Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

ANALISI DI CONTRATTI LEGALI IN STIPULA

Relatore:
Chiar.mo Prof.
COSIMO LANEVE

Presentata da:
SAMUELE EVANGELISTI

Correlatore:
Dott.
ALESSANDRO PARENTI

Sessione III
Anno Accademico 2022/2023

A chi ce l'ha fatta, a chi ce la sta facendo;
a chi non ce l'ha fatta. Per chi non ce l'ha potuta fare.
Congratulazioni, incoraggiamenti;
consolazioni. Rammarico.

"Il successo non è definitivo, il fallimento non è fatale:
è il coraggio di andare avanti che conta."
[Winston Churchill]

Abstract

I contratti legali sono insiemi di clausole che specificano protocolli in grado di regolare le interazioni tra le diverse entità. Questa clausole possono contenere errori, come per esempio la definizione di regole che non possono mai essere applicate a causa di circostanze irraggiungibili o a causa di vincoli temporali sbagliati. Quindi è importante poter riconoscere ed eliminare queste clausole errate in quanto non sono significative per il contratto.

In questa tesi analizzeremo il problema delle clausole errate presenti nei contratti legali scritti in Stipula, un linguaggio specifico per la progettazione di contratti legali. In particolare verrà riportata l'attenzione sul linguaggio Stipula, verrà presentata la teoria della raggiungibilità, verrà presentata l'implementazione dell'analizzatore e, infine, verranno eseguiti alcuni test.

Indice

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Contratti

I contratti legali sono insiemi di clausole che specificano protocolli in grado di regolare le interazioni tra le diverse entità. Questa clausole possono contenere errori, come per esempio la definizione di regole che non possono mai essere applicate a causa di circostanze irraggiungibili o a causa di vincoli temporali sbagliati. Quindi è importante poter riconoscere ed eliminare queste clausole errate in quanto non sono significative per il contratto.

Il riconoscimento delle clausole errate di un contratto è in generale un'operazione non banale. Quando si esegue un contratto legale è rilevante sia l'ordine con cui entrano in gioco le diverse clausole, sia i vincoli temporali che queste definiscono. Per questo motivo, durante l'analisi di un contratto legale, è necessario considerare contemporaneamente entrambi gli aspetti.

Generalmente i contratti legali sono scritti in linguaggio naturale. L'utilizzo del linguaggio naturale porta con sé alcuni aspetti significativi quali l'ambiguità e la non trascurabile complessità che si genera nei tesi dei contratti. Questi aspetti aggiungono difficoltà all'analisi dei contratti legali e rendono necessario considerare anche l'aspetto strutturale riferito alla scrittura dei contratti stessi.

Per risolvere i problemi elencati è possibile utilizzare un linguaggio di

programmazione in quanto, grazie alla sua definizione formale, un linguaggio di programmazione ben fatto è in grado di eliminare l'ambiguità ed è in grado di ridurre drasticamente la complessità del testo. In questa tesi si prenderà come riferimento Stipula [8, 13, 9], un linguaggio di programmazione specifico per la progettazione di contratti legali. L'utilizzo di un linguaggio di programmazione è possibile grazie al principio di libertà di forma riconosciuto dai moderni ordinamenti giuridici [15].

Stipula permette di definire in maniera più semplice un contratto legame ma non è sufficiente a impedire la presenza di clausole errate. I motivi per cui una clausola può essere errata vanno oltre la definizione formale del linguaggio. I principali motivi per cui una clausola potrebbe essere errata sono riconducibili all'esecuzione del contratto che la contiene, in particolar modo ai tempi di esecuzione delle clausole. Di conseguenza, è possibile produrre un contratto, scritto in Stipula, che sia sintatticamente e semanticamente corretto ma che contenga clausole che non verranno mai eseguite. Essendo Stipula un linguaggio di programmazione, è comunque possibile svolgere un'analisi statica dei contratti scritti in Stipula allo scopo di determinare le caratteristiche delle clausole. Si sottolinea che le caratteristiche che si vogliono analizzare emergono in fase di esecuzione, quindi ottenerle tramite un'analisi statica comporta comunque alcune difficoltà.

Un contratto scritto in Stipula ha la forma di un automa a stati finiti. I contratti scritti in Stipula sono composti da stati e clausole in grado di portare da uno stato a un altro. Inoltre, per ogni contratto, è presente un orologio globale con lo scopo di tracciare il tempo di esecuzione. Le clausole sono di due tipi: funzioni ed eventi. Le funzioni sono invocabili dalle parti legali del contratto e possono essere invocate quando il contratto si trova allo stato di inizio della funzione. Terminata la funzione il contratto passerà allo stato di fine della funzione. Gli eventi vengono definiti all'interno delle funzioni e vengono generati dall'invocazione delle funzioni che li definiscono. Gli eventi dispongono di un tempo di innesco derivante dall'analisi dell'espressione temporare a loro associata, nel momento della generazione. Un evento,

una volta generato, verrà messo in attesa e verrà eseguito automaticamente se il contratto si troverà allo stato di inizio dell'evento nell'istante temporale identificabile come tempo di innesco dell'evento.

A questo punto è possibile introdurre due caratteristiche che permettono di identificare a priori le clausole errate. La prima caratteristica è topologica. Un contratto inizia nel suo stato iniziale e passa di stato in stato tramite l'esecuzione delle sue clausole. Una clausola non è mai eseguibile se non è possibile raggiungerla dallo stato iniziale del contratto. Questa analisi, nel caso degli eventi, diventa ancora più stringente. Infatti, gli eventi vengono generati invocando le funzioni nelle quali sono definiti. Quindi, partendo dallo stato iniziale del contratto, deve essere possibile prima eseguire la funzione e successivamente eseguire l'evento. La seconda caratteristica è temporale ed è relativa solamente agli eventi. Gli eventi dispongono di un tempo di innesco che deve essere rispettato per poter eseguire l'evento. Viceversa, l'evento viene semplicemente scartato da quelli in attesa. Quindi per poter eseguire un evento, il contratto dovrà trovarsi allo stato di inizio dell'evento, al tempo di innesco dell'evento. Se non è mai possibile soddisfare la condizione descritta allora l'evento non verrà mai eseguito.

Si fa da subito presente che un contratto scritto in Stipula può contenere al suo interno dei cicli. Questa condizione fa si che alcune funzioni possano essere eseguite ciclicamente e, nel caso definiscano eventi, generare ciclicamente gli eventi definiti. In questo caso l'analisi delle clausole presenta delle complicanze per cui non sarà più possibile considerare la caratteristica temporale. Più nel dettaglio, una funzione potrebbe generare ripetutamente gli eventi che definisce, in istanti temporali differenti. Di conseguenza gli eventi potrebbero essere duplicati ma essere associati a tempi di innesco differenti. Questo comportamento è analizzabile in fase di esecuzione ma difficilmente analizzabile tramite un'analisi statica.

In questa tesi, per ovviare al problema delle computazioni infinite, si sono utilizzate le computazioni astratte ovvero sequenze di clausole, senza ripetizioni, in grado di descrivere le computazioni. Il limite delle computazioni

astratte sta nel fatto che non sono in grado di descrivere esplicitamente i cicli presenti nelle computazioni, così come non sono in grado di descrivere esplicitamente quante volte il ciclo è stato eseguito e in quali tempi. Quindi non si hanno a disposizione le informazioni relative ai gruppi di eventi generati. Ad ogni modo, si è cercato di effettuare comunque un'analisi anche dei contratti che contengono cicli, considerando opportunamente i tempi logici di esecuzione in concomitanza con l'incertezza che i vari cicli possono generare.

1.2 Struttura

In questa tesi si descrive l'analisi effettuata sui contratti e l'implementazione del prototipo dell'analizzatore in grado di identificare le clausole errate. L'analizzatore è anche in grado di riconoscere condizioni e vincoli da rispettare durante l'esecuzione del contratto in analisi per fare in modo di non generare, in fase di esecuzione, clausole non eseguibili.

Nel capitolo (2) viene riportata l'attenzione sul linguaggio Stipula e sulle sue caratteristiche. La comprensione di come è definito il linguaggio Stipula è un requisito fondamentale per le successive fasi di analisi dei contratti. Vengono anche proposti alcuni esempi di contratti scritti in Stipula allo scopo di rendere chiara la sintassi e il comportamento dei costrutti del linguaggio.

Nel capitolo (3) viene presentata la teoria della raggiungibilità, fondamentale per l'implementazione dell'analizzatore. La teoria della raggiungibilità rappresenta la formalizzazione di funzioni, proprietà e algoritmi con il fine di riconoscere le clausole errate, poi definite irraggiungibili. Anche in questo caso vengono riportati alcuni contratti di esempio allo scopo di rendere chiaro il comportamento di quanto definito.

Nel capitolo (4) viene presentata l'implementazione dell'analizzatore. Vengono riportate le tecnologie utilizzate, la struttura del codice e le scelte progettuali. Infine viene descritto l'output atteso dall'analizzatore e la sintassi per l'utilizzo di questo strumento.

Nel capitolo (5) vengono riportati i risultati dei test, effettuati utilizzando contratti definiti ad hoc, con lo scopo sia di valutare l'efficacia dell'analizzatore, sia di mostrare come viene prodotto l'output nei vari casi in cui viene utilizzato.

Nel capitolo (6) vengono riassunti i punti salienti di tutta la tesi presentando le conclusioni di questo lavoro.

Capitolo 2

Stipula

2.1 Stipula

I contratti legali sono insiemi di clausole che specificano protocolli in grado di regolare le interazioni tra le diverse parti in termini di autorizzazioni, obblighi e divieti. Secondo i moderni ordinamenti giuridici, questi protocolli possono essere espressi dalle parti usando il linguaggio e il mezzo che preferiscono (principio di libertà di forma) [15], incluso un linguaggio di programmazione. I benefici dell'utilizzo di linguaggi di programmazione sono evidenti e sono stati riconosciuti da diversi progetti [12, 14, 16]. Tra i principali benefici si trovano:

- riduzione dei costi complessivi delle transizioni, coinvolte nel ciclo di vita di un contratto, identificando potenziali inconsistenze;
- riduzione della complessità del testo;
- riduzione dell'ambiguità del testo;
- esecuzione automatica delle clausole.

Per questi motivi in [7, 8, 13] è stato sviluppato un linguaggio specifico per contratti legali chiamato Stipula. Il linuaggio Stipula disponde di:

- una semantica operazionale in modo da specificare completamente il comportamento di un contratto [8];
- una toolchain che include l'implementazione di un prototipo, un'interfaccia grafica e un'estensione per la modifica dei contratti in fase di esecuzione [9].

2.2 Sintassi e semantica

$$\texttt{stipula C}\{\texttt{assets } \overline{\texttt{h}} \quad \texttt{fields } \overline{\texttt{x} = \texttt{v}}, \overline{\texttt{x}'} \quad \texttt{init Q} \quad F\}$$

Functions	F	::=	$\_ \mid @\texttt{Q A} : \texttt{f}(\overline{\texttt{y}})[\overline{\texttt{k}}]\{S\ W\} \Rightarrow @\texttt{Q}'\ F$
Prefixes	P	::=	$E \to \texttt{x} \mid E \to \texttt{A} \mid E \multimap \texttt{h}, \texttt{h}' \mid E \multimap \texttt{h}, \texttt{A}$
Statements	S	::=	$\_ \mid P\ S \mid \texttt{if}(E)\{S\}\texttt{else}\{S\}\ S$
Events	W	::=	$\_ \mid \texttt{t} \gg @\texttt{Q}\{S\} \Rightarrow @\texttt{Q}'\ W$
Expressions	E	::=	$v \mid X \mid E\ \texttt{op}\ E \mid \texttt{uop}\ E$
Values	v	::=	$n \mid \texttt{false} \mid \texttt{true} \mid s \quad (n \in \texttt{Float}, s \in \texttt{String})$
Time expressions	$\texttt{t}$	::=	$\texttt{now} \mid \texttt{t} + \texttt{x} \mid \texttt{t} + n \qquad (n \in \texttt{Nat})$

Tabella 2.1: Sintassi di Stipula

Un contratto legale scritto in Stipula deve rispettare la sintassi definita nella tabella (2.1) dove `C` è il nome del contratto. I contratti sono composti da:

- un insieme di asset, chiamati $\texttt{h}, \texttt{k}, ...$, e un insieme di campi, chiamati $\texttt{x}, \texttt{x}', \texttt{y}, ...$. I campi possono essere inizializzati, gli asset sono inizialmente vuoti.
- un insieme di stati, chiamati $\texttt{Q}, \texttt{Q}', ...$. Lo stato iniziale del contratto è definito dal costrutto `init`.
- una sequenza F di funzioni.

Per semplificare la sintassi viene usata la notazione $\overline{\texttt{x}}$ e $\overline{\texttt{h}}$ che indica sequenze anche vuote di elementi.

Un contratto scritto in Stipula può passare da uno stato ad un altro tramite:

- invocazione di una funzione;
- esecuzione di un evento.

Le *Functions* F, chiamate $\mathtt{f}, \mathtt{g}, ...$, sono invocate dalle parti, chiamate $\mathtt{A}, \mathtt{B}, ...$, e definiscono lo stato da cui l'invocazione è ammessa. I parametri di funzione sono divisi in due liste:

- i parametri che rappresentano campi, chiamati $\overline{\mathtt{y}}$, tra parentesi tonde;
- i parametri che rappresentano asset, chiamati $\overline{\mathtt{k}}$, tra parentesi quadre.

Infine il corpo $S\ W \Rightarrow \mathtt{Q}'$ specifica gli *Statements* S, gli *Events* W e lo stato $\mathtt{Q}'$ su cui termina la funzione. Si assume che $\overline{\mathtt{y}}$ e $\overline{\mathtt{k}}$ non occorreranno mai all'interno di W in modo che gli eventi vengano correttamente eseguiti all'esterno dello scope della funzione.

Gli *Statements* S sono sequenze di *Prefixes* P. I simboli $\rightarrow$ e $\multimap$ differenziano le operazioni rispettivamente sui campi e sugli asset. Il significato dei *Prefixes* P è il seguente:

- $E \rightarrow \mathtt{x}$ aggiorna il capo o il parametro $\mathtt{x}$ con il valore di E;
- $E \rightarrow \mathtt{A}$ invia il valore di E alla parte $\mathtt{A}$;
- $E \multimap \mathtt{h}, \mathtt{h}'$ sottrae il valore di E ad $\mathtt{h}$ e lo aggiunge ad $\mathtt{h}'$;
- $E \multimap \mathtt{h}, \mathtt{A}$ sottrae il valore di E ad $\mathtt{h}$ e lo trasferisce ad $\mathtt{A}$.

Le risorse contenute negli asset possono essere trasferite ma non distrutte. La semantica operazionale previene asset con valori negativi. Lo *Statements* $\mathtt{if}(E)\{S\}\mathtt{else}\{S'\}$ è il construtto condizionale e ha la semantica standard. Nel resto della tesi verranno sempre abbreviate $\mathtt{h} \multimap \mathtt{h}, \mathtt{h}'$ e $\mathtt{h} \multimap \mathtt{h}, \mathtt{A}$ rispettivamente in $\mathtt{h} \multimap \mathtt{h}'$ e $\mathtt{h} \multimap \mathtt{A}$.

Gli *Events* W sono sequenze di clausole temporizzate che schedulano codice per esecuzioni future. $\mathtt{t} \gg \mathtt{Q}\{S\} \Rightarrow \mathtt{Q}'$ schedula un'esecuzione che viene innescata quando l'orologio globale è equivalente al valore di $\mathtt{t}$. Quando innescato, S verrà eseguito se il contratto si trova sullo stato $\mathtt{Q}$. Al termine di S il contratto si troverà sullo stato $\mathtt{Q}'$.

Le *Expressions* E includono:

- valori costanti (numeri reali n, booleani, stringhe s);
- nomi di asset, campi e parametri indincati con X;
- operazioni unarie e binarie:
  - operazioni aritmetiche standard (`+`, `-`, `*`, `/`);
  - operazioni logiche (congiunzione `&&`, disgiunzione `||`, negazione `!`);
  - concatenazione di stringhe (`^`);
  - operazioni di relazione standard (`<`, `>`, `<=`, `>=`, `==`).

Si assume che una funzione è determinata univocamente dalla tupla `Q A.f Q'` formata dallo stato iniziale, dalla parte che invoca la funzione, dal nome della funzione e dallo stato finale. Analogamente, un evento è determinato univocamente dalla tupla `Q ev.n Q'` dove `n` è la linea di codice dell'evento. Si usa `H.c` per indicare indistintamente `A.f` o `ev.n` e si indica con il termine clausola la tupla `Q H.c Q'`.

Con un abuso di notazione si indica il codice del contratto usando il nome del contratto e si scrive `Q A.f Q'` $\in$ `C` se `@Q A : f`$(\overline{\texttt{y}})[\overline{\texttt{k}}]\{$`S W`$\} \Rightarrow$ `@Q'` è presente all'interno del contratto `C`. Si scrive anche `Q'' ev.n Q'''` $\in$ `Q A.f Q'` se `t` $\gg$ `@Q''`$\{S'\} \Rightarrow$ `@Q'''` è definito nella funzione `@Q A : f`$(\overline{\texttt{y}})[\overline{\texttt{k}}]\{$`S W`$\} \Rightarrow$ `@Q'`. Nell'ultimo caso si scrive anche `Q'' ev.n Q'''` $\in$ `C`.

Esempio Si consideri il seguente contratto:

```
1 stipula Bet {
2         assets wallet1, wallet2
3         fields val1, val2, amount = 10
4         init Init
5
6         @Init Better1: place_bet(x)[h](h == amount) {
7                 h -o wallet1
```

```
8                   x -> val1
9                   now + 60 >> @First {
10                          wallet1 -o Better1
11                  } => @Fail
12          } => @First
13
14          @First Better2: place_bet(x)[h](h == amount) {
15                  h -o wallet2
16                  x -> val2
17                  now + 60 >> @Run {
18                          wallet1 -o Better1
19                          wallet2 -o Better2
20                  } => @Fail
21          } => @Run
22
23          @Run Authority: declare_winner()[] {
24                  if(val1 >= val2) {
25                          wallet1 -o Better1
26                          wallet2 -o Better1
27                  } else {
28                          wallet1 -o Better2
29                          wallet2 -o Better2
30                  }
31          } => @End
32  }
```

Quando `Better1` fa la sua scommessa, lo stato passa da `Init` a `First` dove `Better1` non può impedire a `Better2` di fare la sua scommessa. La clausola per fare la scommessa è definita dalla funzione `place_bet`; si noti che la funzione raggruppa i parametri in due tipi di parentesi: parentesi tonde per i parametri che rappresentano campi, parentesi quadre per i parametri che rappresentano asset. Gli asset usano l'operatore ad hoc $\multimap$ per essere spostati da un posto all'altro. Per esempio, alla riga 10, `h` $\multimap$ `wallet1` significa che `h`

viene spostato in `wallet1`, il cui valore viene incrementato da quello di `h`, e `h` viene svuotato. Questo non è il caso dell'operazione `x` → `val1` nella riga 8 dove `x` non viene svuotato ma il valore di `val1` diventa quello di `x`.

È importante che il contratto definisca limiti temporali precisi per accettare i pagamenti e per proclamare il vincitore perché un certo numero di problemi potrebbero incombere: uno scommettitore potrebbe non depositare la quota o l'autorità potrebbe fallire nel comunicare il risultato, per esempio potrebbe essere un servizio online momentaneamente non raggiungibile. Questi limiti temporali sono definiti dalgi eventi. Per esempio, alla riga 9 si specifica che la scommessa fallisce se `Better2` non scommette entro 60 minuti dalla scommessa di `Better1`.

2.3 Semantica operazionale

La semantica operazionale di Stipula è definita tramite un sistema di transizioni. Sia $\mathtt{C}(\mathtt{Q}, \ell, \Sigma, \Psi)$ un contratto in esecuzione dove:

- $\mathtt{C}$ è il nome del contratto;
- $\mathtt{Q}$ è lo stato attuale del contratto;
- ℓ è una mappatura da campi, asset e parametri di funzioni a valori;
- Σ è il residuo, anche vuoto, del corpo di una funzione o di un evento;
- Ψ è un multinsieme, anche vuoto, di eventi, in attesa, che sono sono già stati schedulati per un'esecuzione futura ma non sono ancora stati innescati. In particolare Ψ può essere $\_$, quando non ci sono eventi in attesa, oppure $W_1 \mid ... \mid W_n$ dove $W_i = \mathbb{t}_i \gg_{\mathtt{n}_i} \mathtt{Q}_i\{S_i\} \Rightarrow \mathtt{Q}'_i$. Il tempo $\mathbb{t}_i$ è un tempo assoluto e si ottiene valutando l'espressione temporale $\mathtt{t}_i$ dell'evento dove `now` viene sostituito con il valore dell'orologio globale nel momento di generazione dell'evento. L'indice $\mathtt{n}_i$ è la linea di codice dove viene definito l'evento.

I contratti in esecuzione sono chiamati $\mathbb{C}, \mathbb{C}', ...$. Una configurazione è una coppia $\mathbb{C}, \mathbb{t}$ dove $\mathbb{t}$ è il valore del tempo dell'orologio globale del sistema. La relazione di transizione di Stipula è $\mathbb{C}, \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$ dove μ può essere:

- $\_$ (vuota)
- $\mathtt{A.f}(\overline{u})[\overline{a}]$
- `ev.n`
- $v \rightarrow \mathtt{A}$
- $a \multimap \mathtt{A}$

$$\text{[Function]}\ \frac{@\mathtt{Q}\ \mathtt{A} : \mathtt{f}(\overline{\mathtt{y}})[\overline{\mathtt{k}}]\{S\ W\} \Rightarrow @\mathtt{Q}' \in \mathtt{C} \qquad W' = \mathtt{LC}_{\mathtt{Q\ A.f\ Q'}}(W) \qquad \Psi, \mathbb{t} \nrightarrow \qquad \ell' = \ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{a}]}{\mathtt{C}(\mathtt{Q}, \ell, \_, \Psi), \mathbb{t} \xrightarrow{\mathtt{A.f}(\overline{u})[\overline{a}]} \mathtt{C}(\mathtt{Q}, \ell', S\ W' \Rightarrow \mathtt{Q}', \Psi), \mathbb{t}}$$

$$\text{[State-Change]}\ \frac{[\![W\{^{\mathbb{t}}/_{\mathtt{now}}\}]\!]_\ell = \Psi'}{\mathtt{C}(\mathtt{Q}, \ell, \_\ W \Rightarrow \mathtt{Q}', \Psi), \mathbb{t} \to \mathtt{C}(\mathtt{Q}', \ell, \_, \Psi' \mid \Psi), \mathbb{t}}$$

$$\text{[Event-Match]}\ \frac{\Psi = \mathbb{t} \gg_{\mathtt{n}} \mathtt{Q}\{S\} \Rightarrow \mathtt{Q}' \mid \Psi'}{\mathtt{C}(\mathtt{Q}, \ell, \_, \Psi), \mathbb{t} \xrightarrow{\mathtt{ev.n}} \mathtt{C}(\mathtt{Q}, \ell, S \Rightarrow \mathtt{Q}', \Psi'), \mathbb{t}}$$

$$\text{[Tick]}\ \frac{\Psi, \mathbb{t} \nrightarrow}{\mathtt{C}(\mathtt{Q}, \ell, \_, \Psi), \mathbb{t} \to \mathtt{C}(\mathtt{Q}, \ell, \_, \Psi), \mathbb{t} + 1}$$

$$\text{[Value-Send]}\ \frac{[\![E]\!]_\ell = v}{\mathtt{C}(\mathtt{Q}, \ell, E \to \mathtt{A}\ \Sigma, \Psi), \mathbb{t} \xrightarrow{v \to \mathtt{A}} \mathtt{C}(\mathtt{Q}, \ell, \Sigma, \Psi), \mathbb{t}}$$

$$\text{[Asset-Send]}\ \frac{[\![E]\!]^{\mathtt{a}}_\ell = a \qquad [\![\mathtt{h} - a]\!]^{\mathtt{a}}_\ell = a'}{\mathtt{C}(\mathtt{Q}, \ell, E \multimap \mathtt{h}, \mathtt{A}\ \Sigma, \Psi), \mathbb{t} \xrightarrow{a \multimap \mathtt{A}} \mathtt{C}(\mathtt{Q}, \ell[\mathtt{h} \mapsto a'], \Sigma, \Psi), \mathbb{t}}$$

$$\text{[Field-Update]}\ \frac{[\![E]\!]_\ell = v}{\mathtt{C}(\mathtt{Q}, \ell, E \to \mathtt{x}\ \Sigma, \Psi), \mathbb{t} \to \mathtt{C}(\mathtt{Q}, \ell[\mathtt{x} \mapsto v], \Sigma, \Psi), \mathbb{t}}$$

$$\text{[Asset-Update]}\ \frac{[\![E]\!]^{\mathtt{a}}_\ell = a \qquad [\![\mathtt{h} - a]\!]^{\mathtt{a}}_\ell = a' \qquad [\![\mathtt{h}' + a]\!]^{\mathtt{a}}_\ell = a'' \qquad \ell' = \ell[\mathtt{h} \mapsto a', \mathtt{h}' \mapsto a'']}{\mathtt{C}(\mathtt{Q}, \ell, E \multimap \mathtt{h}, \mathtt{h}'\ \Sigma, \Psi), \mathbb{t} \to \mathtt{C}(\mathtt{Q}, \ell', \Sigma, \Psi), \mathbb{t}}$$

$$\text{[Cond-True]}\ \frac{[\![E]\!]_\ell = \mathtt{true}}{\mathtt{C}(\mathtt{Q}, \ell, \mathtt{if}(E)\{S\}\mathtt{else}\{S'\}\ \Sigma, \Psi), \mathbb{t} \to \mathtt{C}(\mathtt{Q}, \ell, S\ \Sigma, \Psi), \mathbb{t}}$$

$$\text{[Cond-False]}\ \frac{[\![E]\!]_\ell = \mathtt{false}}{\mathtt{C}(\mathtt{Q}, \ell, \mathtt{if}(E)\{S\}\mathtt{else}\{S'\}\ \Sigma, \Psi), \mathbb{t} \to \mathtt{C}(\mathtt{Q}, \ell, S'\ \Sigma, \Psi), \mathbb{t}}$$

Tabella 2.2: Semantica operazionale di Stipula

La definizione formale di $\mathbb{C}, \mathbb{t} \xrightarrow{\mu} \mathbb{C}', \mathbb{t}'$ è riportata nella tabella (2.2) usando le seguenti funzioni e predicati ausiliari:

- $[\![E]\!]_\ell$ è una funzione parziale che restituisce il valore di E nella memoria ℓ. La definizione è standard eccetto quando E è un asset poiché $[\![\mathtt{a}]\!]_\ell$ è il valore grezzo dell'asset che è sempre un numero reale. La funzione

è parziale perché $[\![\mathtt{x}]\!]_\ell$ non è definita quando $\mathtt{x} \notin dom(\ell)$ oppure E contiene un'operazione parziale.

- $[\![E]\!]^{\mathtt{a}}_\ell$ è la funzione parziale che restituisce i valori degli asset. E è un'operazione sugli asset. $[\![E]\!]^{\mathtt{a}}_\ell$ non è definita quando $[\![E]\!]_\ell$ non è definita oppure $[\![E]\!]_\ell$ è negativa, poiché gli asset sono sempre non negativi.
- il predicato $\Psi, \mathbb{t} \nrightarrow$, dato $\Psi = \mathbb{t}_1 \gg_{\mathtt{n}_1} \mathtt{Q}_1\{S_1\} \Rightarrow \mathtt{Q}'_1 \mid ... \mid \mathbb{t}_k \gg_{\mathtt{n}_k} \mathtt{Q}_k\{S_k\} \Rightarrow \mathtt{Q}'_k$, è definito come segue:

$$\Psi, \mathbb{t} \nrightarrow = \begin{cases} true & se\ \mathbb{t}_i \neq \mathbb{t} \wedge 1 \leq i \leq k \\ false & altrimenti \end{cases}$$

Esempio Si consideri il seguente contratto:

```
1  stipula Sample1 {
2          fields x
3          init Init
4
5          @Init A: f()[] {
6                  now + 60 >> @Ready {} => @End
7          } => @Wait
8
9          @Wait B: g()[] {} => @Ready
10 }
```

`Sample1` è un contratto molto semplice, privo di operazioni su campi e asset. Si sottolinea che nella computazione riportata di seguito, tra i passaggi 3 e 64, non è importante l'ordine con cui si alternano l'invocazione della funzione `B.g` e lo scorrere del tempo. La cosa importante è che la funzione `B.g` venga eseguita prima che l'orologio globale si trovi a 61 (per la semantica operazione, se venisse invocata con l'orologio globale a 60, questo resterebbe invariato a fine invocazione). Una volta invocata `B.g` non resta che attendere lo scorrimento del tempo e l'esecuzione dell'evento generato dall'invocazione di `A.f`. Di seguito vengono applicate le regole della semantica operazionale:

0. $\texttt{C}(\texttt{Init}, \ell, \_, \_), 0$
1. [Function] $\xrightarrow{\texttt{A.f()[]}} \texttt{C}(\texttt{Init}, \ell, \_\ W \Rightarrow @\texttt{Wait}, \_), 0$ dove $W = \texttt{now} + 60 \gg @\texttt{Ready}\{\} \Rightarrow @\texttt{End}$
2. [State-Change] $\texttt{C}(\texttt{Wait}, \ell, \_, \Psi'), 0$ dove $\Psi' = 60 \gg @\texttt{Ready}\{\} \Rightarrow @\texttt{End}$
3. [Tick] $\texttt{C}(\texttt{Wait}, \ell, \_, \Psi'), 1$
4. [Function] $\xrightarrow{\texttt{B.g()[]}} \texttt{C}(\texttt{Wait}, \ell, \_\ \_ \Rightarrow @\texttt{Ready}, \Psi'), 1$
5. [State-Change] $\texttt{C}(\texttt{Ready}, \ell, \_, \Psi'), 1$
6. [Tick] $\texttt{C}(\texttt{Ready}, \ell, \_, \Psi'), 2$

...

64. [Tick] $\texttt{C}(\texttt{Ready}, \ell, \_, \Psi'), 60$
65. [Event-Match] $\xrightarrow{\texttt{ev.6}} \texttt{C}(\texttt{Ready}, \ell, \_ \Rightarrow @\texttt{End}, \_), 60$
66. [State-Change] $\texttt{C}(\texttt{End}, \ell, \_, \_), 60$

Capitolo 3

Raggiungibilità

3.1 Introduzione

Durante l'esecuzione di un contratto vengono eseguite sequenze di clausole. Queste sequenze di clausole vengono chiamate **computazioni** e rappresentano esecuzioni reali delle clausole di un contratto. La sequenza di clausole di una computazione può essere contenuta all'interno della sequenza di clausole di un'altra computazione. Il termine **raggiungibilità**, riferito a una clausola, rappresenta la possibilità di trovare quella clausola all'interno di una computazione che parte dallo stato iniziale. Formalmente, il concetto di raggiungibilità viene espresso dalla seguente definizione.

Definizione 1. *Sia* `C` *un contratto scritto in Stipula con configurazione iniziale* $\mathbb{C}, \mathbb{t}$ *e sia* $\mathbb{C}, \mathbb{t} \rightarrow^* \xrightarrow{\mu} \mathtt{C}(\mathtt{Q}, \ell, S\ W \Rightarrow \mathtt{Q}', \Psi'), \mathbb{t}'$. *Se* $\mu = \mathtt{A.f}(\overline{u})[\overline{a}]$ *allora la funzione* `Q A.f Q'` $\in$ `C` *è* ***raggiungibile****. Se* $\mu =$ `ev.n` *allora l'evento* `Q ev.n Q'` $\in$ `C` *è* ***raggiungibile****.*

Esempio Si consideri il seguente contratto:

```
1        stipula Sample2 {
2                fields x
3                init Init
```

```
4
5                   @Init A: f()[] {
6                           now + 60 >> @Closing {} => @End
7                   } => @Wait
8
9                   @Wait B: g()[] {
10                          now + 30 >> @Ready {} => @Closing
11                  } => @Ready
12          }
```

Facendo una prima analisi informale, è chiaro che l'esecuzione dell'evento genereato dall'invocazione di `A.f` non è certa. Questo accade perché dopo aver invocato `A.f` e aver generato l'evento `ev.5`, è possibile aspettare un tempo indefinito prima di invocare `B.g`. Tuttavia, anche se non è certo che `ev.5` venga eseguito, risulta comunque possibile eseguirlo. Applicando la definizione (1), viene riportata una possibile computazione che rende raggiungibili tutte le clausole del contratto:

$$
\begin{aligned}
&\mathtt{C}(\mathtt{Init}, \ell, \_, \_), 0 \\
&\quad \xrightarrow{\mathtt{A.f()[]}} \rightarrow^* \mathtt{C}(\mathtt{Wait}, \ell, \_, \Psi_1), 0 \\
&\quad \xrightarrow{\mathtt{B.g()[]}} \rightarrow^* \mathtt{C}(\mathtt{Ready}, \ell, \_, \Psi_2 \mid \Psi_1), 30 \\
&\quad \xrightarrow{\mathtt{ev.9}} \rightarrow^* \mathtt{C}(\mathtt{Closing}, \ell, \_, \Psi_1), 60 \\
&\quad \xrightarrow{\mathtt{ev.5}} \rightarrow^* \mathtt{C}(\mathtt{End}, \ell, \_, \_), 60
\end{aligned}
$$

dove

- $\Psi_1 = 30 \gg @\mathtt{Ready}\{\} \Rightarrow @\mathtt{Closing}$
- $\Psi_2 = 60 \gg @\mathtt{Closing}\{\} \Rightarrow @\mathtt{End}$

Rispettando la semantica operazionale di Stipula, le transizioni sono state riportate come $\xrightarrow{\mu}\rightarrow^*$ per sottolineare il fatto che dopo l'esecuzione delle regole [Function] e [Event-Match] avvengono le regole [State-Change] e [Tick], le quali non sono state riportate per rendere più snello l'esempio.

Un contratto scritto in Stipula è assimilabile a un automa a stati finiti che, in presenza di cicli, può generare un numero infinito di computazioni di lunghezza finita. Per risolvere questo problema si utilizzano sequenze di clausole senza ripetizioni chiamate **computazioni astratte**. Nelle sezioni successive, le computazioni astratte vengono spesso indicate con i simboli $\mathbb{A}, \mathbb{A}', \ldots$. Ogni computazione è rappresentabile tramite una computazione astratta che contiene le clausole eseguite durante la computazione. Possono esistere computazioni astratte che non rappresentano nessuna computazione poiché queste computazioni astratte non rispettano i vincoli definiti dalla semantica operazionale oppure generano errori in fase di esecuzione. Un contratto scritto in Stipula è rappresentabile da un numero finito di computazioni astratte.

Il concetto di computazione astratta è abbastanza grezzo e risulta necessario sottolineare alcuni aspetti significativi:

- una computazione astratta è semplicemente una sequenza di clausole, senza vincoli, che potrebbe non rappresentare nessuna computazione. Per considerare solamente le computazioni astratte che rappresentano computazioni è necessario integrare, nelle definizioni delle funzioni che si implementano, le regole espresse dalla semantica operazionale di Stipula.

- essendo una sequenza senza ripetizioni, una computazione astratta non è in grado di catturare esplicitamente i cicli presenti nella computazione. Per questo motivo una computazione astratta può rappresentare anche un numero infinito di computazioni.

Anche se può sembrare una perdita di espressività, l'utilizzo delle computazioni astratte permette di definire le funzioni riportate nelle sezioni successive su un domionio sempre finito.

3.2 Caratteristiche notevoli

Nella definizione (1) viene definita la raggiungibilità in termini di esecuzione di un contratto, ovvero in termini di computazioni. Nel nostro caso siamo interessati alla caratteristica opposta, ovverso siamo interessati a determinare quando una clausola sia irraggiungibile. In questo caso la definizione (1) non è d'aiuto in quanto in caso di cicli la ricerca diverge. Nello specifico dovremmo verificare che non esiste una computazione che contiene la clausola in esame ma in caso di cicli è presente un numero infinito di computazioni. Come detto in precedenza, l'utilizzo delle computazioni astratte permette di rendere finito il dominio della ricerca ma resta comunque necessario analizzare quali siano le caratteristiche che possono determinare la non raggiungibilità di una clausola. Queste caratteristiche sono:

- **raggiungibilità topologica**: le clausole fungono da collegamento tra gli stati dei contratti quindi un contratto scritto in Stipula può essere visto come un grafo dove i nodi sono gli stati e gli archi sono le clausole. Si sottolinea che gli archi del grafo sono direzionati dallo stato iniziale della clausola verso quello finale. È presente una differenza tra le funzioni e gli eventi:
  - una funzione è sicuramente non raggiungibile quando non esiste un percorso sul grafo che partendo dallo stato iniziale del contratto raggiunga lo stato di inizio della funzione;
  - un evento è sicuramente non raggiungibile quando non esiste un percorso sul grafo che partendo dallo stato finale della funzione che genera l'evento raggiunga lo stato di inizio dell'evento oppure quando la funzione che definisce l'evento è irraggiungibile.
- **raggiungibilità temporale**: gli eventi vengono eseguiti se lo stato attuale del contratto in fase di esecuzione è lo stesso di inizio degli eventi quando l'orologio globale equivale al loro tempo di innesco. Per questo motivo un evento è sicuramente non eseguibile se tutte le computazio-

ni che raggiungono lo stato iniziale dell'evento lo fanno ad un tempo maggiore rispetto al tempo di innesco dell'evento.

Utilizzando le computazioni astratte non è possibile catturare esplicitamente i cicli. Se una funzione è eseguibile ciclicamente allora può generare altrettanto ciclicamente gli eventi definiti al suo interno. In queste condizioni l'analisi dei tempi di esecuzione delle clausole risulterà complesso quindi, nel caso di eventi generabili ciclicmanete, si è deciso di cosiderare solamente la raggiungibilità topologica. Si sottolinea che, anche se in fase di identificazione delle clausole irraggiungibili potrebbero essere presenti dei falsi negativi, lo scopo è quello di identificare come irraggiungibili clausole sicuramente irraggiungibili, ammettendo falsi negativi ma non ammettendo falsi positivi.

3.3 Teoria della raggiungibilità

Si introducono le seguenti operazioni ausiliari e notazioni:

- $\mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \in \mathbb{A}$ se nella sequenza $\mathbb{A}$ è presente un elemento uguale a $\mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}'$;

- sia $\mathcal{A} = \{\mathtt{Q}_1\ \mathtt{H}_1.\mathtt{c}_1\ \mathtt{Q}'_1, ..., \mathtt{Q}_n\ \mathtt{H}_n.\mathtt{c}_n\ \mathtt{Q}'_n\}$. $\mathbb{A}|_{\mathcal{A}}$ è l'insieme definito come segue:

$$\mathbb{A}|_{\mathcal{A}} \stackrel{\mathtt{def}}{=} \begin{cases} \varnothing & se\ \mathbb{A} = \varepsilon \\ \mathbb{A}'|_{\mathcal{A}} & se\ \mathbb{A} = \mathbb{A}'; \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \wedge \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \notin \mathcal{A} \\ \mathbb{A}'|_{\mathcal{A}} \cup \{\mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}'\} & se\ \mathbb{A} = \mathbb{A}'; \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \wedge \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \in \mathcal{A} \end{cases}$$

- si definisce:

$$\mathbb{A} \lhd \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \stackrel{\mathtt{def}}{=} \begin{cases} \mathbb{A}; \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' & se\ \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \notin \mathbb{A} \\ \mathbb{A} & se\ \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \in \mathbb{A} \end{cases}$$

  Si definisce anche:

$$\{\mathbb{A}_1, ..., \mathbb{A}_n\} \lhd \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}' \stackrel{\mathtt{def}}{=} \{\mathbb{A}_1 \lhd \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}', ..., \mathbb{A}_n \lhd \mathtt{Q}\ \mathtt{H.c}\ \mathtt{Q}'\}$$

- siano $\mathbb{R}'$ e $\mathbb{R}''$ due mappature da clausole a insiemi di computazioni astratte. Si definisce $\mathbb{R}' \leq \mathbb{R}''$ se per ogni $\mathtt{Q}$, $\mathbb{R}'(\mathtt{Q}) \subseteq \mathbb{R}''(\mathtt{Q})$. Il dominio delle mappature con l'ordinamento parziale $\leq$ è un reticolo [10] e dato un contratto questo reticolo è sempre finito.

Nelle successive definizioni, con lo scopo di ottenere una notazione più snella, si ometterà sempre il riferimento al contratto $\mathtt{C}$.

Come detto in precedenza, per efettuare l'analisi della raggiungibilità è necessario conoscere quali funzioni sono eseguibili ciclicamente. Quindi viene fatta una prima analisi del contratto analizzando la raggiungibilità topologica allo scopo di poter definire successivamente una funzione in grado di discriminare le funzioni eseguibili ciclicamente.

Definizione 2. *Sia* $\mathtt{C}$ *un contratto scritto in Stipula con stato iniziale* $\mathtt{Q}$*. La sequenza di mappature* $\mathbb{R}_{\mathtt{Q}}{}^{(0)}, \mathbb{R}_{\mathtt{Q}}{}^{(1)}, ...,$ *che mappano clausole in* $\mathtt{C}$ *su insiemi di computazioni astratte, è definita come segue:*

1. $$\mathbb{R}_{\mathtt{Q}}{}^{(0)}(\mathtt{Q}_1\ \mathtt{H.c}\ \mathtt{Q}_2) = \begin{cases} \{\mathtt{Q}_1\ \mathtt{H.c}\ \mathtt{Q}_2\} & \textit{se } \mathtt{Q} = \mathtt{Q}_1 \wedge \mathtt{H.c} = \mathtt{A.f} \\ & \quad \wedge \mathtt{Q}\ \mathtt{A.f}\ \mathtt{Q}_2 \in \mathtt{C} \\ \varnothing & \textit{altrimenti} \end{cases}$$

2. $$\begin{aligned} &\mathbb{R}_{\mathtt{Q}}{}^{(i+1)}(\mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2) = \\ &\quad \mathbb{R}_{\mathtt{Q}}{}^{(i)}(\mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2) \\ &\quad \cup \left(\bigcup_{\mathtt{Q}_3\ \mathtt{H'.c'}\ \mathtt{Q}_1 \in \mathtt{C}} \{\mathbb{A} \triangleleft \mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2 \mid \mathbb{A} \in \mathbb{R}_{\mathtt{Q}}{}^{(i)}(\mathtt{Q}_3\ \mathtt{H'.c'}\ \mathtt{Q}_1)\}\right) \end{aligned}$$

3. $$\begin{aligned} &\textit{se } \mathtt{Q}_1\ \mathtt{ev.n}\ \mathtt{Q}_2 \in \mathtt{Q'}\ \mathtt{A.f}\ \mathtt{Q''} \textit{ allora} \\ &\mathbb{R}_{\mathtt{Q}}{}^{(i+1)}(\mathtt{Q}_1\ \mathtt{ev.n}\ \mathtt{Q}_2) = \\ &\quad \mathbb{R}_{\mathtt{Q}}{}^{(i)}(\mathtt{Q}_1\ \mathtt{ev.n}\ \mathtt{Q}_2) \\ &\quad \cup \left(\bigcup_{\mathtt{Q}_3\ \mathtt{H'.c'}\ \mathtt{Q}_1 \in \mathtt{C}} \left\{ \mathbb{A} \triangleleft \mathtt{Q}_1\ \mathtt{ev.n}\ \mathtt{Q}_2 \;\middle|\; \begin{array}{r} \mathbb{A} \in \mathbb{R}_{\mathtt{Q}}{}^{(i)}(\mathtt{Q}_3\ \mathtt{H'.c'}\ \mathtt{Q}_1) \\ \wedge \mathtt{Q'}\ \mathtt{A.f}\ \mathtt{Q''} \in \mathbb{A} \end{array} \right\}\right) \end{aligned}$$

Per definizione $\mathbb{R}_{\mathtt{Q}}{}^{(i)} \leq \mathbb{R}_{\mathtt{Q}}{}^{(i+1)}$*. Poiché l'insieme delle possibili mappature da clausole a insiemi di computazioni astratte è un reticolo finito, esiste* κ *tale che* $\mathbb{R}_{\mathtt{Q}}{}^{(\kappa)} \leq \mathbb{R}_{\mathtt{Q}}{}^{(\kappa+1)}$ *[10]. Sia* $\mathbb{R}_{\mathtt{Q}}$ *il punto fisso.*

Come detto in precedenza, la definizione (2) costruisce le computazioni astratte e gli insiemi di computazioni astratte considerando anche le caratteristiche del linguaggio Stipula. In particolare al punto 1 vengono considerate solamente le funzioni il cui stato iniziale coincide con lo stato iniziale del contratto e, per questo, sicuramente raggiungibili. Con i punti 2 e 3 si considera la raggiungibilità topologica rispettivamente di funzioni ed eventi.

A questo punto è possibile considerare irraggiungibili tutte le clausole $\mathtt{Q'}\ \mathtt{H.c}\ \mathtt{Q''}$ per le quali $\mathbb{R}_{\mathtt{Q}}(\mathtt{Q'}\ \mathtt{H.c}\ \mathtt{Q''}) = \varnothing$. Per costruzione, se ad una clausola non sono associate computazioni astratte i motivi possono essere:

- nel caso delle funzioni, nessuna clausola raggiunge la funzione dallo stato iniziale;

- nel caso degli eventi, nessuna computazione permette di eseguire l'evento dopo aver eseguito la funzione che lo genera.

Esempio Si consideri il seguente contratto:

```
1  stipula Sample3 {
2          fields x
3          init Init
4
5          @Init A: f()[] {} => @First
6
7          @Init B: g()[] {} => @First
8
9          @First C: h()[] {} => @Init
10 }
```

Il contratto `Sample3` presenta un ciclo. Invocando `C.h` è possibile tornare allo stato `Init` dopo aver precedentemente invocato `A.f` oppure `B.g`. Si costruiscono gl insiemi di computazioni astratte secondo quanto definito nella definizione (2):

0.
$$\mathbb{R}_{\texttt{Init}}{}^{(0)}(\texttt{Init A.f First}) = \{\texttt{Init A.f First}\}$$
$$\mathbb{R}_{\texttt{Init}}{}^{(0)}(\texttt{Init B.g First}) = \{\texttt{Init B.g First}\}$$
$$\mathbb{R}_{\texttt{Init}}{}^{(0)}(\texttt{First C.h Init}) = \varnothing$$

1.
$$\mathbb{R}_{\texttt{Init}}{}^{(1)}(\texttt{Init A.f First}) = \{\texttt{Init A.f First}\}$$
$$\mathbb{R}_{\texttt{Init}}{}^{(1)}(\texttt{Init B.g First}) = \{\texttt{Init B.g First}\}$$
$$\mathbb{R}_{\texttt{Init}}{}^{(1)}(\texttt{First C.h Init}) = \{\texttt{Init A.f First};\texttt{First C.h Init}$$
$$\texttt{Init B.g First};\texttt{First C.h Init}\}$$

2.
$$\begin{aligned}
&\mathbb{R}_{\texttt{Init}}{}^{(2)}(\texttt{Init A.f First}) = \{\texttt{Init A.f First}\\
&\quad \texttt{Init A.f First};\texttt{First C.h Init}\\
&\quad \texttt{Init B.g First};\texttt{First C.h Init};\texttt{Init A.f First}\}\\
&\mathbb{R}_{\texttt{Init}}{}^{(2)}(\texttt{Init B.g First}) = \{\texttt{Init B.g First}\\
&\quad \texttt{Init B.g First};\texttt{First C.h Init}\\
&\quad \texttt{Init A.f First};\texttt{First C.h Init};\texttt{Init B.g First}\}\\
&\mathbb{R}_{\texttt{Init}}{}^{(2)}(\texttt{First C.h Init}) = \{\texttt{Init A.f First};\texttt{First C.h Init}\\
&\quad \texttt{Init B.g First};\texttt{First C.h Init}\}
\end{aligned}$$

3.
$$\begin{aligned}
&\mathbb{R}_{\texttt{Init}}{}^{(3)}(\texttt{Init A.f First}) = \{\texttt{Init A.f First}\\
&\quad \texttt{Init A.f First};\texttt{First C.h Init}\\
&\quad \texttt{Init B.g First};\texttt{First C.h Init};\texttt{Init A.f First}\}\\
&\mathbb{R}_{\texttt{Init}}{}^{(3)}(\texttt{Init B.g First}) = \{\texttt{Init B.g First}\\
&\quad \texttt{Init B.g First};\texttt{First C.h Init}\\
&\quad \texttt{Init A.f First};\texttt{First C.h Init};\texttt{Init B.g First}\}\\
&\mathbb{R}_{\texttt{Init}}{}^{(3)}(\texttt{First C.h Init}) = \{\texttt{Init A.f First};\texttt{First C.h Init}\\
&\quad \texttt{Init B.g First};\texttt{First C.h Init}\\
&\quad \texttt{Init A.f First};\texttt{First C.h Init};\texttt{Init B.g First}\\
&\quad \texttt{Init B.g First};\texttt{First C.h Init};\texttt{Init A.f First}\}
\end{aligned}$$

Viene omesso il passo 4 in quanto sarebbe equivalente al passo 3, di conseguenza è stato trovato il punto fisso.

Definizione 3. *Sia* $\texttt{C}$ *un contratto scritto in Stipula con configurazione iniziale* $\texttt{C}(\texttt{Q}, \ell, \_, \_), \mathbb{t}$. *La computazione astratta corrispondente a* $\texttt{C}(\texttt{Q}, \ell, \_, \_), \mathbb{t} \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_n} \texttt{C}(\texttt{Q}', \ell', \_, \Psi), \mathbb{t}'$ *è la sequenza di clausole* $\texttt{Q}_1\ \texttt{H}_1.\texttt{c}_1\ \texttt{Q}'_1; \ldots; \texttt{Q}_k\ \texttt{H}_k.\texttt{c}_k\ \texttt{Q}'_k$ *tale che:*

- *se* $\texttt{Q}_j\ \texttt{H}_j.\texttt{c}_j\ \texttt{Q}'_j$ *è la clausola corrispondente a* μ_i *allora non esiste un altro* μ'_i *con* $i' < i$ *a cui corrisponde la stessa clausola;*
- *per ogni* $\texttt{Q}_{j'}\ \texttt{H}_{j'}.\texttt{c}_{j'}\ \texttt{Q}'_{j'}$ *con* $j' < j$, *esiste* $\mu_{i'}$ *con* $i' < i$ *tale che* $\texttt{Q}_{j'}\ \texttt{H}_{j'}.\texttt{c}_{j'}\ \texttt{Q}'_{j'}$ *è la clausola corrispondente a* $\mu_{i'}$.

Esempio Si faccia riferimento al contratto `Sample3` presente nell'esempio precedente e alla mappatura $\mathbb{R}_{\texttt{Init}}$ che è stata calcolata. Si consideri la computazione $\texttt{C}(\texttt{Init}, \ell, \_, \_), \mathbb{t} \xrightarrow{\texttt{A.f}} \texttt{C}(\texttt{First}, \ell, \_, \_), \mathbb{t}' \xrightarrow{\texttt{C.h}} \texttt{C}(\texttt{Init}, \ell, \_, \_), \mathbb{t}'' \xrightarrow{\texttt{B.g}} \texttt{C}(\texttt{First}, \ell, \_, \_), \mathbb{t}''' \xrightarrow{\texttt{C.h}} \texttt{C}(\texttt{Init}, \ell, \_, \_), \mathbb{t}'''' \rightarrow^*$. Anche se tutte le computazioni sono finite, la computazione riportata in questo esempio può fungere da nucleo per la generazione di infinite computazioni di lunghezza finita; questo aspetto viene indicato con il simbolo $\rightarrow^*$. Come detto in precedenza, le computazioni astratte sono sempre finite e sono in grado di catturare tutte le computazioni. Si sta considerando una computazione che come ultima clausola invoca `First C.h Init`, quindi la computazione astratta corrispondente dovrà essere presente all'interno di $\mathbb{R}_{\texttt{Init}}(\texttt{First C.h Init})$. In effetti la computazione astratta corrispondente risulta essere `Init A.f First`; `First C.h Init`; `Init B.g First`. Si consideri la definizione (3). Il primo punto della definizione indica che ogni clausola della computazione astratta è presente soltanto una volta all'interno della sequenza. In questo caso le tre clausole della sequenza sono differenti, quindi il primo punto è rispettato, come anche da definizione (2). Il secondo punto indica che le clausole della computazione astratta rispettano l'ordine di prima apparizione nella computazione alla quale corrisponde la computazione astratta. In questo caso è banale notare che l'ordine nella computazione astratta corrisponde all'ordine delle prime tre clausole della computazione. È importante ricordare che la computazione usata come esempio può ripetersi n volte, quindi la quarta e la quinta clausola invocate, all'interno della sequenza generale, sarebbero `First H.c Init` e `Init A.f First`. Tuttavia, essendo già presenti perché già comparse nella computazione, le due clausole non verranno nuovamente aggiunte e rispettano già l'ordine di prima apparizione.

A questo punto è possibile stabilire quando una funzione sia eseguibile ciclicamente.

Definizione 4. *Sia* `C` *un contratto scritto in Stipula con stato iniziale* `Q`*. Si*

definisce:

$$Cyclic_{\mathtt{Q}}(\mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2) = \begin{cases} & se\ \mathbb{A}; \mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2; \mathbb{A}' \in \mathbb{R}_{\mathtt{Q}}(\mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2) \\ true & \wedge \mathtt{Q}'_1\ \mathtt{H.c}\ \mathtt{Q}''_1 \in \mathbb{A}; \mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2 \\ & \wedge \mathtt{Q}'_2\ \mathtt{H}'\mathtt{.c}'\ \mathtt{Q}''_2 \in \mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2; \mathbb{A}' \\ & \wedge \mathtt{Q}''_2 = \mathtt{Q}'_1 \\ false & altrimenti \end{cases}$$

($Cyclic_{\mathtt{Q}}$ non è definita sugli eventi)

$Cyclic_{\mathtt{Q}}$ sfrutta la proprietà delle computazioni astratte per cui le clausole sono aggiunte a destra della sequenza. Quindi, presa la funzione in analisi come punto di riferimento, è sufficiente cercare una computazione astratta per cui da una delle clausole successive alla funzione sia possibile ritornare ad una delle clausole precedenti alla funzione. Questo fa si che la funzione sia eseguibile ciclicamente. In base all'utilizzo che la definizione fa di $\mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2$, si identificano correttamente anche le funzioni che ciclano su sé stesse.

Esempio Si consideri il precedente contratto `Sample2`. Si vuole valutare se la funzione `Init A.f First` sia ciclica. La mappatura calcolata per il contratto `Sample2` contiene il seguente insieme di computazioni astratte:

$$\begin{aligned} \mathbb{R}_{\mathtt{Init}}{}^{(3)}(\mathtt{Init\ A.f\ First}) = \{ & \mathtt{Init\ A.f\ First} \\ & \mathtt{Init\ A.f\ First; First\ C.h\ Init} \\ & \mathtt{Init\ B.g\ First; First\ C.h\ Init; Init\ A.f\ First}\} \end{aligned}$$

Analizzando la computazione astratta `Init A.f First; First C.h Init` tramite la definizione (4) si ottiene:

- $\mathbb{A}; \mathtt{Init\ A.f\ First}; \mathbb{A}' = \mathtt{Init\ A.f\ First; First\ C.h\ Init}$ ($\mathbb{A}$ non è presente);
- `Init A.f First` $\in$ `Init A.f First` (la parte destra è intesa come computazione astratta);
- `First C.h Init` $\in$ `Init A.f First; First C.h Init`;

- $\texttt{Init} = \texttt{Init}$.

Quindi `Init A.f First` è una funzione eseguibile ciclicamente.

3.3.1 Calcolo dei tempi logici

Ad ogni computazione astratta viene associato un **tempo logico** che rappresenta il tempo che la computazione astratta richiede per essere eseguita. Un tempo logico è un termine nella forma $\zeta_1 + ... + \zeta_n + a_1\phi_1 + ... + a_n\phi_n + k$ dove:

- ζ_i rappresentano i tempi d'esecuzione delle funzioni presenta nella computazione astratta;
- $a_i\phi_i$ rappresentano i termini derivanti dall'analisi delle espressioni temporali degli eventi;
- k rappresenta la parte costante.

Precedentemente si è fatto riferimento alla condizione per cui le funzioni eseguibili ciclicamente non permettono di definire con precisione la rggiungibilità degli eventi che generano. Tuttavia, indipendentemente dalla ciclicità, i tempo logici devono essere calcolati correttamente fornendo termini attendibili.

Si definiscono le seguenti operazioni e funzioni ausiliari:

- $\texttt{EV}_\texttt{C}(\texttt{Q A.f Q}') \stackrel{\texttt{def}}{=} \{\texttt{Q}_1\ \texttt{ev.n}\ \texttt{Q}_2 \mid \texttt{Q}_1\ \texttt{ev.n}\ \texttt{Q}_2 \in \texttt{Q A.f Q}'\}$
- $\texttt{TE}_\texttt{C}\left(\bigcup_{i\in 1..m}\{\texttt{Q}_i\ \texttt{ev.n}_i\ \texttt{Q}'_i\}\right) \stackrel{\texttt{def}}{=} \left\{ \texttt{t}_i\{{}^{0}/_{\texttt{now}}\} \;\middle|\; \begin{array}{l} i \in 1..m \wedge \texttt{Q}_i\ \texttt{ev.n}_i\ \texttt{Q}'_i \\ = \texttt{t}_i \gg_{\texttt{n}_i} @\texttt{Q}_i\{S_i\} \Rightarrow @\texttt{Q}'_i \end{array} \right\}$

Definizione 5. *Sia* `C` *un contratto scritto in Stipula con stato iniziale* $\texttt{Q}_0$*; in aggiunta, sia* ν *una mappatura iniettiva da funzioni* $\texttt{Q A.f Q}' \in \texttt{C}$ *a variabili. Siano*

- Θ_ν *la seguente funzione da computazioni astratte a insiemi di tempi logici:*

$$\Theta_\nu(\mathbb{A}) \stackrel{\text{def}}{=} \begin{cases} \{\Theta_\nu(\mathbb{A}') + \nu(\texttt{Q A.f Q}')\} & se\ \mathbb{A} = \mathbb{A}'; \texttt{Q A.f Q}' \\ \bigcup_{t \in T}(\Theta_\nu(\mathbb{A}') + \nu(\texttt{Q A. Q}') + t) & se\ \mathbb{A} = \mathbb{A}'; \texttt{Q A.f Q}'; \mathbb{A}''; \texttt{Q}_1\ \texttt{ev.n Q}_2 \\ & \wedge \texttt{Q}_1\ \texttt{ev.n Q}_2 \in \texttt{Q A.f Q}' \\ & \wedge T = \texttt{TE}_\texttt{C}(\mathbb{A}''; \texttt{Q}_1\ \texttt{ev.n Q}_2)|_{\texttt{EV}_\texttt{C}(\texttt{Q A.f Q}')} \end{cases}$$

- $\mathbb{T}_\nu$, *chiamata funzione tempo, la seguente funzione da insiemi di computazioni astratte a insiemi di tempi logici:*

$$\mathbb{T}_\nu(\{\mathbb{A}_1, ..., \mathbb{A}_n\}) \stackrel{\text{def}}{=} \bigcup_{i \in 1..n} \Theta_\nu(\mathbb{A}_i)$$

La precedente definizione si basa sul fatto che se durante una computazione incontriamo prima una funzione e poi un evento che è stato generato dalla funzione, tutte le clausole eseguite dopo la funzione e prima dell'evento non concorrono al calcolo in quanto il tempo di esecuzione dell'evento viene definito dalla sua espressione temporale e se l'evento è presente nella computazione allora è stato eseguito necessariamente nel suo tempo di esecuzione. Questo è vero anche in caso di eventi generabili ciclicamente. Tuttavia, quando si utilizzano le computazioni astratte, non è possibile rappresentare esplicitamente i cicli della computazione associata. In caso di cicli, diventa impossibile scartare arbitratiamente le clausole presenti tra la funzione e l'evento generato. Infatti, se tra le clausole che sarebbero state scartate sono presenti eventi della funzione, è necessario generare altrettanti tempi logici con lo scopo di catturare l'incertezza causata dalla presenza dei cicli.

Esempio Si consideri il precedente contratto `Sample2`. Per l'evento `Closing ev.5 End` si ha a disposizione:

$$\mathbb{R}_\texttt{Q}(\texttt{Closing ev.5 End}) = \{\texttt{Init A.f Wait}; \texttt{Wait B.g Ready}; \\ \texttt{Ready ev.9 Closing}; \texttt{Closing ev.5 End}\}$$

Di seguito si calcola il tempo logico tramite la definizione (5):

$$\mathbb{A} = \texttt{Init A.f Wait}; \texttt{Wait B.g Ready}; \texttt{Ready ev.9 Closing}; \texttt{Closing ev.5 End}$$

$$\mathbb{T}_\nu(\{\mathbb{A}\}) = \Theta_\nu(\mathbb{A})$$

$$\Theta_\nu = \{\zeta_{\texttt{A.f}} + 60\}$$

Nell'esempio è stato riportato un caso estremo in quanto l'evento che compare a destra della computazione astratta viene generato dalla funzione che compare a sinistra della computazione astratta. Per la definizione (5) questa condizione esclude dal calcolo tutte le clausole presenti tra l'evento e la funzione. Tuttavia, analizzando il risultato si può notare che il tempo logico associato all'evento corrisponde esattamente alla sua espressione temporale. In effetti, per la definizione (2), l'evento è l'ultima cosa che è stata eseguita nella computazione associata alla computazione logica. Poiché l'evento dipende dalla sua espressione temporale, averlo eseguito significa che attualmente l'orologio globale si trova al suo tempo di innesco. Quindi il risultato del calcolo del tempo logico è corretto.

3.3.2 Rimozione degli eventi scaduti

Ogni clausola è raggiungibile tramite diverse computazioni astratte. Ogni computazione astratta genera insiemi di tempi logici. Ogni tempo logico rappresenta un possibile tempo necessario a raggiungere la clausola. Questa considerazione va integrata nella precedente definizione (2) per ottenere una mappatura più raffinata che possa considerare anche i tempi di esecuzione delle clausole.

Definizione 6. *Siano T e T' due insiemi di tempi logici. Si dice che $T \leq T'$ è* ***risolvibile*** *se esistono $t \in T$, $t' \in T'$ e una sostituzione σ tali che $t\sigma \leq t'\sigma$ (σ mappa variabili su numeri naturali). Altrimenti si dice che $T \leq T'$ è* ***irrisolvibile****.*

Grazie alla definizione (6) viene stabilita la regola per poter confrontare gli insiemi di tempi logici appartenenti a due clausole. Infatti, ad eccezione delle funzioni che iniziano nello stato iniziale del contratto, per due computazioni subito consecutive, almeno un tempo logico associato alla prima deve essere minore o uguale ad almeno un tempo logico associato alla seconda. Questo

significa che esiste una computazione dove la prima clausola viene subito prima della seconda.

Esempio Si consideri il precedente contratto `Sample2`. Si hanno a disposizione i seguenti insiemi di tempi logici:

- per `Ready ev.9 Closing`: $\{\zeta_{\texttt{A.f}} + \zeta_{\texttt{B.g}} + 30\}$;
- per `Closing ev.5 End`: $\{\zeta_{\texttt{A.f}} + 60\}$.

I due insiemi $\mathbb{T}_\nu$ sono composti solamente da un elemento ciascuno, quindi l'analisi è banale. Secondo la definizione (6) $\{\zeta_{\texttt{A.f}} + \zeta_{\texttt{B.g}} + 30\} \leq \{\zeta_{\texttt{A.f}} + 60\}$ è risolvibile in quanto $\zeta_{\texttt{A.f}} + \zeta_{\texttt{B.g}} + 30 \leq \zeta_{\texttt{A.f}} + 60$ quando $\zeta_{\texttt{B.g}} \leq 30$. Si faccia attenzione al fatto che mentre la risolvibilità permetterò di classificare come raggiungibile una clausola, le relazioni prodotte dalla definizione (6) produrranno vincoli, da rispettare a runtine, per fare in modo che la raggiungibilità sia effettivamente soddisfatta.

Quanto è stato detto è riferito solamente ai tempi d'esecuzione. Non bisogna dimenticare tutti i restanti vincoli derivanti dalla semantica operazionale di Stipula. Si definisce quindi la mappatura raffinata che mette insieme tutta la teoria della raggiungibilità.

Definizione 7. *Sia* `C` *un contratto scritto in Stipula con stato iniziale* `Q` *e stato iniziale della memoria* ℓ*. La sequenza di mappature* $\mathbb{R}^{+(0)}_{\texttt{Q}}, \mathbb{R}^{+(1)}_{\texttt{Q}}, \ldots$ *è definita come segue:*

1. $$\mathbb{R}^{+(0)}_{\texttt{Q}}(\texttt{Q}_1\ \texttt{H.c}\ \texttt{Q}_2) = \begin{cases} \{\texttt{Q}_1\ \texttt{H.c}\ \texttt{Q}_2\} & \textit{se } \texttt{Q} = \texttt{Q}_1 \wedge \texttt{H.c} = \texttt{A.f} \\ & \qquad \wedge \texttt{Q}\ \texttt{A.f}\ \texttt{Q}_2 \in \texttt{C} \\ \varnothing & \textit{altrimenti} \end{cases}$$

2. $$\begin{aligned} &\mathbb{R}^{+(i+1)}_{\texttt{Q}}(\texttt{Q}_1\ \texttt{A.f}\ \texttt{Q}_2) = \mathbb{R}^{+(i)}_{\texttt{Q}}(\texttt{Q}_1\ \texttt{A.f}\ \texttt{Q}_2) \\ &\qquad \cup \left(\bigcup_{\texttt{Q}_3\ \texttt{H}'\texttt{.c}'\ \texttt{Q}_1 \in \texttt{C}} \{\mathbb{A} \triangleleft \texttt{Q}_1\ \texttt{A.f}\ \texttt{Q}_2 \mid \mathbb{A} \in \mathbb{R}^{+(i)}_{\texttt{Q}}(\texttt{Q}_3\ \texttt{H}'\texttt{.c}'\ \texttt{Q}_1)\}\right) \end{aligned}$$

3.
$$\begin{aligned}
&\textit{se } \mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'' \in \mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2 \wedge Cyclic_{\mathtt{Q}}(\mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2) = true, \textit{ allora}\\
&\mathbb{R}_{\mathtt{Q}}^{+(i+1)}(\mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'') =\\
&\quad \mathbb{R}_{\mathtt{Q}}^{+(i)}(\mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'')\\
&\quad \cup \left(\bigcup_{\mathtt{Q}''' \ \mathtt{H}'.\mathtt{c}'\ \mathtt{Q}' \in \mathtt{C}} \left\{ \mathbb{A} \triangleleft \mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'' \;\middle|\; \begin{array}{r} \mathbb{A} \in \mathbb{R}_{\mathtt{Q}}^{+(i)}(\mathtt{Q}''' \ \mathtt{H}'.\mathtt{c}'\ \mathtt{Q}') \\ \wedge \mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2 \in \mathbb{A} \end{array} \right\} \right)
\end{aligned}$$

$$\begin{aligned}
&\textit{se } \mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'' \in \mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2 \wedge Cyclic_{\mathtt{Q}}(\mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2) = false, \textit{ allora}\\
&\mathbb{R}_{\mathtt{Q}}^{+(i+1)}(\mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'') =\\
&\quad \mathbb{R}_{\mathtt{Q}}^{+(i)}(\mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'')\\
4.\quad &\cup(\bigcup_{\mathtt{Q}''' \ \mathtt{H}'.\mathtt{c}'\ \mathtt{Q}' \in \mathtt{C}} \{ \mathbb{A} \triangleleft \mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'' \mid \mathbb{A} \in \mathbb{R}_{\mathtt{Q}}^{+(i)}(\mathtt{Q}''' \ \mathtt{H}'.\mathtt{c}'\ \mathtt{Q}') \\
&\qquad\qquad \wedge \mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2 \in \mathbb{A} \\
&\qquad\qquad \wedge \mathbb{T}_{\nu}(\{\mathbb{A}\}) \leq \mathbb{T}_{\nu}(\mathbb{R}_{\mathtt{Q}}^{+(i)}(\mathtt{Q}_1\ \mathtt{A.f}\ \mathtt{Q}_2) \triangleleft \mathtt{Q}' \ \mathtt{ev.n}\ \mathtt{Q}'') \\
&\qquad\qquad \textit{è } \textbf{risolvibile}\})
\end{aligned}$$

Per definizione $\mathbb{R}_{\mathtt{Q}}^{+(i)} \leq \mathbb{R}_{\mathtt{Q}}^{+(i+1)}$*. Poiché l'insieme delle possibili mappature da clausole a insiemi di computazioni astratte è un reticolo finito, esiste* κ *tale che* $\mathbb{R}_{\mathtt{Q}}^{+(\kappa)} \leq \mathbb{R}_{\mathtt{Q}}^{+(\kappa+1)}$ *[10]. Sia* $\mathbb{R}_{\mathtt{Q}}^{+}$ *il punto fisso.*

A questo punto è possibile stabilire la raggiungibilità:

- $\mathtt{Q}'\ \mathtt{H.c}\ \mathtt{Q}''$ è raggiungibile se $\mathbb{R}_{\mathtt{Q}}^{+}(\mathtt{Q}'\ \mathtt{H.c}\ \mathtt{Q}'') \neq \varnothing$
- $\mathtt{Q}'\ \mathtt{H.c}\ \mathtt{Q}''$ è irraggiungibile se $\mathbb{R}_{\mathtt{Q}}^{+}(\mathtt{Q}'\ \mathtt{H.c}\ \mathtt{Q}'') = \varnothing$

Possono verificarsi casi in cui prese due clausole consecutive, dopo aver eseguito la prima non sarà mai possibile eseguire la seconda anche se entrambi sono raggiungibili. Questa proprietà è definita su coppie di clausole e viene identificata come **warning code**. La coppia $\mathtt{Q}'\ \mathtt{H.c}\ \mathtt{Q}'', \mathtt{Q}''\ \mathtt{H}'.\mathtt{c}'\ \mathtt{Q}'''$ è warning code se per ogni $\mathbb{A} \in \mathbb{R}_{\mathtt{Q}}^{+}(\mathtt{Q}''\ \mathtt{H}'.\mathtt{c}'\ \mathtt{Q}''')$, $\mathtt{Q}'\ \mathtt{H.c}\ \mathtt{Q}'' \notin \mathbb{A}$.

Come detto nel precedente esempio, durante il calcolo della risolvibilità tra due insiemi di tempi logici, possono essere confrontati tempi logici contenenti variabili. In questo caso, quando la relazione è risolvibile, è comunque specificare con quali vincoli la risolvibilità è soddisfatta. Tutti questi vincoli che vengono generati prendono il nome di **vincoli di raggiungibilità**

e forniscono un'informazioni in più per la quale, volendo garantire in fase di esecuzione la raggiungibilità delle clausole, è necessario rispettare quanto riportato dai vincoli.

Capitolo 4

Implementazione

4.1 Tecnologie utilizzate

Il codice è stato rilasciato sulla piattaforma GitHub [11] sotto licenza GNU GPL-3.0 [2]. Le tecnologie utilizzate per l'implementazione del software sono:

- Antlr [1]
- Python [3]

Antlr è un potente generatore di parser per linguaggi. Questo software permette di generare facilmente gli strumenti per l'analisi di programmi, quali lexer, parser e visitor, prendendo come input il file dove si definisce la grammatica del linguaggio con cui i programmi sono scritti. L'output di Antlr può essere prodotto per i principali linguaggi di programmazione e costituisce lo scheletro con il quale implementare le feature desiderate. Nel nostro caso l'output di Antlr è stato prodotto in Python.

Python è un linguaggio di programmazione orientato a oggetti. Ci sono diversi motivi per cui Python è stato scelto come linguaggio per l'implementazione del software:

- multipiataforma: è possibile installare Python su vari sistemi operativi, per esempio Linux, MacOS, Windows e Android;

- facilità d'uso: una volta installato Python, questo permetterà di eseguire facilmente i programmi scritti in Python;
- package manager: Python fornisce il suo package manager, pip, con il quale installare i moduli non inclusi nativamente;
- tipi di dato: Python implementa nativamente i tipi di dato `dict` e `set`, rispettivamente utilizzabili per mappature e insiemi.

Sono stati utilizzati tre moduli Python esterni le cui dipendenze sono state definite nel file **requirements.txt**:

- **antlr4-python3-runtime** [4]: contiene le classi base di Antlr implementate in Python;
- **click** [5]: permette di gestire uno script Python come comando da terminale strutturando le opzioni e i parametri in input;
- **python-dateutil** [6]: aggiunge feature per la gestione dei tipi di dato `datetime` e `timedelta`.

4.2 Grammatica

Nel nostro caso Antlr prende in input la grammatica del linguaggio Stipula e restituisce in output il lexer, il parser e la classe base del visitor. Il visitor permette di visitare l'albero di sintassi astratta relativo al programma che si fornisce in input. Il visitor prodotto da Antlr effettua una semplice visita completa dell'albero di sintassi astratta, quindi è necessario arricchire il visitor base per implementare le feature di analisi desiderate.

È importante sottolineare che Antlr produce il visitor partendo dalla definizione della grammatica, quindi esiste una dipendenza molto forte tra la definizione della grammatica e il comportamento del visitor in fase di visita dell'albero di sintassi astratta.

Di seguito vengono riportati i costrutti e le caratteristiche principali della grammatica utilizzata per l'implementazione dell'analizzatore:

- `stipula`: identifica il nome del contratto;
- `assets`: identifica gli asset. Questo costrutto è opzionale.
- `fields`: identifica i field. I field possono essere inizializzati con un valore di default.
- `init`: identifica lo stato iniziale del contratto;
- `agreement`: identifica i party e i field che sono interessati durante l'accordo. Questo costrutto è opzionale.
- `timeExpression`: rappresentano le espressioni temporali usate nella definizione degli eventi. Possono essere date, espresse come `DATESTRING`, campi che rappresentano date oppure somme nella forma `now` $+ t_1 +$ $... + t_n$. Nel terzo caso i vari t_1, ..., t_n possono essere qantità di tempo, espresse come `TIMEDELTA`, oppure campi che rappresentano quantità di tempo.
- `TIMEDELTA`: rappresentano quantità di tempo. Si esprimono con un valore accompagnato da un'unità di misura. Se l'unità di misura è assente

il valore viene considerato in minuti. Le unità di misura possibili sono `Y` per gli anni, `M` per i mesi, `D` per i giorni, `h` per le ore, `m` per i minuti, `s` per i secondi. I valori sono numeri interi ad eccezione dei secondi che possono anche essere numeri reali.

- `DATESTRING`: rappresentano date. Come pre le stringhe si possono usare sia gli apici (') che i doppi apici("). Il formato della data deve essere YYYY-MM-DD.

Nel caso delle `timeExpression` è necessario aggiungere delle precisazioni:

- `now` si riferisce all'istante temporale di esecuzione della funzione che contiene la `timeExpression`;
- `1Y` significa la data attuale tra un anno;
- `1M` significa la data attuale tra un mese. La gestione viene automatizzata tramite il modulo **python-dateutil**. Si segnala che nel caso il mese successivo abbia meno giorno dell'attuale, tutti i giorni che il mese attuale ha in eccesso rispetto al mese successivo vengono riportati come ultimo giorno del mese successivo. Per esempio se la data attuale è "1970-01-31", `now` + `1M` sarà la data "1970-02-28".

4.3 Struttura del codice

Prima di procedere all'analisi del codice con il quale viene implemenentato lo strumento descritto nel capitolo (3) è necessario descrivere la struttura del codice e dei componenti che implementano le varie feature.

Escludendo i file e le cartelle di utilità per il software, all'interno del repository si può trovare:

- **Stipula.g4**: contiene la grammatica di Stipula scritta per Antlr;
- **analyzer.py**: è il programma che, dato in input un contratto scritto in Stipula, permette di analizzare il contratto;
- **generated/**: contiene i file generati automaticamente da Antlr:
  - **StipulaVisitor.py**: contiene la classe base del visitor, chiamata `StipulaVisitor`, per il linguaggio Stipula generata da Antlr.
- **classes/**: contiene le classi create appositamente per questo software:
  - **visitor.py**: contiene la sottoclasse dello `StipulaVisitor` che permette di analizzare l'albero di sintassi astratta del contratto scritto in Stipula fornito come input;
  - **exceptions/**: contiene le eccezioni create appositamente per questo software;
  - **data/**: contiene le classi che permettono di strutturare i dati complessi:
    - **visitorentry.py**: contiene la classe che implementa la rappresentazione a quadruple delle clausole;
    - **visitoroutput.py**: contiene la classe che rappresenta l'output finale dell'analizzatore.

4.3.1 analyzer.py

Il file **analyzer.py** è uno script in Python che rappresenta il punto di ingresso del software implementato. Lo script prende in input un contratto scritto in Stipula e fornisce in output l'analisi relativa alla raggiungibilità del codice.

Inizialmente vengono svolti i passaggi definiti nella documentazione di Antlr [1]:

1. lettura del file in input;
2. generazione del lexer;
3. generazione della lista di token;
4. generazione del parser;
5. generazione dell'albero di sintassi astratta;
6. istanziazione del visitor;
7. visita dell'albero di sintassi astratta.

Come visitor verrà utililizzata la classe `Visitor`, sottoclasse dello `StpulaVisitor`, ovvero il visitor che implementa le feature di analisi descritte nel capitolo (3). Al termine della visita dell'albero di sintassi astratta l'output sarà un oggetto della classe `VisitorOutput`. Tutte le operazioni di analisi della raggiungibilità sono interne al metodo `Visitor.visit` quindi sono invisibili all'analizzatore. Di seguito viene riportato il frammento di codice appena descritto.

```
1 input_stream = antlr4.FileStream(file_path)
2 lexer = StipulaLexer(input_stream)
3 stream = antlr4.CommonTokenStream(lexer)
4 parser = StipulaParser(stream)
5 tree = parser.stipula()
```

```
6  if parser.getNumberOfSyntaxErrors() > 0:
7  print('Syntax errors')
8  sys.exit(1)
9  visitor = Visitor()
10 visitor_output = visitor.visit(tree)
```

Le fasi successive prevedono la lettura del `VisitorOutput` e la formattazione dell'output relativo. A questo scopo viene anche utilizzato il modulo **visitoroutputprints**.

4.3.2 VisitorEntry

I `VisitorEntry` sono utilizzati pre la rappresentazione a quadruple delle clausole. In Python, nei parametri di funzione, gli oggetti vengono passati per riferimento. Inoltre il riferimento all'oggetto viene utilizzato per definire l'uguaglianza tra oggetti. Quindi un oggetto sarà uguale solamente a sé stesso. Questa proprietà è di estrema importanza in quanto fa si che la stessa quadrupla, all'interno di strutture dati differenti, possa essere effettivamente lo stesso oggetto passato per riferimento.

Le tuple sono nativamente oggetti della classe `tuple`. Tuttavia si è preferito definire la classe `VisitorEntry` con lo scopo di rendere più chiaro il significato dei valori che compongono la quadrupla e poterli accedere tramite il nome dell'attributo. Un `VisitorEntry` è composto dai seguenti attributi:

- `start_state`: stato di inizio della clausola;
- `handler`: per le funzioni è la parte che invoca la funzione, per gli eventi è un placeholder;
- `code_id`: per le funzioni è il nome della funzione, per gli eventi è l'indice del carattere al quale inizia la definizione dell'evento, all'interno del contratto scritto in Stipula;
- `end_state`: stato su cui termina la clausola;

- `code_reference`: riferimento alla definizione della clausola, viene utilizzato un fase di produzione dell'output per rintracciare il codice originale della clausola.

Diversamente da quanto riportato nel capitolo (3), per gli eventi, in fase di implementazione, si è scelto di usare l'indice del carattere all'interno dell'input piuttosto che la linea di codice. Il motivo è che i contratti scritti in Stipula potrebbero essere scritti su un'unica linea di codice. Inoltre Antlr, tramite la sua API, permette di accedere facilmente all'indice che è stato utilizzato.

Successivamente sono state implementate due sottoclassi con lo scopo di distinguere strutturalmente le funzioni e gli eventi:

- `FunctionVisitorEntry`: per le funzioni;
- `EventVisitorEntry`: per gli eventi.

4.3.3 Visitor

Il `Visitor` è la sottoclasse dello `StipulaVisitor` generata da Antlr. Lo scopo primario di questa classe è quello di visitare l'albero di sintassi astratta estraendo le informazioni utili per l'analisi della raggiungibilità delle clausole. Successivamente vengono chiamati i metodi del `VisitorOutput` per il calcolo della raggiungibilità delle clausole.

La gestione dei metodi è racchiusa nel metodo `visitStipula` che identifica la visita del costrutto `stipula` ovvero la radice del contratto scritto in Stipula. Questo metodo viene chiamato dal metodo `Visitor.visit` utilizzato all'interno dell'`analyzer.py`. Il `Visitor` esegue le seguenti operazioni:

1. analisi dei campi. Nel caso di preassegnamenti il valore del capo viene momentaneamente salvato;
2. analisi dello stato iniziale del contratto;
3. analisi delle clausole.

L'analisi delle clausole viene effettuata analizzando una ad una le funzioni e il loro contenuto. Durante l'analisi di una funzione si svolgono i seguenti passaggi:

1. la funzione viene salvata in forma di `FunctionVisitorEntry`;

2. si analizzano gli `statement`, se il valore di un campo viene modificato, il valore del campo non viene più considerato attendibile. Nel caso il campo avesse un preassegnamento, il precedente valore viene rimosso;

3. si analizzano gli eventi che vengono salvati come `EventVisitorEntry`. Per gli eventi viene anche analizzata la `timeExpression`. Alla fine dell'analisi di ogni eventi vengono salvati:

   - la funzione che definisce l'evento;
   - i campi utilizzati nella `timeExpression`;
   - il valore numerico della `timeExpression` derivante dalla parte calcolabile.

Durante l'analisi di una `timeExpression` i valori vengono calcolati come `timedelta` di Python rispetto all'istante temporale di inizio dell'analisi. Durante l'analisi si possono incontrare i seguenti casi:

- quando la `timeExpression` inizia col costrutto `now` i successivi valori possono essere `TIMEDELTA` oppure campi che rappresentano `TIMEDELTA`:
  - i `TIMEDELTA` vengono valutati e salvati come `timedelta` di Python;
  - i campi rappresentano dei `TIMEDELTA`, se il valore del preassegnamento è disponibile e attendibile allora viene utilizzato, altrimenti si salva il campo come dipendenza per il calcolo.

- quando è presente una `DATESTRING` questa viene convertita in `timedelta` e viene verificato che sia nel futuro e non nel passato;

- quando è presente un campo questo sarà sicuramente una `DATESTRING`, se il valore preassegnato è disponibile e attendibile allora si considera il valore, altrimenti si salva il campo come dipendenza per il calcolo.

Si sottolinea che in questa fase anche il costrutto `now` viene salvato come dipendenza nel calcolo di una `timeExpression` in quanto sarà un'informazione necessaria per i passaggi successivi dell'analisi.

4.3.4 VisitorOutput

Il `VisitorOutput` rappresenta l'output dell'analizzatore. Lo scopo è quello di contenere gli attributi e i metodi necessari a costriure, analizzare e aggiornare correttamente la mappatura $\mathbb{R}_{\mathtt{Q}}^{+}$ relativa al contratto scritto in Stipula fornito come input. Per quanto riguarda i metodi, nelle sezioni successive verrà presentata un'analisi approfondita in relazione a quanto descritto nel capitolo (3). Di seguito vengono presentati gli attributi relativi al `VistorOutput`:

- `field_id_map`: è il dizionario che associa ad ogni campo definito nel contratto il suo valore in caso di preassegnamento. Se il valore del campo non viene preassegnato o il suo valore viene modificato da uno statement, al campo verrà assegnato il valore `None`;
- `Q0`: è l'identificativo dello stato iniziale del contratto;
- `C`: è l'insieme contenente tutte le clausole definite nel contratto;
- `Gamma`: è il dizionario che associa ad ogni evento la funzione che lo definisce;
- `dependency_t_map`: è il dizionario che associa ad ogni evento l'insieme contenente i costrutti e i campi da cui dipende il calcolo della relativa `timeExpression`;
- `t`: è il dizionario che associa ad ogni evento il valore della quantità di tempo calcolata analizzando la relativa `timeExpression`;

- `R`: è il dizionario che associa ad ogni clausola un insieme di tuple di clausole. Le tuple di clausole rappresentano le computazioni astratte;
- `reachability_constraint`: è l'insieme contenente coppie di tempi logici. Per poter eseguire tutto il codice del contratto, in ogni coppia il primo tempo logico deve essere minore o uguale del secondo tempo logico. In questo caso i tempi logici non sono semplici valori numerici ma somme tra campi e valori numerici. Quindi queste somme vengono utilizzate per stabilire dei vincoli;
- `warning_code`: è l'insieme contenente le coppie nelle quali dalla prima clausola non si può raggiungere la seconda, anche se la seconda risulta raggiungibile;
- `expired_code`: è il dizionario che associa ad ogni evento, con trigger temporale nel passato, la data contenuta nella `timeExpression`;
- `unreachable_code`: è l'insieme contenente le clausole irraggiungibili.

4.4 Implementazione dell'algoritmo

Durante la visita dell'albero di sintassi astratta, il `Visitor` costruisce già l'insieme `VisitorOutput.C`. All'interno di questo insieme saranno presenti tutti i `VisitorEntry` che rappresentano le clausole del contratto scritto in Stipula, ad eccezione degli eventi già scaduti perché utilizzano come espressione temporale una data nel passato.

A questo punto viene costruita la mappatura $\mathbb{R}_{\mathtt{Q}}^{+}$ tramite la funzione `VisitorOutput.compute_R`. La mappatura viene costruita tramite l'algoritmo del punto fisso rispettando le definizioni (2) e (7). Tuttavia viene introdotta una piccola ottimizzazione. Le due definizioni sono molto simili ad eccezione dell'utilizzo degli insiemi di tempi logici $\mathbb{T}_{\nu}$ in associazione alla funzione $Cyclic_{\mathtt{Q}}$. Il metodo `VisitorOutput.compute_R` è implementato come un gigantesco ciclo `while` che viene eseguito fino a quando non è più possibile effettuare modifiche al dizionario `VisitorOutput.R`. Per evitare codice duplicato, il metodo `VisitorOutput.compute_R` utilizza la definizione (2) per la prima iterazione e la definizione (7) per tutte le iterazioni successive. L'implementazione dei restanti metodi è banale e segue in maniera molto aderente quanto descritto nel capitolo (3). Di seguito vengono elencati i principali metodi utilizzati nell'algoritmo:

- `VisitorOutput.is_cyclic`: restituisce `True` quando la `FunctionVisitorEntry` fornita come input è eseguibile ciclicamente, `False` altrimenti. Rappresenta l'implementazione della funzione $Cyclic_{\mathtt{Q}}$.
- `VisitorOutput.Theta`: rappresenta l'implementazione della funzione Θ_{ν};
- `VisitorOutput.T`: rappresenta l'implementazione della funzione $\mathbb{T}_{\nu}$;
- `VisitorOutput.reduce_reachability_constraint`: permette di ridurre due tempi logici per poterli successivamente analizzare. I tempi logici sono gestiti sotto forma di `ValueDependency` ovvero ogget-

ti contenenti una parte numerica e una tupla di campi. Dati due `ValueDependency`, di cui il primo deve essere minore o uguale del secondo, questo metodo permette di ottenere due `ValueDependency` ridotti in modo che la relativa relazione sia valutabile più facilmente.

- `VisitorOutput.is_solvable`: dati due `ValueDependency` ridotti, restituisce `True` se la relazione è soddisfacibile, `False` altrimenti.

La funzione Θ_ν inserisce nel calcolo dei tempi logici i simboli $\nu(\texttt{Q A.f Q}')$. Nell'analizzatore questo aspetto è stato implementato generando variabili fittizie per cui $\nu(\texttt{Q1 A.f Q2})$ genera la variabile `Q1:A.f:Q2` che rappresenta il tempo di esecuzione della funzione associata. Quando nella `timeExpression` di un evento è presente una data, precedentemente si è detto che la data viene convertita in una `timeExpression` nella forma $\texttt{now} + \texttt{n}$. Tuttavia, in fase di implementazione dell'analizzatore, questa conversione non è così semplice. Risulta semplice, invece, convertire la data presente nella `timeExpression` in una differenza di tempo rispetto all'istante di inizio dell'analisi che può essere assimilato all'istante di inizio del contratto. La presenza di un evento all'interno di una computazione astratta significa che l'evento, nella relativa computazione, viene eseguito e sarà rispettato il suo innesco temporale. Se l'innesco temporale è formato da una data fissa, allora l'evento verrà eseguito esattamente in quella data. A questo punto la correzione al metodo `visitorOutput.Theta` prevede che, esprimendo le computazioni astratte come tuple, quando in una computazione astratta è presente un evento con data fissa come `timeExpression`, tutte le clausole precedenti non vengono considerate per il calcolo del tempo logico. È possibile riconoscere gli eventi con data fissa in quanto non utilizzano il costrutto `now`.

Al termine dell'esecuzione dell'algoritmo, analizzando il dizionario `VisitorOutput.R`, e per differenza dai `VisitorEntry` presenti nell'insieme `VisitorOutput.C`, sarà possibile determinare quali clausole sono irraggiungibili e quali sono le condizioni di warning code. Questo è possibile secondo quanto descritto al termine della sezione (3.3.2).

4.4.1 Vincoli di raggiungibilità

L'algoritmo per determinare se una clausola è raggiungibile confronta i tempi logici assegnabili ad una clausole con i tempi logici di tutte le clausole il cui stato finale coincide con lo stato iniziale della clausola in analisi. Facendo questo, l'insieme delle computazioni astratte della clausola in analisi viene generato includendo le computazioni astratte che possono rappresentare computazioni. Tuttavia, i tempi logici sono somme di variabili, field e valori numerici. Quindi il fatto che il confronto tra due tempi logici, $\mathtt{t}_1 \leq \mathtt{t}_2$, sia risolvibile non garantisce che in fase di esecuzione del contratto tutte le clausole siano raggiungibili.

Con lo scopo di fornire un'indicazione più precisa in fase di analisi di un contratto scritto in Stipula, vengono introdotti i **vincoli di raggiungibilità**. Quando si analizzano due tempi logici che includono variabili, il confronto, se risolvibile, viene riportato come vincolo di raggiungibilità. Al termine dell'analisi del contratto, sarà possibile determinare i vincoli per i quali le clausole del contratto classificate come raggiungibili lo siano effettivamente anche in fase di esecuzione.

4.5 Utilizzo dell'analizzatore

L'analizzatore è uno script python invocabile da console. La sintassi del comando è:

```
python analyzer.py [OPTIONS] <PATH>
```

dove:

- `PATH` è il percorso del file contenente il contratto scritto in Stipula da analizzare;
- `OPTIONS` sono parametri aggiuntivi:
  - `-r`, `--readable`: modifica il template con cui vengono rappresentate le clausole;
  - `-c`, `--compact`: riduce l'output grafico mostrando solamente le informazioni importanti.

L'analizzatore fornisce in output l'analisi del contratto. Le clausole vengono rappresentate in due modi diversi:

- `fn("Q1", "A", "f", "A2")` per le funzioni. Utilizzando l'opzione `-r` diventa `Q1 A.f Q2`.
- `ev("Q1", "Ev", 33, "Q2")` per gli eventi. Usando l'opzione `-r` divente `Q1 ev.22 Q2`.

Nell'output vengono mostrati le seguenti informazioni:

- un dump del `VisitorOutput`;
- la sezione `REACHABILITY CONSTRAINT` contenente i vincoli di raggiungibilità delle clausole. Questi vincoli vengono mostrati in forma `t1 + ... + tn <= k1 + ... + km`;
- la sezione `WARNING CODE` contenente le coppie di clausole per cui dalla prima non è possibile eseguire la seconda;

- la sezione `EXPIRED CODE` che contiene tutti gli eventi che utilizzano una data nel passato come `timeExpression`;
- la sezione `UNREACHABLE CODE` che contiene tutte le clausole irraggiungibili.

Se non si utilizza l'opzione `-c`, per ogni clausola sarà presente nell'output il corpo della clausola e l'indicazione alla riga di codice dove inizia la definizione.

Capitolo 5

Test

5.1 Raggiungibilità topologica

Di seguito viene riportato un contratto di esempio dove l'unico evento generabile non è mai eseguibile in quanto una volta generato l'evento non sarà mai possibile raggiungere lo stato da cui l'evento inizia.

```
stipula SampleTopo {
        fields x
        init Q0

        @Q0 A: f(y)[k] {
                now >> @Q2 {} => @Q3
        } => @Q1

        @Q0 A1: f1(y)[k] {} => @Q2
}
```

Eseguendo l'analizzatore si ottiene il seguente output:

```
+-------------+
|   Results   |
+-------------+
```

```
{
        "field_id_map": {
                "x": "None"
        },
        "Q0": "Q0",
        "C": [
                "fn('Q0', 'A1', 'f1', 'Q2')",
                "fn('Q0', 'A', 'f', 'Q1')",
                "ev('Q2', 'Ev', 41, 'Q3')"
        ],
        "Gamma": {
                "ev('Q2', 'Ev', 41, 'Q3')": "fn('Q0', 'A', 'f', 'Q1')"
        },
        "dependency_t_map": {
                "ev('Q2', 'Ev', 41, 'Q3')": [
                        "now"
                ]
        },
        "t": {
                "ev('Q2', 'Ev', 41, 'Q3')": "0:00:00"
        },
        "R": {
                "fn('Q0', 'A1', 'f1', 'Q2')": [
                        [
                                "fn('Q0', 'A1', 'f1', 'Q2')"
                        ]
                ],
                "fn('Q0', 'A', 'f', 'Q1')": [
                        [
                                "fn('Q0', 'A', 'f', 'Q1')"
                        ]
                ],
                "ev('Q2', 'Ev', 41, 'Q3')": []
```

```
        },
        "reachability_constraint": [],
        "warning_code": [],
        "expired_code": {},
        "unreachable_code": [
                "ev('Q2', 'Ev', 41, 'Q3')"
        ]
}
UNREACHABLE CODE [
        ev('Q2', 'Ev', 41, 'Q3') :
line 10:   now >> @Q2 {} => @Q3
]
```

5.2 Raggiungibilità temporale

Di seguito viene riportato un contratto di esempio dove il secondo evento definito non può essere eseguito in quanto sarà sepre scaduto quando il contratto si troverà in quello stato.

```
stipula SampleTime {
        fields x
        init Q0

        @Q0 A: f(y)[k] {
                now + 1D >> @Q1 {} => @Q2

                now >> @Q2 {} => @Q3
        } => @Q1
}
```

Eseguendo l'analizzatore si ottiene il seguente output:

```
+-------------+
|   Results   |
+-------------+
{
        "field_id_map": {
                "x": "None"
        },
        "Q0": "Q0",
        "C": [
                "ev('Q1', 'Ev', 37, 'Q2')",
                "fn('Q0', 'A', 'f', 'Q1')",
                "ev('Q2', 'Ev', 48, 'Q3')"
        ],
        "Gamma": {
                "ev('Q1', 'Ev', 37, 'Q2')": "fn('Q0', 'A', 'f', 'Q1')",
```

```
                "ev('Q2', 'Ev', 48, 'Q3')": "fn('Q0', 'A', 'f', 'Q1')"
        },
        "dependency_t_map": {
                "ev('Q1', 'Ev', 37, 'Q2')": [
                        "now"
                ],
                "ev('Q2', 'Ev', 48, 'Q3')": [
                        "now"
                ]
        },
        "t": {
                "ev('Q1', 'Ev', 37, 'Q2')": "1 day, 0:00:00",
                "ev('Q2', 'Ev', 48, 'Q3')": "0:00:00"
        },
        "R": {
                "ev('Q1', 'Ev', 37, 'Q2')": [
                        [
                                "fn('Q0', 'A', 'f', 'Q1')",
                                "ev('Q1', 'Ev', 37, 'Q2')"
                        ]
                ],
                "fn('Q0', 'A', 'f', 'Q1')": [
                        [
                                "fn('Q0', 'A', 'f', 'Q1')"
                        ]
                ],
                "ev('Q2', 'Ev', 48, 'Q3')": []
        },
        "reachability_constraint": [],
        "warning_code": [],
        "expired_code": {},
        "unreachable_code": [
        "ev('Q2', 'Ev', 48, 'Q3')"
```

```
        ]
}
UNREACHABLE CODE [
        ev('Q2', 'Ev', 48, 'Q3') :
line 12:   now >> @Q2 {} => @Q3
]
```

5.3 Warning code

Di seguito viene riportato un contratto di esempio dove viene generato warning code. Si ricorda che il warning code si genera quando per due clausole che sarebbero subito consecutive, eseguendo la prima non è mai possibile eseguire la seconda.

```
stipula SampleWCod {
        fields x
        init Q0

        @Q0 A: f(y)[k] {
                now + 2 >> @Q3 {} => @Q4
        } => @Q1

        @Q1 B: g(y)[k] {
                now + 1 >> @Q2 {} => @Q3
        } => @Q2

        @Q1 C: d(y)[k] {
                now + 3 >> @Q2 {} => @Q3
        } => @Q2
}
```

Eseguendo l'analizzatore si ottiene il seguente output:

```
+-------------+
|   Results   |
+-------------+
{
        "field_id_map": {
                "x": "None"
        },
        "Q0": "Q0",
```

```
        "C": [
                "ev('Q3', 'Ev', 37, 'Q4')",
                "fn('Q0', 'A', 'f', 'Q1')",
                "fn('Q1', 'B', 'g', 'Q2')",
                "ev('Q2', 'Ev', 67, 'Q3')",
                "fn('Q1', 'C', 'd', 'Q2')",
                "ev('Q2', 'Ev', 97, 'Q3')"
        ],
        "Gamma": {
                "ev('Q3', 'Ev', 37, 'Q4')": "fn('Q0', 'A', 'f', 'Q1')",
                "ev('Q2', 'Ev', 67, 'Q3')": "fn('Q1', 'B', 'g', 'Q2')",
                "ev('Q2', 'Ev', 97, 'Q3')": "fn('Q1', 'C', 'd', 'Q2')"
        },
        "dependency_t_map": {
                "ev('Q3', 'Ev', 37, 'Q4')": [
                        "now"
                ],
                "ev('Q2', 'Ev', 67, 'Q3')": [
                        "now"
                ],
                "ev('Q2', 'Ev', 97, 'Q3')": [
                        "now"
                ]
        },
        "t": {
                "ev('Q3', 'Ev', 37, 'Q4')": "0:02:00",
                "ev('Q2', 'Ev', 67, 'Q3')": "0:01:00",
                "ev('Q2', 'Ev', 97, 'Q3')": "0:03:00"
        },
        "R": {
                "ev('Q3', 'Ev', 37, 'Q4')": [
                        [
                                "fn('Q0', 'A', 'f', 'Q1')",
```

```
                        "fn('Q1', 'B', 'g', 'Q2')",
                        "ev('Q2', 'Ev', 67, 'Q3')",
                        "ev('Q3', 'Ev', 37, 'Q4')"
                ]
        ],
        "fn('Q0', 'A', 'f', 'Q1')": [
                [
                        "fn('Q0', 'A', 'f', 'Q1')"
                ]
        ],
        "fn('Q1', 'B', 'g', 'Q2')": [
                [
                        "fn('Q0', 'A', 'f', 'Q1')",
                        "fn('Q1', 'B', 'g', 'Q2')"
                ]
        ],
        "ev('Q2', 'Ev', 67, 'Q3')": [
                [
                        "fn('Q0', 'A', 'f', 'Q1')",
                        "fn('Q1', 'B', 'g', 'Q2')",
                        "ev('Q2', 'Ev', 67, 'Q3')"
                ]
        ],
        "fn('Q1', 'C', 'd', 'Q2')": [
                [
                        "fn('Q0', 'A', 'f', 'Q1')",
                        "fn('Q1', 'C', 'd', 'Q2')"
                ]
        ],
        "ev('Q2', 'Ev', 97, 'Q3')": [
                [
                        "fn('Q0', 'A', 'f', 'Q1')",
                        "fn('Q1', 'C', 'd', 'Q2')",
```

```
                                "ev('Q2', 'Ev', 97, 'Q3')"
                        ]
                ]
        },
        "reachability_constraint": [
                [
                        [
                                "Q1:B.g:Q2"
                        ],
                        [
                                "0:01:00"
                        ]
                ]
        ],
        "warning_code": [
                [
                        "fn('Q1', 'B', 'g', 'Q2')",
                        "ev('Q2', 'Ev', 97, 'Q3')"
                ],
                [
                        "ev('Q2', 'Ev', 97, 'Q3')",
                        "ev('Q3', 'Ev', 37, 'Q4')"
                ],
                [
                        "fn('Q1', 'C', 'd', 'Q2')",
                        "ev('Q2', 'Ev', 67, 'Q3')"
                ]
        ],
        "expired_code": {},
        "unreachable_code": []
}
REACHABILITY CONSTRAINT [
        Q1:B.g:Q2 <= 0:01:00
```

```
]
WARNING CODE [
        fn('Q1', 'B', 'g', 'Q2') --X-> ev('Q2', 'Ev', 97, 'Q3')
        fn('Q1', 'B', 'g', 'Q2') :
line 13:   @Q1 B: g(y)[k] (true) {
        now + 1 >> @Q2 {} => @Q3
} => @Q2
        ev('Q2', 'Ev', 97, 'Q3') :
line 18:   now + 3 >> @Q2 {} => @Q3

        ev('Q2', 'Ev', 97, 'Q3') --X-> ev('Q3', 'Ev', 37, 'Q4')
        ev('Q2', 'Ev', 97, 'Q3') :
line 18:   now + 3 >> @Q2 {} => @Q3
        ev('Q3', 'Ev', 37, 'Q4') :
line 10:   now + 2 >> @Q3 {} => @Q4

        fn('Q1', 'C', 'd', 'Q2') --X-> ev('Q2', 'Ev', 67, 'Q3')
        fn('Q1', 'C', 'd', 'Q2') :
line 17:   @Q1 C: d(y)[k] (true) {
        now + 3 >> @Q2 {} => @Q3
} => @Q2
        ev('Q2', 'Ev', 67, 'Q3') :
line 14:   now + 1 >> @Q2 {} => @Q3
]
```

5.4 Vincoli di raggiungibilità

Di seguito viene riportato un contratto di esempio dove per rendere tutte le clausole effettivamente eseguibili in fase di esecuzione è necessario rispettare un vincolo sui tempi.

```
stipula EsempioWCon {
        fields t1, t2
        init Q0

        @Q0 A: f(y)[k] {
                now + t1 >> @Q1 {} => @Q2

                now + t2 >> @Q2 {} => @Q3
        } => @Q1
}
```

Eseguendo l'analizzatore si ottiene il seguente output:

```
+-------------+
|   Results   |
+-------------+
{
        "field_id_map": {
                "t1": "None",
                "t2": "None"
        },
        "Q0": "Q0",
        "C": [
                "ev('Q1', 'Ev', 43, 'Q2')",
                "fn('Q0', 'A', 'f', 'Q1')",
                "ev('Q2', 'Ev', 54, 'Q3')"
        ],
        "Gamma": {
```

```
                "ev('Q1', 'Ev', 43, 'Q2')": "fn('Q0', 'A', 'f', 'Q1')",
                "ev('Q2', 'Ev', 54, 'Q3')": "fn('Q0', 'A', 'f', 'Q1')"
        },
        "dependency_t_map": {
                "ev('Q1', 'Ev', 43, 'Q2')": [
                        "now",
                        "t1"
                ],
                "ev('Q2', 'Ev', 54, 'Q3')": [
                        "now",
                        "t2"
                ]
        },
        "t": {
                "ev('Q1', 'Ev', 43, 'Q2')": "0:00:00",
                "ev('Q2', 'Ev', 54, 'Q3')": "0:00:00"
        },
        "R": {
                "ev('Q1', 'Ev', 43, 'Q2')": [
                        [
                                "fn('Q0', 'A', 'f', 'Q1')",
                                "ev('Q1', 'Ev', 43, 'Q2')"
                        ]
                ],
                "fn('Q0', 'A', 'f', 'Q1')": [
                        [
                                "fn('Q0', 'A', 'f', 'Q1')"
                        ]
                ],
                "ev('Q2', 'Ev', 54, 'Q3')": [
                        [
                                "fn('Q0', 'A', 'f', 'Q1')",
                                "ev('Q1', 'Ev', 43, 'Q2')",
```

```
                                "ev('Q2', 'Ev', 54, 'Q3')"
                        ]
                ]
        },
        "reachability_constraint": [
                [
                        [
                                "t1"
                        ],
                        [
                                "t2"
                        ]
                ]
        ],
        "warning_code": [],
        "expired_code": {},
        "unreachable_code": []
}
REACHABILITY CONSTRAINT [
        t1 <= t2
]
```

Capitolo 6

Conclusioni

Il fulcro di questa tesi è stato l'implementazione di un prototipo di strumento in grado di analizzare i contratti legali scritti in Stipula, allo scopo di evidenziare clausole irraggiungibili, ovvero che non possono mai essere eseguite durante lo svolgimento del contratto. Una volta implementato, sono stati generati dei contratti di test con lo scopo sia di fornire esempi di output, forniti dallo strumento, sia di verificare il comportamento dello strumento.

A sostegno del prototipo è stata presentata la teoria della raggiungibilità, ovvero la formalizzazione di algoritmi, funzioni e proprietà necessarie a determinare la raggiungibilità delle clausole. Di conseguenza, in fase di implementazione dell'analizzatore, si è deciso di mantenere una discreta aderenza rispetto all'algoritmo descritto nella teoria della raggiungibilità.

Nel capitolo (3) si è fatto riferimento al principio di libertà di forma [15] secondo cui un contratto legale può essere scritto anche utilizzando un linguaggio di programmazione. L'utilizzo di un linguaggio di programmazione presenta diversi vantaggi: sostituendo il linguaggio naturale permette di rimuovere l'ambiguità e la difficoltà del linguaggio naturale stesso; inoltre, utilizzando un linguaggio di programmazione, è possibile analizzare, ottimizzare ed eseguire automaticamente parti dei contratti. Nel caso di questa tesi, è stato scelto il linguaggio Stipula.

Anche elencando i vantaggi dell'utilizzo di Stipula, non è da trascurare

la possibile resistenza da parte di chi non ha familiarità con i linguaggi di programmazione e fa dei contratti legali l'oggetto del proprio lavoro. In questo caso, prendendo un contratto legale scritto in linguaggio naturale, è sempre possibile produrre un contratto equivalente scritto in Stipula [7, 8]. A questo punto è possibile utilizzare l'analizzatore, proposto in questa tesi, per analizzare la raggiungibilità delle clausole del contratto equivalente scritto in Stipula. Il risultato, riportato sul contratto originale, permetterà di identificare i passaggi del contratto che non hanno mai nessun effetto sullo svolgimento del contratto, evidenziando quindi possibili errori.

Quindi, l'analizzatore proposto, secondo quanto descritto, risulta generalmente efficace nell'analisi dei contratti legali, indipendentemente dal linguaggio utilizzato.

Vanno comunque considerati gli aspetti non catturabili dalla teoria della raggiungibilità e, di conseguenze, dall'analizzatore. Precedentemente sono state descritte le computazioni astratte, estremamente importanti per tutta la teoria della raggiungibilità, ma che non permettono di catturare esplicitamente i cicli. Per questo motivo gli eventi generabili ciclicamente non possono essere analizzati in termini di tempi logici. Tuttavia è presente un'altra condizione, che genera falsi negativi, derivante dalla semantica operazionale di Stipula. Si sta facendo riferimento al fatto che gli eventi hanno sempre precedenza sulle funzioni, quindi un evento potrebbe essere eseguito sempre prima di una funzione, rendendo la funzione irraggiungibile.

Esempio Si consideri il seguente contratto `EventBeforeFunction`:

```
1 stipula EventBeforeFunction {
2         fields x
3         init Init
4
5         @Init A: f()[] {
6                 now >> @First {} => @End
7         } => @First
```

```
8
9           @First B: g()[] {} => @End
10  }
```

Nel contratto `EventBeforeFunction` invocando la funzione `A.f` si genera l'evento `ev.6` e il contratto passa allo stato `First`. Sullo stato first sarebbe possibile eseguire funzione `B.g`, ma l'evento `ev.6` avrà sempre massima precedenza e sarà sempre eseguito dopo `A.f`. Di conseguenza `B.g` è irraggiungibile.

Secondo la teoria della raggiungibilità `First B.g End` viene classificata come reachable. Quindi, in fase di analisi delle clausole irraggiungibili, questa è un falso negativo. Tuttavia, come dichiarato, sono ammessi i falsi negativi ma non sono ammessi i falsi positivi.

In conclusione, la teoria della raggiungibilità resta comunque un'ottima soluzione al problema della raggiungibilità e, di conseguenza, l'analizzatore si dimostra essere uno strumento molto utile nella ricerca di clausole irraggiungibili.

Bibliografia

[1] Antlr.
`https://www.antlr.org`.

[2] Gnu General Public Licence 3.
`https://www.gnu.org/licenses/gpl-3.0.html`.

[3] Python.
`https://www.python.org`.

[4] `antlr4_python3_runtime` Package.
`https://pypi.org/project/antlr4-python3-runtime/`.

[5] `click` Package.
`https://pypi.org/project/click/`.

[6] `python_dateutil` Package.
`https://pypi.org/project/python-dateutil/`.

[7] Silvia Crafa and Cosimo Laneve. Programming legal contracts - A beginners guide to *Stipula*. In *The Logic of Software. A Tasting Menu of Formal Methods - Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, volume 13360 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 2022.

[8] Silvia Crafa, Cosimo Laneve, Giovanni Sartor, and Adele Veschetti. Pacta sunt servanda: Legal contracts in *Stipula*. *Sci. Comput. Program.*, 225:102911, 2023.

[9] Silvia Crafa, Cosimo Laneve, and Adele Veschetti. The Stipula Project, July 2022.
Disponibile su github: `https://github.com/stipula-language`.

[10] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 1990.

[11] Samuele Evangelisti. The Stipula Analyzer, 2024.
Disponibile su github:
`https://github.com/stipula-language/stipula-analyzer`.

[12] Lexon Foundation. Lexon Home Page, 2019.
`http://www.lexon.tech`.

[13] Cosimo Laneve, Alessandro Parenti, and Giovanni Sartor. Legal contracts amending with Stipula. In *Proceedings of COORDINATION'23*, volume 13908 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2023.

[14] Open Source Contributors. The Accord Project, 2018.
`https://accordproject.org`.

[15] Research Group on EC Private Law (Acquis Group) Study Group on a European Civil Code. *Principles, Definitions and Model Rules of European Private Law: Draft Common Frame of Reference (DCFR), Outline Edition*. Sellier, 2009.

[16] Aaron Wright, David Roon, and ConsenSys AG. OpenLaw Web Site, 2019.
`https://www.openlaw.io`.

Ringraziamenti

Ringrazio il mio Relatore, Prof. Cosimo Laneve, per il prezioso supporto fornitomi, sia formativo sia in fase di stesura di questo elaborato, per la sensibilità e per l'onestà dimostrate nei miei confronti durante tutti i momenti del mio percorso accademico che ho avuto il piacere di condividere.

Ringrazio il mio Correlatore, Dott. Alessandro Parenti, per il prezioso supporto, talvolta multidisciplinare, fornitomi in fase di implementazione e analisi dell'analizzatore.

Ringrazio la mia fidanzata, i miei familiari, i miei parenti, i miei amici e i miei conoscenti che abbiano dimostrato empatia durante i vari momenti del mio percorso accademico.