanza per questa tesi.

2) La semantica di Cauder:

In questo punto introduco la semantica derivante dalla sintassi di Cauder. Più precisamente mostro solo un sottoinsieme di queste regole, in quanto le altre non sono rilevanti per questa tesi. Evidenzio **4** regole, che vado ad illustrare, lavorando su un processo generico P *che esegue la regola descritta*, ovvero $\langle p, \theta, e, h, lm\}$ e su un sistema generico S, ovvero $\Gamma;\Pi$.

Essendo un linguaggio funzionale ad ogni passo creo delle nuove strutture dati sulla base delle vecchie, costruendo nuovi stati a partire dai precedenti. Come già accennato in precedenza, sia gli insiemi che le sequenze vengono implementati come *liste*. A livello implementativo, per l'aggiunta di un elemento in una lista, uso la notazione [H|L], ovvero creo una nuova lista aggiungendo H in testa alla lista L. Nella spiegazione della semantica, per la modifica degli insiemi utizzo la notazione insiemistica mentre per le sequenze utilizzo direttamente la notazione implementativa. Ciò viene fatto per tenere evidenziato cosa è un insieme e cosa è una sequenza. Denoto le costanti tramite valore_atomo, mentre per riferirmi ai campi delle strutture dati, utilizzo la notazione punto (esempio: flag di un processo P = P.f).

Se non esplicitamente utilizzata, allora mi riferisco ai campi di un processo. Nel punto 1 ho posto enfasi solo su 3 regole sintattiche, ovvero la spawn per generare processi, la send per inviare un messaggio e la receive per elaborare un messaggio presente nella local mailbox del processo. **Ma non è definito come un messaggio venga recapitato dalla global mailbox alla local mailbox, dato l'asincronicità illustrata all'inizio di questa sezione**. La quarta regola semantica (*rule sched*) serve per simulare questa asincronicità.

SEMANTICA IN AVANTI:

- *rule spawn*: Informalmente questa regola va ad inserire in Π un *proc* vuoto. Formalmente:
  Sia P'=$\langle p', \theta' =, e' = \mathsf{apply}(fun(expr_1...expr_n)), h' = [], lm = []\rangle$ il processo spawnato.

Sia h"=[{spawn, θ, e, p'} | h].
Sia P"=$\langle p, \theta'', e'', h'', lm\rangle$.
Il sistema risultante da questo passaggio sarà S'=$\Gamma; \Pi \setminus \{P\} \cup \{P', P''\}$.

- ***rule send***: Informalmente questa regola va ad inserire in Γ un *msg*. Formalmente:
  Sia msg=*(dest_pid,payload,time)*.
  Sia h'=[{send, θ, e, msg} | h].
  Sia P'=$\langle p, \theta', e'', h', lm\rangle$. Il sistema risultante da questo passaggio sarà S'=$\Gamma \cup \{msg\}; \Pi \setminus \{P\} \cup \{P'\}$.

- ***rule sched***: Sia msg=*(dest_pid,payload,time)* il messaggio selezionato per la schedulazione. Sia lm'=[msg | lm]. Sia P'=$\langle p, \theta, e, h, lm'\rangle$. Il sistema risultante da questo passaggio sarà S'=$\Gamma \setminus \{msg\}; \Pi \setminus \{P\} \cup \{P'\}$.

SEMANTICA ALL'INDIETRO:

- ***rule spawn***: Informalmente questa regole va ad eliminare da Π un *proc* spawnato. Come precondizione per effettuare l'undo della spawn, bisogna che il processo spawnato sia vuoto. Più formalmente bisogna che P'=$\langle p', \theta', e', h' = [], lm = []\rangle$. Se non fosse rispettata questa precondizione si rischierebbe di violare il Loop Lemma. Ipotizziamo di avere 3 processi cosi definiti, in cui il processo 1 spawna il secondo e il secondo spawna il terzo. Abbiamo una situazione del genere:

  1. $\langle p', \theta', e', h', lm'\rangle$ con h'=[{spawn, θ, e, p''} | h].
  2. $\langle p'', \theta'', e'', h'', lm''\rangle$ con h"=[{spawn, θ, e, p'''} | h].
  3. il terzo processo non è rilevante.

  Ora facciamo l'undo della spawn del secondo processo per poi rifarla. Ci ritroviamo nella seguente situazione:

  1. $\langle p', \theta', e', h', lm'\rangle$ con h'=[{spawn, θ, e, p''} | h].
  2. $\langle p'', \theta'', e'', h'', lm''\rangle$ con h"=[] ed lm"=[].
  3. il terzo processo non è rilevante.

  Abbiamo fatto l'undo di un'azione per poi rifarla e siamo finiti su uno stato diverso, violando il Loop Lemma.
  Posta vi sia questa condizione, l'undo di una spawn si formalizza così:
  Sia h=[{spawn, θ, e, p'} | h'].
  Sia P"=$\langle p, \theta'' = \theta, e'' = e, h', lm\rangle$.
  Allora il sistema evolve in S'=$\Gamma; \Pi \setminus \{P', P\} \cup \{P''\}$.

- ***rule send***: Informalmente questa regola elimina un *msg* da Γ. Come precondizione per effettuare l'undo di una send, bisogna che il messaggio *msg* sia presente in Γ. Se così non fosse si rischierebbe di violare il Loop Lemma. Ipotizziamo questo scenario: Un messaggio *msg=(p',val,time)* è presente nella local mailbox di p' (è già stato schedulato). Quindi ho questi due processi:

  - P'=$\langle p', \theta', e', h', lm' \rangle$ con lm'=[msg | lm] che è il destinatario del messaggio.
  - $\Gamma' = \Gamma \setminus \{msg\}$.

  Se facessi l'undo della send per poi rifarla subito dopo mi ritroverei in questa situazione:

  - P'=$\langle p', \theta', e', h', lm' \rangle$ con lm'=[msg | lm] che è il destinatario del messaggio.
  - $\Gamma' = \Gamma \cup \{msg\}$.

  In sostanza mi troverei con un messaggio duplicato, che chiaramente viola il Loop Lemma.
  Posto vi sia questa condizione, l'undo di una send si formalizza in questo modo: Sia *msg=(p',val,time)*. Sia h'=[$\{\mathsf{send}, \theta, e, msg\}$ | h]. Sia P'=$\langle p, \theta' = \theta, e' = e, h, lm \rangle$. Allora il sistema evolve in S'=$\Gamma \setminus \{msg\}; \Pi \setminus \{P\} \cup \{P'\}$.

- ***rule sched***: Informalmente questa regola elimina un *msg* dalla local mailbox del processo destinatario, per reinserirla in Γ. Come precondizione per effettuare l'undo di una sched, bisogna che il *msg* sia in testa alla local mailbox del processo ricevente. Se così non fosse si rischierebbe di violare il Loop Lemma. Ipotizziamo di voler invertire la *sched* di *msg* e di essere in questa situazione:
  Sia *lm'=[msg',msg | lm]* la local mailbox del processo destinatario di *msg*.
  Se ora facessi l'undo della sched di *msg* per poi rifare subito dopo la sched di *msg* mi ritroverei in quest'altra situazione:
  lm'=[msg,msg' | lm]. Ho fatto l'undo di un'azione per poi rifarla e mi sono ritrovato in uno stato diverso, violando chiaramente il Loop Lemma. Posto vi sia questa condizione, l'undo della *sched* si formalizza in questo modo:
  Sia *lm=[msg | lm']* la local mailbox del processo destinatario di *msg*.
  Sia P'=$\langle p, \theta, e, h, lm' \rangle$. Allora il sistema evolve in S'=$\Gamma \cup \{msg\}; \Pi \setminus \{P\} \cup \{P'\}$.

ROLL:
Informalmente, l'operatore *roll* di un'azione si va a ridurre ad un passo della semantica all'indietro della relativa azione, **andando a creare in primis le condizioni per cui poter effettuare quel passo** (annullando tutte le sue conseguenze), per poi effettuare quel passo. Indico con *back_step(S,p)* una funzione che dato uno stato S e un pid p, osserva l'elemento in cima alla history h del processo con pid p ed effettua un passo all'indietro su di esso, ritornando il nuovo stato S'. Indico invece con *back_rule_xxx* l'undo della *rule_xxx*.

- ***spawn***: A livello informale la *roll spawn* va a "svuotare" un processo (annullando tutte le sue conseguenze ricorsivamente), per poi infine fare l'undo della spawn di esso. Algoritmicamente: Sia $h = [\{\mathsf{spawn}, \theta, e, p'\} \mid h']$.
  Sia P' il processo con pid *p'*.

Algorithm 2 roll_spawn(State,p,p')

```
if P'.lm==[] ∧ P'.h==[] then
  return back_rule_spawn(State,p)
else
  State'=back_step(State,p')
  roll_spawn(State',p,p')
end if
```

- ***rule send***: A livello informale la *roll send* va a riportare un *msg* all'interno di Γ, per poi effettuare l'undo della send. Diversamente dalla *roll spawn*, la *roll send* deve anche valutare le conseguenze derivate dalla schedulazione del messaggio oltre che le conseguenze del processo destinatario. Algoritmicamente: Sia *msg=(p',val,time)*. Sia h=$[\{\mathsf{send}, \theta, e, msg\} \mid h_{old}]$. Sia *lm'* la localmail box di P'. Sia

Algorithm 3 roll_send(State,p,p',time)

```
if msg ∈ Γ then
  return back_step(State,p)
else
  if lm'==[msg | lm_rest] then
    State'=back_rule_sched(State,msg)
    return roll_send(State',p,p',time)
  else
    State'=back_step(State,p')
    return roll_send(State',p,p',time)
  end if
end if
```

3 ESTENSIONE DI CAUDER CON ERROR-HANDLING BASATO SUI LINK

In questa sezione darò una panoramica dell'estensione effettuata su Cauder, partendo da quali espressione vengono aggiunte al linguaggio, quali strutture dati vengono aggiunte e come le nuove espressioni modifichino tali strutture dati.
La comunicazione di Erlang viene eseguita tramite *segnalazione asincrona*. Tutte le diverse entità in esecuzione, come processi e porte, comunicano tramite segnali *asincroni*[13]. Il segnale più comunemente usato è il *messaggio*. Altri segnali comuni sono i segnali di uscita, collegamento, scollegamento, monitoraggio e demonitoraggio.
In questa tesi useremo il termine segnale per riferirci unicamente ai segnali di uscita. In BEAM, un *link* [14] tra due processi può essere visto come un canale bidirezionale in cui vengono trasmessi gli errori che avvengono nei rispettivi processi linkati. Un link tra due processi può essere creato tramite apposite funzioni.
[4] In questa tesi verrà trattata solo la funzione *spawn_link*, che sarà spiegata più in dettaglio nella sezione successiva.
Quando un processo si arresta, sia in modo anomalo (esempio: errore di un pattern match) che in modo normale (il codice termina), i segnali di uscita vengono inviati a *tutti* i processi a cui è attualmente linkato il processo morente. A seconda di come sia impostato il *flag trap_exit* di un processo ricevente, il segnale viene gestito in modo differente. Questo flag viene

[4]Può esistere un solo link tra due processi, quindi qualsiasi tentativo di crearne ulteriori, non produrrà nessun effetto.

impostato tramite la funzione *process_flag*, che verrà anch'essa spiegata in dettaglio nella sezione successiva.

3.1 Estensione della sintassi del linguaggio

La sintassi del linguaggio mostrata in 2.2 viene estesa tramite queste 4 espressioni:

- process_flag(trap_exit, *Boolean*) $\xrightarrow{return}$ *OldBool*: Se *Boolean*=true, i segnali di uscita che arrivano al processo chiamante vengono convertiti in messaggi {*'EXIT', From, Reason*}, che possono essere ricevuti come messaggi ordinari. Se *Boolean*=false, il processo chiamante *termina* se riceve un segnale di uscita diverso da *Reason*=normal, **che a sua volta viene ripropagato a tutti i processi collegati con esso e cosi già**. La chiamata a process_flag ritorna il valore del vecchio flag(*OldBool*).

- spawn_link($expr, [expr_1, ..., expr_n]$) $\xrightarrow{return}$ *SpawnedPid*: funziona esattamente per la spawn, con l'aggiunta che si linka *atomicamente* con *SpawnedPid*.

- exit(*Reason*) $\xrightarrow{no\_return}$: il processo chiamante termina con motivo di uscita Reason. Se *Reason*=normal, viene intesa come *normale* terminazione del codice, altrimenti segnala una terminazione *anomala* con motivo *Reason*.

- error(*Reason*) $\xrightarrow{no\_return}$: il processo chiamante termina in modo *anomalo* la sua esecuzione con motivo *Reason*. A differenza di exit, error segnala **sempre** un errore durante l'esecuzione. Errori a run-time, tipo divisioni per 0 ecc., equivalgono ad una chiamata a error, con la relativa *Reason*.

$$
\begin{array}{rcl}
module & ::= & \mathsf{module}\ Atom = fun_1, \ldots, fun_n \\
fun & ::= & fname = \mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow expr \\
fname & ::= & Atom / Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid [\,] \\
expr & ::= & Var \mid lit \mid fname \mid [expr_1 | expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \mathsf{call}\ expr\ (expr_1, \ldots, expr_n) \mid \mathsf{apply}\ expr\ (expr_1, \ldots, expr_n) \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clause_1; \ldots; clause_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ Var = expr_1\ \mathsf{in}\ expr_2 \mid \mathsf{receive}\ clause_1; \ldots; clause_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(expr, [expr_1, \ldots, expr_n]) \mid \mathsf{spawn\_link}(expr, [expr_1, \ldots, expr_n]) \\
 & \mid & expr_1\ !\ expr_2 \mid \mathsf{self}() \mid \mathsf{process\_flag}(\mathsf{trap\_exit}, Atom) \\
 & \mid & \mathsf{error}(lit) \mid \mathsf{exit}(lit) \\
clause & ::= & pat\ \mathsf{when}\ expr_1 \rightarrow expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1 | pat_2] \mid \{pat_1, \ldots, pat_n\}
\end{array}
$$

Figura 6: Regole sintattiche del linguaggio esteso

3.2 Estensione delle strutture dati del linguaggio

Il sistema S=Γ; Π si estende in S=Γ; Ψ; Π, ove Ψ denota la *global signal box*, che è un *insieme* di *signal*, derivanti dalla terminazione di un *processo*, e in attesa di essere consegnati. Ciò in Cauder si traduce con l'aggiunta al tipo *record sys* di un campo *signals=[]* che corrisponde a Ψ. Lo *scheduler* viene modificato tramite l'estensione dell'opzione PRIO_RANDOM, in cui viene data priorità anche ai *segnali* assieme ai *processi*. *Signal* è una tupla del tipo {*p',p,type,reason,time*} dove:

- *p'* denota il pid del processo a cui recapitare il signale.
- *p* denota il pid del processo che ha generato il segnale.
- *type*=error oppure normal denota il tipo di segnale, ovvero un errore o una terminazione normale del codice.
- *reason* denota, nel caso di errore, la ragione per cui l'errore si è verificato. Nel caso di *type*=normal, *reason* sarà uguale ad undefined.
- *time* analogo come per i messaggi.Da notare che, **in questa specifica estensione**, il campo *time* non è necessario per la discriminazione dei segnali in sè, bensì, nel caso il *flag* del processo ricevente fosse true e quindi quel segnale fosse convertito in messaggio, serve per discriminare quel particolare messaggio che verrà creato.

In Cauder *Signal* si traduce nel tipo *record signal* composto da:

- *dest = p'*
- *from = p*
- *type = type*
- *reason = reason*
- *time = time*

La struttura di un processo P=$\langle p, \theta, e, h, lm\rangle$ si estende in P=$\langle p, \theta, e, h, lm, l, f\rangle$ dove:

- l denota l'insieme dei *pid* con cui il processo è linkato.
- f denota il valore del *process_flag* del processo.

In Cauder si traduce nell'aggiunta, al tipo *record proc* di un campo *flag=false* e *links=[]*.
La figura sottostante mostra graficamente lo stato di Cauder esteso. Viene aggiunto *GS* che corrisponde a Ψ (*signals*), mentre la struttura dati del processo viene estesa tramite l'aggiunta di *PROC_FLAG* che corrisponde a f (*flag*) ed *LINKS* che corrisponde ad l (*links*).

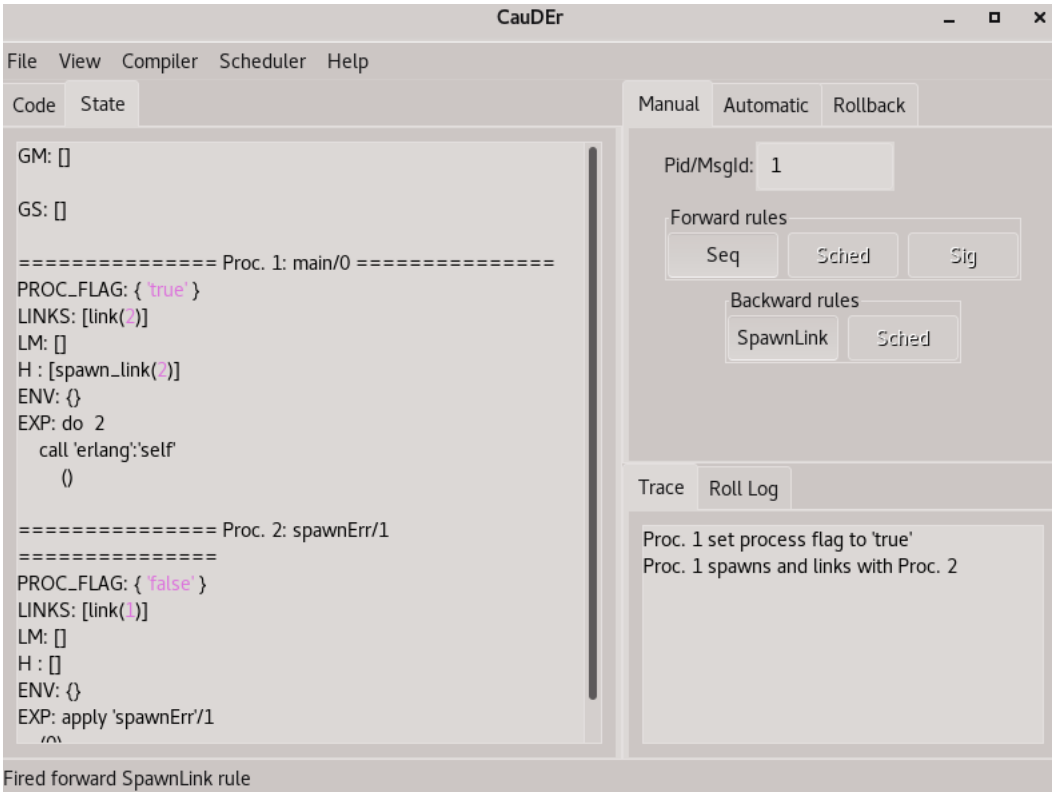

Figura 7: Stat di Cauder esteso.

3.3 Le regole indotte dalla nuova sintassi

Ora che è stata introdotta la nuova sintassi, spiegando quali funzioni vengono aggiunte e quali effetti producono, assieme alle nuove features di sistema, andrò a definire, **come nel punto 3 della sezione 2.2**, la semantica reversibile estesa del nuovo linguaggio, lavorando su un processo generico P esteso *che esegue la regola descritta*, ovvero $\langle p, \theta, e, h, lm, l, f \rangle$ e su un sistema generico S esteso, ovvero $\Gamma; \Psi; \Pi$.
SEMANTICA IN AVANTI:
Alle varie regole che seguiranno, associerò la loro relativa rappresentazione grafica utilizzando il tool *Graphic Viewer* integrato in Cauder. Qui sotto riporto alcune immagini di esempio in cui spiego a grandi linee come interpretare le visualizzazioni del *Graphic Viewer*, con riferimento a [15].
In relazione alla figura sottostante, i numeri rossi in alto, sopra le linee verticali colorate, indicano il pid di un processo. Le linee verticali possono assumere due colorazioni: *verde* se il processo è presente nel sistema ed è attivo, *rosso* se il processo è presente nel sistema ma non è attivo. Una freccia

uscente da un pallino, indica un'azione eseguita da un processo verso un altro processo, con la relativa *label* sopra di essa, indicante che tipo di azione viene eseguita. Se un'azione non coinvolge un secondo processo oppure si ha un invio di qualche informazione senza aver effettuato la relativa ricezione (vedi *send* qui sopra), allora verrà rappresentato solo da un pallino, composta dalla *label e dalla relativa informazione*.

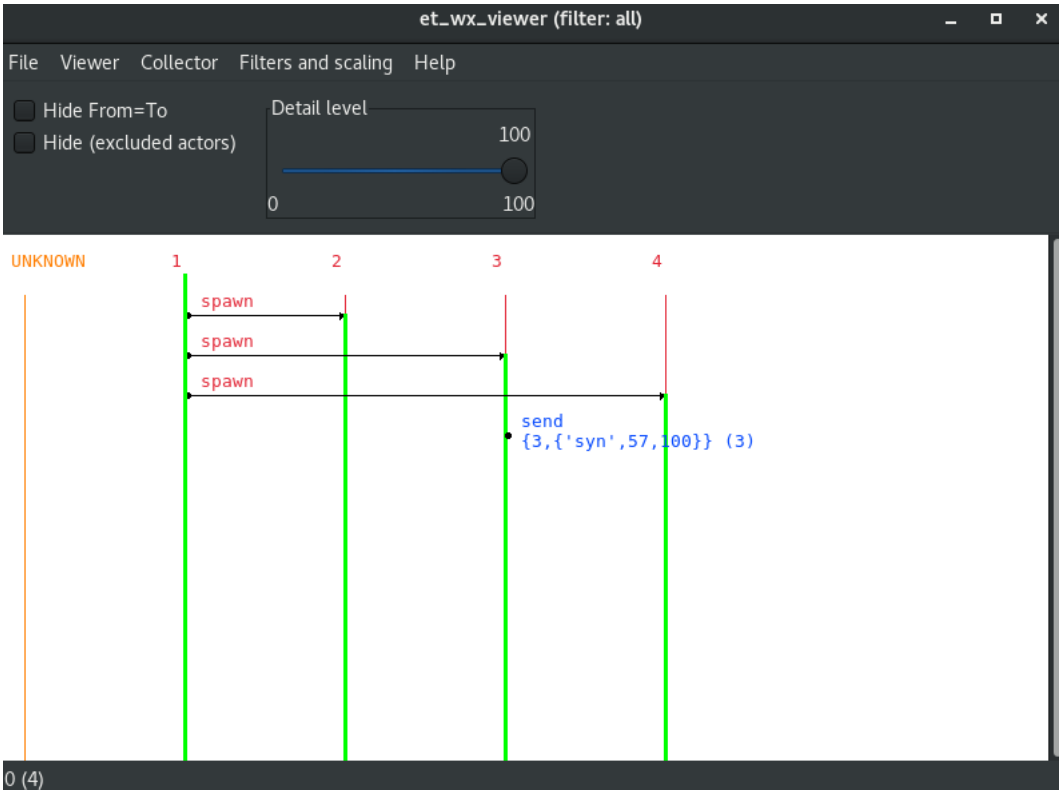

Figura 8: Esempio Graphic Viewer

Invece, nella figura sottostante, vediamo la ricezione della send della figura precedente. Uso una freccia *obliqua*, in modo da evidenziare l'asincronia nella comunicazione di Erlang. La label sopra la freccia riporta l'informazione scambiata.
L'informazione è composta dal *payload*, quindi il vero contenuto informativo e dal *time*, che risulta **sempre** visibile, tra le parentesi tonde a fine label. Questo esempio riprende la send della figura precedente. Questo tipo di visualizzazione verrà usato anche per la propagazione e ricezione di segnali.

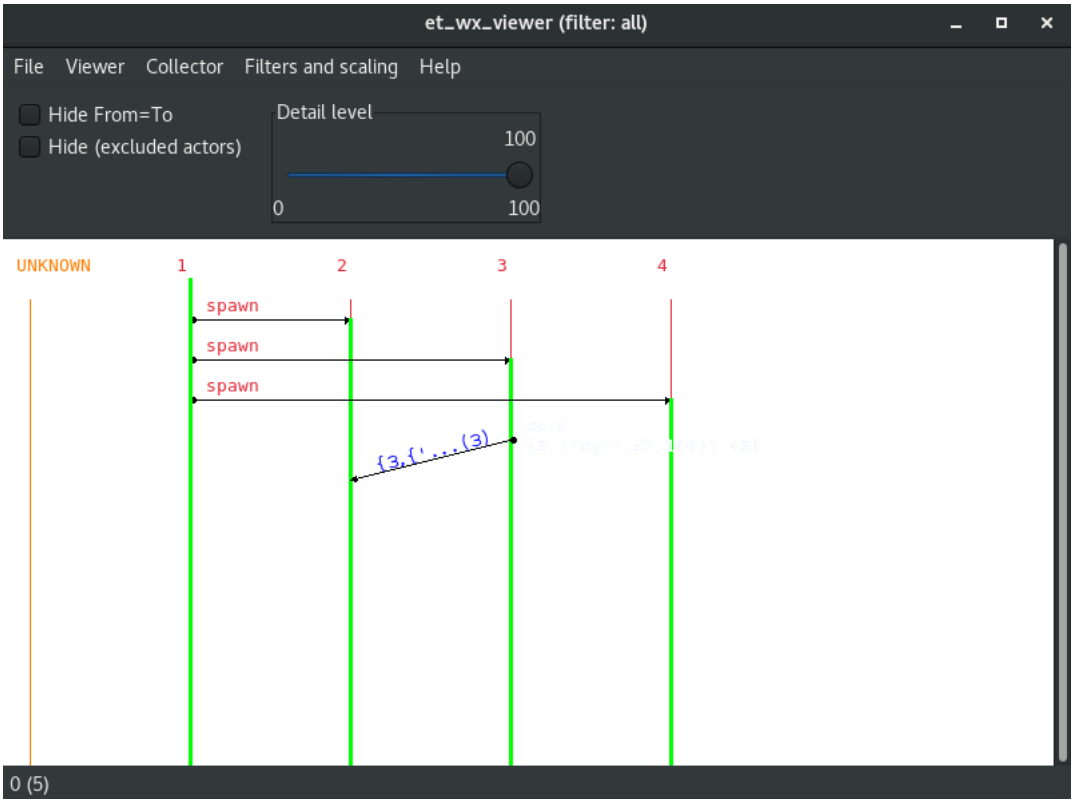

Figura 9: Esempio Graphic Viewer Send/Receive

Nella figura sottostante mostro un esempio di visualizzazione di uno spaccato dell'esecuzione del seguente codice Erlang, a cui associo le varie regole che spiegherò nella semantica in avanti:

```
main()->
        process_flag(trap_exit,true),
        spawn_link(?MODULE,spawnNorm,[0]),
        receive Msg->Msg end.

spawnNorm(2)->self(),self(),3/1;
spawnNorm(N)->
        spawn_link(?MODULE,spawnNorm,[N+1]),
        3/1.
```

Il processo 1 setta il suo *process_flag* a *true* e spawna con link il processo 2. Successivamente il processo 2 spawna con link il processo 3, per poi proseguire con l'intera esecuzione fino alla propria *terminazione del codice*. Alla terminazione, il processo 2 esegue la *propagazione del segnale*. Dopo

di chè vengono schedulati i vari segnali propagati. Da notare come il segnale propagato dopo (con time=2) venga schedulato prima del segnale con time=1.

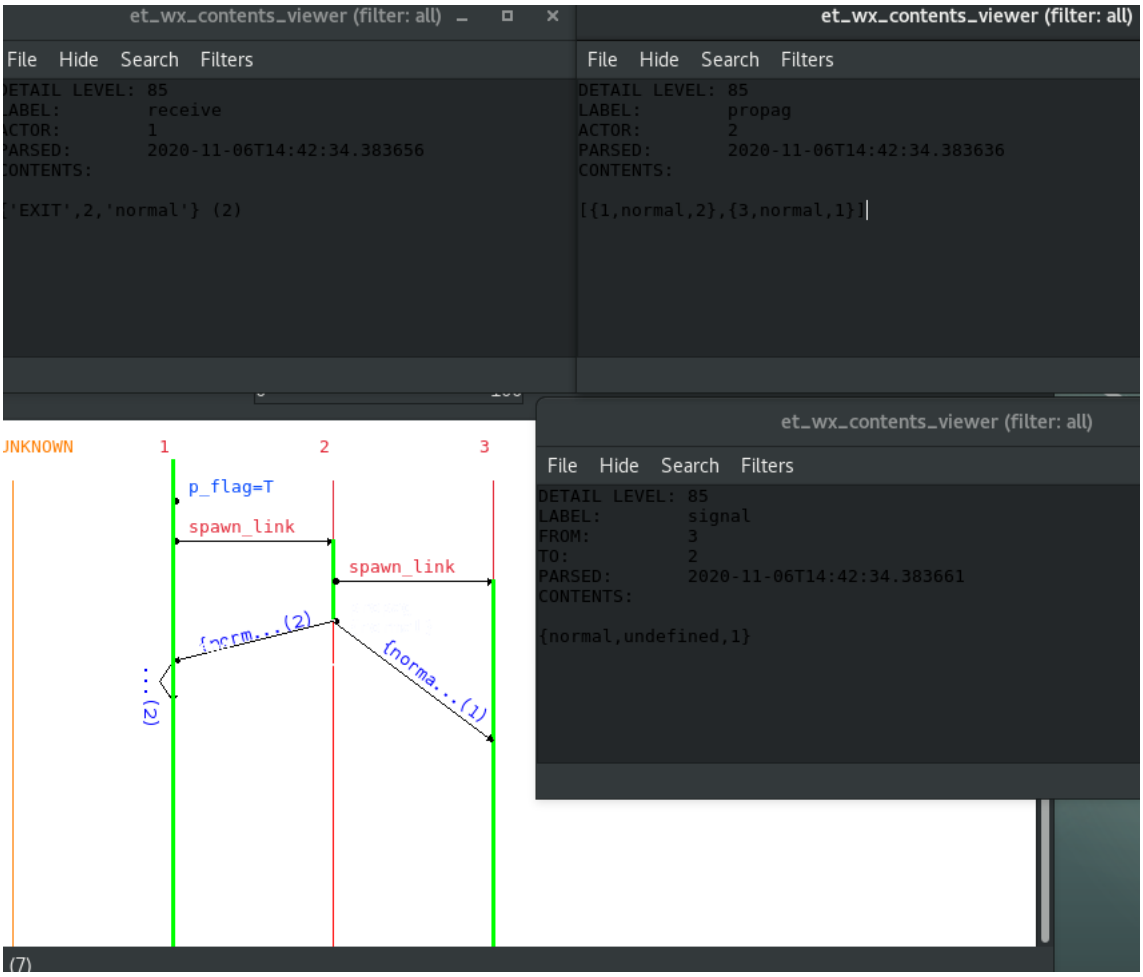

Figura 10: Esempio Graphic Viewer Esteso

- Process_flag: La chiamata a process_flag, oltre a settare il flag del processo al valore booleano passato come argomento, ritorna il vecchio flag del processo che chiamerò *OldBool*. Ciò si concretizza in Cauder, cambiando il campo f di P in f' settato a *Bool*, mentre h di P evolve $h' = [\{\text{process\_flag}, \theta, e, OldBool\} \mid h]$. Sia P'=$\langle p, \theta, e', h', lm, l, f' = Bool\rangle$, lo stato risultante da questo passaggio sarà S'=$\Gamma; \Psi; \Pi \setminus \{P\} \cup \{P'\}$. Nel graphic viewer, in relazione alla figura 10, process_flag viene mostrata tramite la label *p_flag=Bool* con Bool= T (true) or F (false).

- Spawn_link: La funzione spawn_link è molto simile alla funzione spawn, con la differenza che opera anche sui link tra processi, per cui nella descrizione della semantica di essa, tratterò solo la parte relativa ai link. La chiamata a spawn_link, oltre a creare un nuovo processo P' vuoto, setta i relativi link nei due processi coinvolti.
  Più formalmente, viene creato un P'=$\langle p', \theta' = \emptyset, e', h' = [], lm' = [], l' = \{p\}, f' = false\rangle$, mentre P evolve in P"=$\langle p, \theta, e'', h'', lm, l'' = l \cup \{p'\}, f\rangle$ con h"=[{spawn_link, θ, e, p'} | h]. Il sistema risultante da questo passaggio sarà S'=Γ;Ψ;Π\{P}∪{P',P"}. Nel graphic viewer, in relazione a 10, spawn_link viene mostrata tramite la label *spawn_link*.

- Error e exit: Semanticamente, queste due regole sono molto simili tra loro, in quanto aggiungono o tolgono lo stesso tipo di informazioni.Indico con *call*= error se avviene una chiamata ad error o exit se avviene una chiamata ad exit, mentre con *callExp*[5] {exit,Reason} nel caso di chiamata ad exit o {error,Reason,stack} nel caso di chiamata ad error.
  Sia h'=[{$call, \theta, e, Reason$} | h]. Il processo P evolve in P'=$\langle p, \theta, e' = callExp, h', lm, l, f\rangle$, con il sistema S che evolve in S'=$\Gamma; \Psi; \Pi \setminus \{P\} \cup \{P'\}$. Non essendo visualizzato nel tab *trace* di Cauder, non lo rappresento neanche nel graphic viewer.

- Propagazione dei segnali: Concettualmente, la propagazione dei segnali risulta essere molto simile alla *rule send*, in quanto un carico di informazioni viene inviato e immagazzinato in una struttura dati aggiuntiva in attesa di essere consegnata. Può essere vista come una *send uno-a-tanti*.
  Per ogni link di P viene generato un segnale ed inviato al processo linkato, prima che P termini del tutto.
  Più formalmente:
  Sia histSignals= $\bigcup_{link \in l} \{link, type, reason, time\}$.
  Sia h'='=[{propag, $\theta, e, histSignals$} | h].
  Allora P evolve in P'=$\langle p, \theta, e', h', lm, l' = [], f\rangle$.
  Sia signals= $\bigcup_{hist \in histSignals} \{hist.link, p, hist.type, hist.reason, hist.time\}$.
  Allora il sistema S evolve in S'=$\Gamma; \Psi \cup signals; \Pi \setminus \{P\} \cup \{P'\}$.
  Nel graphic viewer la propagazione dei segnali verrebbe mostrata con la

[5]Sarebbe l'espressione risultante dalla chiamata ad una di esse.Da notare che entrambe le chiamate non hanno nessun valore di ritorno, ma gli si associa una delle tuple di seguito per indicare la terminazione e che tipo di terminazione.

label *propag*{ *Type,Reason*} con *Type*=error o normal e *Reason*= undefined se *Type*=normal.
In relazione alla figura 10 la propagazione dei segnali viene mostrata tramite l'et_context_viewer con **label**=propag, con le varie informazioni sui segnali inviati.

- Ricezione segnali: La trattazione dei segnali è molto simile a quella dei messaggi, seppur vengono coinvolte strutture dati diverse e vengano prodotti effetti diversi.
  Il percorso di un segnale è del tipo: $invio \xrightarrow{prop} \Psi \xrightarrow{signal} ricezione$, mentre quello di un messaggio è del tipo: $invio \xrightarrow{send} \Gamma \xrightarrow{sched} lm \xrightarrow{receive} ricezione$.
  Concettualmente la ricezione dei segnali ingloba atomicamente al suo interno la *rule sched* e la *rule receive*, ovviamente trattando un carico di informazione diverso e con effetti diversi. Come precondizione per effettuare una schedulazione del segnale ad un processo P, bisogna che esso sia vivo, ovvero che e sia effettivamente una espressione.
  Questa precondizione si applica a tutte le casistiche che verranno proposte. Posto che vi sia questa condizione, si presentato 4 casistiche relative alla ricezione dei segnali, che dipendono dal valore del *process_flag* del processo ricevente.
  Per ognuna delle casistiche, il sistema S evolve togliendo il segnale schedulato da Ψ e aggiungendo P' e togliendo P ad Π, quindi presenterò solo come evolve P in P', sottintendendo, alla fine, l'evoluzione di S in S'. Quindi selezionato il segnale da schedulare, **in base al time**, se:

  1. signal.type=error $\wedge$ f=true:
     Sia lm'=$[\{\{\mathsf{EXIT}, signal.from, signal.reason\}, signal.time\} \mid lm]$.
     Sia h'=$[\{\mathsf{signal}, signal.from, error, signal.reason, signal.time\} \mid h]$.
     Il processo P evolve in P'=$\langle p, \theta, e, h', lm', l' = l \backslash \{signal.from\}, f\rangle$.
  2. signal.type=error $\wedge$ f=false:
     Sia h'=$[\{\mathsf{signal}, signal.from, error, \theta, e, signal.time\} \mid h]$.
     Il processo P evolve in P'=$\langle p, \theta, e' = \{\mathsf{error}, signal.reason, stack\}, h', lm, l' = l \setminus \{signal.from\}, f\rangle$. Il motivo per cui viene tolto solo il link del processo che ha inviato il segnale, risiede nel fatto che il processo ricevente potrà ripropagare a tutti i suoi processi linkati il segnale di errore appena ricevuto!.
  3. signal.type=normal $\wedge$ f=true: Il processo P evolve in P' in maniera simile al caso 1, con l'unica differenza nel campo h', che viene ag-

giunta la tupla {'signal',signal.from,normal,signal.time} al posto dell'altra, che segnala un terminazione normale del codice.

4. signal.type=normal $\wedge$ f=false: Il processo P evolve in P' in maniera simile al caso 3, con l'unica differenza che non viene creato il messaggio, ma viene solo rotto il link col processo che ha propagato il segnale.

In relazione alla figura 10, l'et_context_viewer con **label**= signal) mostra la ricezione del segnale con time=1 da parte del processo 3.
Il processo 1 avendo il *process_flag* a *true*, dopo la ricezione del segnale con time=2 (**il punto di partenza della freccia spezzata**, avrebbe l'et_context_viewer simile a quello con label=signal), converte in messaggio il segnale ricevuto, e ne effettua la *receive*, com mostrato nel et_context_viewer con **label**= receive.

SEMANTICA ALL'INDIETRO:

- Process_flag: Avendo h=[{process_flag, $\theta_{old}, e_{old}, OldBool\}$ | h'], per effettuare l'undo del process flag, banalmente si crea un P'= $\langle p, \theta_{old}, e_{old}, h', lm, l, f' = OldBool\rangle$.
  Il sistema evolve come nella semantica in avanti, togliendo P ed inserendo P'.

- Spawn_link: Per poter effettuare un passo all'indietro della spawn_link, si ha la stessa precondizione relativa alla spawn, ovvero il processo spawnato deve essere "vuoto".
  Avendo h=[{spawn_link, $\theta_{old}, e_{old}, p_{spawn}\}$ | h'],
  si crea un processo P"=$\langle p, \theta_{old}, e_{old}, h', l' = l \setminus \{p_{spawn}\}, f\rangle$. Il sistema evolve in S'=$\Gamma; \Psi; \Pi \setminus \{P, P'\} \cup \{P''\}$.

- Error e exit: Sia h=[$\{call, \theta_{old}, e_{old}, Reason\}$ | h'].
  Banalmente, il processo P evolve in P'=$\langle p, \theta_{old}, e_{old}, h', lm, l, f\rangle$ ed S evolve allo stesso modo della semantica in avanti.

- Propagazione dei segnali: Per effettuare l'undo di una propagazione dei segnali, bisogna che tutti i segnali inviato dal processo morente siano presenti in Ψ. Se non si rispettasse questa condizione, quindi si potesse fare l'undo in ogni momento della propagazione, si rischierebbe di violare la consistenza causale.
  Ipotizziamo di avere Ψ=[{1,2,error,badarg,1},{3,2,error,badarg,2}], a seguito di una propagazione da parte del processo 2. Ora se schedulassi in avanti il segnale di tempo=1 avrei Ψ=[{3,2,error,badarg,2}], ma

cosa ben più importante, essa ha prodotto degli effetti sul processo che lo riceve(più avanti si vedrà la regola della consegna dei segnali). Se ora annullassi la propagazione senza aver annullato prima la schedulazione del segnale, otterrei che il processo 1 ha subito degli effetti da parte della schedulazione di un segnale che non è mai esistito. Posta che vi sia questa condizione, l'undo della propagazione ritira tutti i segnali emessi dal processo morente, ricreando i link rotti. Più formalmente:
Sia h=[{propag, $\theta_{old}, e_{old}, histSignals\}$ | h'].
Sia oldLinks= $\bigcup_{histSig \in histSignals} histSig.link$.
Sia signals= $\bigcup_{histSig \in histSignals} \{histSig.link, p, histSig.type, histSig.reason, histSig.time\}$.
Allora P'=$\langle p, \theta_{old}, e_{old}, h', lm, l = oldLinks, f\rangle$. Il sistema S evolve in S'=$\Gamma; \Psi \setminus signals; \Pi \setminus \{P\} \cup \{P'\}$.

- Ricezione segnali: Anche nell'undo della schedulazione di un segnale si presentato 4 casistiche, in funzione delle informazioni in h ed eventualmente di f in quell'istante. Ogni casistica può avere una sua precondizione, che definisco in quella specifica casistica. In ogni casistica, dopo aver evoluto P in P', ciò che viene fatto è:

  1. Ricreare il segnale *signal*, date le informazioni necessarie nella history, ed evolvere Ψ in $\Psi'=\Psi \cup \{\text{signal}\}$.
  2. Evolvere Π in $\Pi'=\Pi \setminus P \cup P'$.
  3. Evolvere il sistema S in S'=$\Gamma; \Psi'; \Pi'$.

  Qui di seguito espongo la regola dell'undo di un segnale,secondo le varie casistiche, illustrando solo come evolve P in P':

  1. h=[{signal, $from$, error, $reason, time\}$ | h']:
     Indica un ricezione di un segnale di errore $\wedge$ f=true(vedi h del caso 1 nella semantica in avanti). Come precondizione, questa casistica necessita che il relativo messaggio derivato dal segnale, *sia in cima ad lm*, ovvero lm=[{{EXIT, $signal.from, signal.reason\}, signal.time\}$ | lm']. Se non ci fosse questa precondizione, si rischierebbe di violare il Loop Lemma. Ipotizziamo di essere in questa situazione: lm=[{{EXIT,error,reason},2},{"ciao",3}].Se facessi l'undo del segnale, per poi rischedularlo al passo successivo, mi troverei con lm=[{"ciao",3},{{EXIT,error,reason},2}], ovvero il messaggio generato dal segnale si ritroverebbe in testa. Da uno stato S, ho fatto l'undo di un'azione per poi rifarla e mi son ritrovato in uno

stato S'≠S, infrangendo il Loop Lemma. Detto questo, avendo lm=[{{EXIT, $signal.from, signal.reason\}, signal.time\} \mid lm'$], si ha P'=$\langle p, \theta, e, h', lm', l \cup \{from\}, f\rangle$. Da notare che θ ed e rimangono inalterati.

2. h=[{signal, $from$, error, $\theta_{old}, e_{old}, time\} \mid h'$]:
   Indica una ricezione di un segnale di errore $\wedge$ f=false(vedi h del caso 2 nella semantica in avanti). Per la ricostruzione del segnale *signal*, *reason* la posso tranquillamente ricavare dal secondo elemento di e. Detto ciò, evolvo P in P'=$\langle p, \theta_{old}, e_{old}, lm, l \cup \{from\}, f\rangle$.
3. h=[{signal, $signal.from$, normal, $signal.time\} \mid h'$]:
   Indica una ricezione di terminazione normale. In questo caso il tipo di comportamento della regola, lo posso discriminare in base a f in quell'istante. *Sia* f=true.
   Allora diventa analogo al caso 1 della semantica all'indietro, ovviamente ricostruendo un *signal* di type=normal. *Sia* f=false.
   Banalmente P evolve in P'=$\langle p, \theta, e, h', lm, l \cup \{from\}, f\rangle$.

ROLLBACK:
Risultano essere interessanti solo due regole ai fini dell'operatore *roll*, in quanto sono gli unici che risultano avere conseguenze che si ripercuotono su altri processi. A dire il vero ne esiste una terza (roll_signal) ma, come si vedrà a breve, essa viene inglobata nella *roll della propagazione dei segnali*.

- Propagazione segnali:
  Sia h=[{propag, $\theta, e, histSignals\} \mid h'$]. Bisogna creare la condizione per cui si possa ripropagare all'indietro. Algoritmicamente, la roll di un propagazione si sviluppa in questo modo:

Algorithm 4 roll_propag(State,Pid)

```
for all histSignal ∈ histSignals do
  Signal=recreateSignal(histSignal,Pid)
  roll_signal(StateAcc,Signal)
end for
return back_propag_step(NewState,Pid)
```

Assumo che *NewState* venga ritornato alla fine del *forall* ed ad ogni passo del *forall*, *StateAcc* sia lo stato ritornato dalla precedente chiamata a *roll_signal*. All'inizio *StateAcc=State*. La funzione *recreateSignal* si

occupare di ricreare il *record signal* con le informazioni che gli vengono passate. A livello molto generico ed informale, la *roll_signal* va a creare la condizione per cui il segnale possa essere ritirato. Più in dettaglio, se il segnale può essere ritirato direttamente allora termina, altrimenti fai fare un passo indietro *al processo che lo ha ricevuto* e rivaluta *roll_signal* sul segnale ricorsivamente, finchè non si arriva al caso base. In termini algoritmici, la roll_signal si sviluppa in questo modo:

Algorithm 5 roll_signal(State,Signal)

```
if Signal ∈ Ψ then
  return State
else
  State'=back_step(State,Signal.p')
  roll_signal(State',Signal)
end if
```

- Spawn_link:
  La roll della spawn_link risulta essere identica alla roll della spawn. Ovvero viene creata la condizione per cui una spawn possa essere invertita, cioè si annullano **tutte** le azione effettuate dal processo spawnato, per poi invertirla, secondo la regola all'indietro della spawn_link.

3.4 Un esempio di Cauder esteso

Il codice che mostro qui sotto implementa una comunicazione client-server.
Un *server* riceve delle richieste da 2 *client*. Questi 2 *client* inviano ognuno 2 valori presi da una lista predefinita di valori per ottenere in output dal *server* una divisione.
Finchè non riesce a soddisfare una richiesta, il *client* continua ad inviare richieste, scegliendo ogni volta degli input casuali dalla lista di valori.
In questa lista sono presenti dei valori non numerici per simulare una richiesta malformata che il *server* deve gestire.
Ad ogni richiesta ricevuta il *server* spawna un *worker* che esegue effettivamente la divisione e che si occupa di inviarla al rispettivo *client*. Nel caso il *worker* non riuscisse a gestire la richiesta, il server si occuperà di riferire al client l'errore.

```
-module(clientServer).
-export([main/0,server/1,client/1,worker/3]).

main()->
  Pid=spawn(?MODULE,server,[[]]),
  spawn(?MODULE,client,[Pid]),
  spawn(?MODULE,client,[Pid]).

client(Server)->
 %%possibili input per la richiesta al server
  Terms=[49,ciao,78,98,0,"term",1000,4.5,74569,259.326,1],
  Inputs={util:get_random_el(Terms),util:get_random_el(Terms)},
  Server !{calculate,self(),Inputs},
  receive
    {ret,badarg}->client(Server);
    {ret,Val}->
          Val
  end.

server(Map)->
  receive
    {calculate,Client,{First,Second}}->
      Worker=spawn_link(?MODULE,worker,[Client,First,Second]),
          server([{Worker,Client}|Map]);
        {'EXIT',Worker,normal}->
          server(util:remove(Worker,Map));
        {'EXIT',Worker,_}->
          Client=util:search(Worker,Map),
          Client ! {ret,badarg},
          server(util:remove(Worker,Map))
  end.

worker(Client,First,Second)->
  Val=First/Second,
  Client ! {ret,Val}.
```

Figura 11: client-server

Il video può essere trovato al link: https://github.com/Taba92/Tesi al file esempio.mk.

4 CONCLUSIONI

Qui finisce la spiegazione dell'estensione tramite link di Cauder. Questa è un'implementazione basilare ma che copre ogni aspetto riguardante i link e la gestione degli errori. Un parola va data alle funzioni error e exit. Queste due funzioni non hanno valori di ritorno ma solo effetti collaterali mentre in Cauder essi vanno a cambiare *P.e.* Ma se il programma terminasse con la tupla {error, *Reason*, stack} scelta dal programmatore? Cauder la gestirebbe come un errore mentre invece è una terminazione normale del codice. Una possibile soluzione potrebbe essere quella di aggiungere al *record proc* un campo *is_alive* che è true se il processo è attivo oppure è {false, {*Type*, *Reason*}} nel caso il processo fosse terminato. Per quanto riguarda i link esistono altre 2 funzioni utilizzabili:

- link(Pid) $\xrightarrow{return}$ true: Crea un collegamento tra il processo chiamante e il process *Pid*. Se il collegamento esiste già o un processo tenta di creare un collegamento a se stesso, non viene eseguita alcuna operazione. Restituisce truese il collegamento viene impostato. Se *Pid* non esiste, viene generato un errore noproc.

- unlink(Pid) $\xrightarrow{return}$ true: Rimuove il collegamento, se presente, tra il processo chiamante e il processo *Pid*. Restituisce true e non ha esito negativo, anche se non è presente alcun collegamento a *Pid* o se *Pid* non esiste.

Per quanto riguarda i segnali si è visto solo la propagazione dovuta a terminazione come mezzo di invio, che è un effetto collaterale. Esiste una funzione apposita per l'invio dei segnali analoga alla *send*:

- exit($Pid, Reason$) $\xrightarrow{return}$ true: Invia un segnale di uscita con motivo uscita *Reason* al processo identificato da *Pid*. Il comportamento in base a *Reason* è lo stesso visto fino ad ora tranne per una specifica *Reason*, ovvero *Reason*=kill. In questo caso viene inviato un segnale di uscita **untrappable** a *Pid*, che esce incondizionatamente con *Reason*=killed.

In Erlang esiste una seconda metodologia per la gestione dell'error-handling ovvero l'uso dei *monitor*. Ne parlerò brevemente per darne un'idea generale del loro funzionamento. Quando un processo *Pid* monitorato esce per un qualsiasi motivo, il processo monitorante riceve un messaggio del tipo {DOWN, $MonitorRef, Type, Pid, Reason$} dove:

- *MonitorRef*: è il riferimento al monitor.

- *Type*: error o exit.

- *Reason*: è il motivo di uscita del processo *Pid*.

Le principali differenze tra i *monitor* e i *link* sono le seguenti:

- I monitor sono unidirezionali: Se il processo monitorante terminasse per **qualsiasi** motivo prima del processo monitorato, quest'ultimo non ne avrebbe ripercussioni.
- I monitor utilizzano solo i *messaggi* per le notifiche di uscita.
- Un processo può avere più di un *monitor*.

Questo argomento è stato molto illuminante poichè fino ad ora non avevo mai utilizzato un debugger. Estenderne uno reversibile ha cambiato radicalmente la mia opinione riguardo ad essi. Dovendolo estendere son stato costretto a doverlo prima utilizzare per poter capirne il flusso di esecuzione. Provandolo su programmi di esempio mi son reso conto effettivamente della loro utilità. Riguardo all'effettiva estensione di Cauder sono rimasto stupito dal notare parecchie analogie con un interprete, una cosa su cui non avevo mai riflettuto fino adesso. In secondo luogo mi son reso conto dell'importanza di scrivere codice ben strutturato in modo tale da poterlo riprendere anche dopo molto tempo. Soprattutto se il programma non è scritto da te e non sarai l'ultimo a scriverci. Rimanere in linea con la struttura del codice già scritto porta ad una comprensione nei minimi dettagli di come e su cosa lavora il programma. Inoltre son rimasto molto affascinato dal campo della reversibilità causal-consistent. Più in specifico è stato molto edificante studiarne le basi teoriche, capirne i ragionamenti e i perchè, per poi applicarne i frutti su un progetto reale.

Riferimenti bibliografici

[1] Mike Vizard. "Report: Debugging Efforts Cost Companies $61B Annually". In: (2020). URL: https://devops.com/report-debugging-efforts-cost-companies-61b-annually/.

[2] University of Cambridge. "Research by Cambridge MBAs for tech firm Undo finds software bugs cost the industry $316 billion a year". In: (2013). URL: https://www.jbs.cam.ac.uk/insight/2013/research-by-cambridge-mbas-for-tech-firm-undo-finds-software-bugs-cost-the-industry-316-billion-a-year/.

[3] Erlang Official Documentation. *Erlang Documentation*. Rapp. tecn. URL: https://www.erlang.org/docs.

[4] Erlang Official Documentation. *Erlang SMP*. Rapp. tecn. URL: http://erlang.org/doc/efficiency_guide/processes.html#smp-emulator.

[5] Vincent Danos e Jean Krivine. "Reversible Communicating Systems". In: *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. A cura di Philippa Gardner e Nobuko Yoshida. Vol. 3170. Lecture Notes in Computer Science. Springer, 2004, pp. 292–307. DOI: 10.1007/978-3-540-28644-8_19. URL: https://doi.org/10.1007/978-3-540-28644-8%5C_19.

[6] Adrian Palacios et al. "Cauder source code". In: (2017). URL: https://github.com/mistupv/cauder.

[7] Core Erlang. *Core Erlang by Example*. Rapp. tecn. URL: http://blog.erlang.org/core-erlang-by-example/.

[8] Rolf Landauer. "Irreversibility and Heat Generation in the Computing Process". In: *IBM J. Res. Dev.* 5.3 (1961), pp. 183–191. DOI: 10.1147/rd.53.0183. URL: https://doi.org/10.1147/rd.53.0183.

[9] Wikipedia. "Computazione reversibile". In: (2020). URL: https://it.wikipedia.org/wiki/Computazione_reversibile.

[10] Irek Ulidowski et al., cur. *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*. Vol. 12070. Lecture Notes in Computer Science. Springer, 2020. ISBN: 978-3-030-47360-0. DOI: 10.1007/978-3-030-47361-7. URL: https://doi.org/10.1007/978-3-030-47361-7.

[11] Ivan Lanese. “From Reversible Semantics to Reversible Debugging”. In: *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*. A cura di Jarkko Kari e Irek Ulidowski. Vol. 11106. Lecture Notes in Computer Science. Springer, 2018, pp. 34–46. DOI: 10.1007/978-3-319-99498-7_2. URL: https://doi.org/10.1007/978-3-319-99498-7%5C_2.

[12] Ivan Lanese et al. “CauDEr: A Causal-Consistent Reversible Debugger for Erlang”. In: *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings*. A cura di John P. Gallagher e Martin Sulzmann. Vol. 10818. Lecture Notes in Computer Science. Springer, 2018, pp. 247–263. DOI: 10.1007/978-3-319-90686-7_16. URL: https://doi.org/10.1007/978-3-319-90686-7%5C_16.

[13] Erlang Official Documentation. *Erlang Communication*. Rapp. tecn. URL: https://erlang.org/doc/apps/erts/communication.html.

[14] Erlang Official Documentation. *Erlang Links*. Rapp. tecn. URL: https://erlang.org/doc/reference_manual/processes.html#links.

[15] Erlang Official Documentation. *Erlang Event Tracer Documentation*. Rapp. tecn. URL: http://erlang.org/doc/apps/et/et_intro.html.