Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Implementazione di un'interfaccia grafica per la scrittura di contratti legali nel linguaggio Stipula

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Presentata da:
Elia Venturi

III Sessione
Anno Accademico 2021/2022

Indice

Introduzione

Stipula è un linguaggio di programmazione che permette la scrittura di contratti legali attraverso un limitato insieme di astrazioni che ne catturano esattamente gli elementi distintivi.
Questa tesi ha come obiettivo il mostrare come un'interfaccia grafica possa essere un utile strumento per permettere di scrivere contratti nel linguaggio Stipula.
Stipula è un linguaggio formale, richiede una certa conoscenza della sua grammatica e sintassi per poter essere utilizzato correttamente. Tuttavia, molte persone che desiderano scrivere contratti legali potrebbero non avere competenze specifiche nel campo della programmazione, quindi avere delle difficoltà ad approcciarsi allo scrivere un contratto non in linguaggio naturale.
L'interfaccia sviluppata mira a semplificare l'utilizzo di Stipula, dando un modo alternativo di scrivere un contratto e rendendolo più accessibile ad un certo pubblico.
La sfida è stata trovare delle soluzioni grafiche che permettono di catturare tutte le funzioni del linguaggio, intermediando tra l'utente e la produzione di codice, mantenendo un'esperienza d'uso il più chiara e minimale possibile.
Lo sviluppo dell'interfaccia è iniziato durante il periodo di tirocinio curriculare svolto all'interno dell'Università, per poi essere integrato e concluso successivamente per la stesura di questa tesi.
Vedremo come prima cosa la struttura e le funzionalità del linguaggio Stipula, per poi approfondire l'interfaccia. Nell'ultimo capitolo vedremo un'estensione del linguaggio e come questo è stato integrato nell'interfaccia.

Capitolo 1

Il linguaggio Stipula

1.1 Introduzione al linguaggio

Le informazioni riportate in questo capitolo sono estratte ed elaborate dall'articolo [1].
Il diritto è uno dei settori che sta subendo maggiormente l'influenza della rivoluzione digitale. La digitalizzazione dei testi giuridici, come leggi, regolamenti, decisioni amministrative, contratti e sentenze, è un compito difficile perché questi testi sono espressi attraverso il linguaggio naturale, che è molto espressivo, ma anche ambiguo.
Il linguaggio Stipula ha come obiettivo la digitalizzazione dei contratti legali, che sono accordi vincolanti tra parti. Un linguaggio di programmazione in questo campo dovrebbe avere le caratteristiche di essere facile da usare e capire per i professionisti del diritto e i cittadini comuni, ma allo stesso tempo essere sufficientemente espressivo e avere una semantica precisa.
Stipula vuole essere per questo un linguaggio intermedio: più concreto di un linguaggio di specifica contrattuale user-friendly, ma più astratto di un linguaggio di programmazione completo. L'obiettivo è la conservazione delle caratteristiche individuate come fondamentali nei contratti legali nella loro rappresentazione in codice computazionale.
Utilizzare un linguaggio di programmazione per scrivere i contratti può offri-

re numerosi vantaggi per quanto riguarda non ambiguità e automatizzazione dell'applicazione delle clausole contrattuali.

1.2 La struttura del linguaggio

In questa sezione vedremo come è strutturato un contratto, e quali sono le parti da cui è composto, utilizzando come esempio un ipotetico contratto di noleggio di una bicicletta.

1.2.1 Gli asset e i field

```
1 stipula BikeRental {
2 asset wallet
3 field cost, rentingTime, code
4 init Inactive
```

Questo codice è la prima parte dell'esempio di contratto, possiamo vedere subito come vengono dichiarati due tipi di componenti, gli **Asset** e i **Field**. Entrambi i tipi vengono utilizzati per memorizzare dei dati, ma presentano delle differenze:

Field: contengono dati semplici che possono essere aggiornati dalle parti in accordo, come ad esempio limiti di tempo, costi o numeri di identificazione di un bene.

Asset: devono rispettare una quantità totale e non possono essere duplicati, quando vengono spostati in un nuovo indirizzo, la loro posizione precedente viene svuotata. Possono rappresentare sia un bene in forma divisibile, rappresentando una somma di denaro, che indivisibile, rappresentando un bene non fungibile.

Successivamente a queste dichiarazioni viene impostato lo **stato** iniziale. La programmazione con Stipula è infatti orientata agli stati, funzionamento ap-

profondito nel corso del capitolo che riguarda le funzioni.

Nell'esempio vediamo nella prima riga la dichiarazione del nome del contratto `BikeRental`, nel corpo del contratto invece viene dichiarato l'asset `wallet`, e i field: `cost, rentingTime, code`. Il nome dello stato iniziale è `Inactive`.

1.2.2 L'agreement

```
1 agreement (Lender,Borrower,Authority)(rentingTime,
     cost){
2   Lender , Borrower: rentingTime , cost
3 } ==> @Inactive
```

L'agreement è il primo punto fondamentale di ogni contratto Stipula, rappresenta la fase di contrattazione e accordo. I primi campi indicati `(Lender, Borrower, Authority)` sono i soggetti partecipanti al contratto, ovvero i **party**; i secondi campi `(rentingTime, cost)` sono i field su cui è necessario mettersi d'accordo.
Nel corpo all'interno delle parentesi graffe troviamo la lista vera e propria degli accordi nella forma:

$P_1 : F_1$

...

$P_n : F_n$

Dove P_1 ... P_n sono sottoinsiemi di party tra quelli dichiarati sopra, mentre F_1 ... F_n sono sottoinsiemi dei field dichiarati sopra: ogni field può essere accordato una sola volta, mentre un party può mettersi d'accordo su più fields. Nell'esempio è presente un solo accordo, in cui `Lender` e `Borrower` si accordano su `rentingTime` e `cost`.
Alla fine dell'agreement troviamo il primo stato dove la funzione transiterà al termine degli accordi.

1.2.3 Le funzioni

```
@Inactive Lender : offer(x)[] {
  x -> code;
  _
} ==> @Payment
@Payment Borrower : pay()[h] (h == cost) {
  h -o wallet
  code -> Borrower;
  now+rentingTime >>
    @Using {
    EndReached -> Borrower
    } ==> @Return
} ==> @Using
@Using Borrower : end()[] {
  "EndReached" -> Lender;
  _
} ==> @Return
@Return Lender : rentalOk()[] {
  wallet -o Lender;
  _
} ==> @End
@Using@Return Lender, Borrower : dispute(x)[] {
  x -> _;
  _
} ==> @Dispute
@Dispute Authority : verdict(x,y)[] (y>=0 && y<=1) {
  x -> Lender
  x -> Borrower
  (y*wallet) -o wallet, Lender
  wallet -o Borrower;
  _
```

```
31 } ==> @End
32 }
```

Come prima cosa introduciamo i due tipi di operatori fondamentali che possiamo vedere nelle azioni nel corpo delle funzioni:

-o : viene utilizzato per operazioni con gli asset, esprime uno spostamento.
`a -o b, c`: significa che una quantità a viene spostata da `b` verso `c`.
`a -o a, b` può essere indicato anche come: `a -o b`.

-> : viene utilizzato per dei semplici invii di informazione, `a -> b`: l'informazione `a` viene inviata a `b`. Ma il valore di `a` rimane inalterato.

Anche se non presente nel nostro esempio, nel corpo di una funzione è possibile anche richiamare il costrutto if nella forma `if (condition) then... else if(condition)... else...`.

Nelle definizioni delle funzioni il primo campo da specificare è una lista di **stati** indicati come `@Nomestato`. Per poter chiamare una funzione è necessario che lo stato corrente del programma sia in uno di questi. Questo permette di essere chiamata solo in dei determinati stati; di consequenza ogni stato avrà un numero limitato di funzioni da chiamare.
Per esempio, abbiamo detto che dopo l'agreement lo stato è stato impostato a `@Inactive`, da qui possiamo osservare come l'unica funzione che è possibile chiamare sia `offer`.
Subito dopo la lista degli stati, è presente la lista dei **party** che possono chiamare la funzione, quindi allo stato `@Inactive`, solo `Lender` può chiamare `offer`. Alla fine di ogni funzione è necessario specificare un nuovo stato in cui il programma deve transitare nella forma `==> @NuovoStato`.

Le funzioni possono necessitare di **parametri**, tra parentesi tonde ci sono parametri del tipo `field`, tra parentesi quadre sono presenti parametri di tipo `asset`. Infine è possibile specificare delle condizioni tra parentesi tonde

per permettere o meno di eseguire la funzione, come nella funzione `pay`.

Seguendo l'esempio chiamiamo la funzione `offer`, quindi prendiamo in input un field che verrà inviato a `code`, per poi transitare allo stato `@Payment`. Da `@Payment` troviamo di nuovo un'unica funzione chiamabile, questa volta è `pay` che può essere chiamata da `Borrower`.
Non sempre è presente un'unica possibile funzione da chiamare, per esempio vediamo che da `@Using` le funzioni chiamabili diventano due: `end` e `dispute`.

Il programma può transitare di stato anche senza che venga richiamata una nuova funzione: alla fine del corpo di una funzione possiamo infatti richiamare un **evento**. Le operazioni specificate all'interno verranno eseguite in un preciso momento che è possibile programmare, solo se ci troveremo in uno specifico stato.

Vediamo ora in dettaglio la funzione `pay`: Se il `Borrower` è autorizzato a pagare, quindi se ci troviamo nello stato `@Payment`, può chiamare la funzione con argomento un asset `h`, se questo corrisponde al costo allora eseguiamo le operazioni. `h` viene spostato nell'asset `wallet`, `code` viene inviato a `Borrower` che quindi potrà utilizzare la bicicletta, intanto nessuno può accedere all'asset `Wallet` dove è stato spostato `h`.
A questo punto definiamo l'evento che si attiverà una volta trascorso il tempo `rentingTime` e passiamo allo stato `@Using`. Se passato il `rentingTime` lo stato sarà ancora `@Using` (in altre parole se nel frattempo nessuno ha chiamato le funzioni `end` o `dispute` passando direttamente a `@Return` o `@Dispute`), verranno eseguite le operazioni specificate all'interno dell'evento: verrà inviata l'informazione di fine del prestito a `Borrower`, per poi passare allo stato `@Return`.

Vediamo in breve cosa può succedere da `@Return`: `Lender` può decidere di chiamare la funzione `rentalOk`: il contenuto di `wallet` verrà spostato verso

`Lender` e finiamo allo stato `end`.

In caso invece di problemi durante l'utilizzo o al ritorno, può essere chiamata la funzione `dispute`, il messaggio `x` verrà inviato a tutti i party (con il comando `x->_`) e passiamo allo stato `@Dispute`.

A questo punto `Authority` potrà chiamare `verdict`: nel field `x` passerà il messaggio da inviare ai due party, nel field `y` sarà presente la parte di `wallet` che dovrà essere spostata verso `Lender`, la restante parte verrà spostata verso il `Borrower`, per infine transitare allo stato `End`.

Capitolo 2

L'interfaccia grafica

2.1 Obiettivi dell'interfaccia

Per quanto Stipula sia un linguaggio più astratto di un completo linguaggio di programmazione, scrivere contratti direttamente richiede delle conoscenze in campo informatico, non scontate per chi ha necessità di scrivere un contratto. Per questo la necessità di un'interfaccia facile da utilizzare[1].

2.2 Vantaggi della programmazione visuale

L'utilizzo di un'interfaccia grafica per un linguaggio di programmazione può essere di grande aiuto per rendere più accessibile il processo di scrittura di codice a chi non ha una formazione in informatica. Se intuitiva e ben progettata può ridurre significativamente la curva di apprendimento per l'uso di nuovi linguaggi di programmazione, permettendo ai nuovi utenti di acquisire competenze in modo più rapido ed efficiente a livello di **sintassi**, **semantica** e **pragmatica**[2].

Inoltre, un'interfaccia grafica può rendere più semplice la correzione di errori e la modifica del codice, facilitando la comprensione della struttura

del programma. Una visione più organizzata porta una migliore capacità di problem solving negli utenti[3].
Per un linguaggio di programmazione non rivolto a dei programmatori come Stipula, l'implementazione di un'interfaccia grafica può rappresentare a maggior ragione un passo in avanti nel rendere più fruibile l'accesso alla scrittura di codice.

2.3 Funzionalità

L'interfaccia permette di scrivere un qualunque contratto, senza scrivere direttamente del codice. Quello di cui l'utente si può occupare è solamente aggiungere e riempire campi. In tempo reale il programma tradurrà in codice Stipula quello che l'utente aggiunge e scrive.
Al caricamento della pagina l'utente potrà iniziare a scrivere partendo da un contratto vuoto, non c'è necessità di creare un nuovo progetto. In caso l'utente voglia salvare il progetto, può scaricare un file `.json` contenente tutte le informazioni del contratto che sta scrivendo, per poterlo caricare in seguito con lo scopo di visualizzarlo graficamente o per modificarlo.
Una volta che l'utente ha terminato di scrivere può copiare il codice, oppure scaricarlo come file con estensione `.stipula` per eseguirlo direttamente.

2.4 Design

L'interfaccia si ispira a interfacce esistenti di programmazione a blocchi come Blockly[4] e Scratch[5]. A differenza di queste l'interfaccia per Stipula adotta uno stile più minimale e non utilizza funzionalità di drag and drop per inserire nuove azioni.

2.4.1 L'agreement e le prime dichiarazioni

Dalla figura 2.1 possiamo osservare come si presenta la prima parte dell'interfaccia, in alto è possibile caricare un progetto già iniziato, oppure salvare

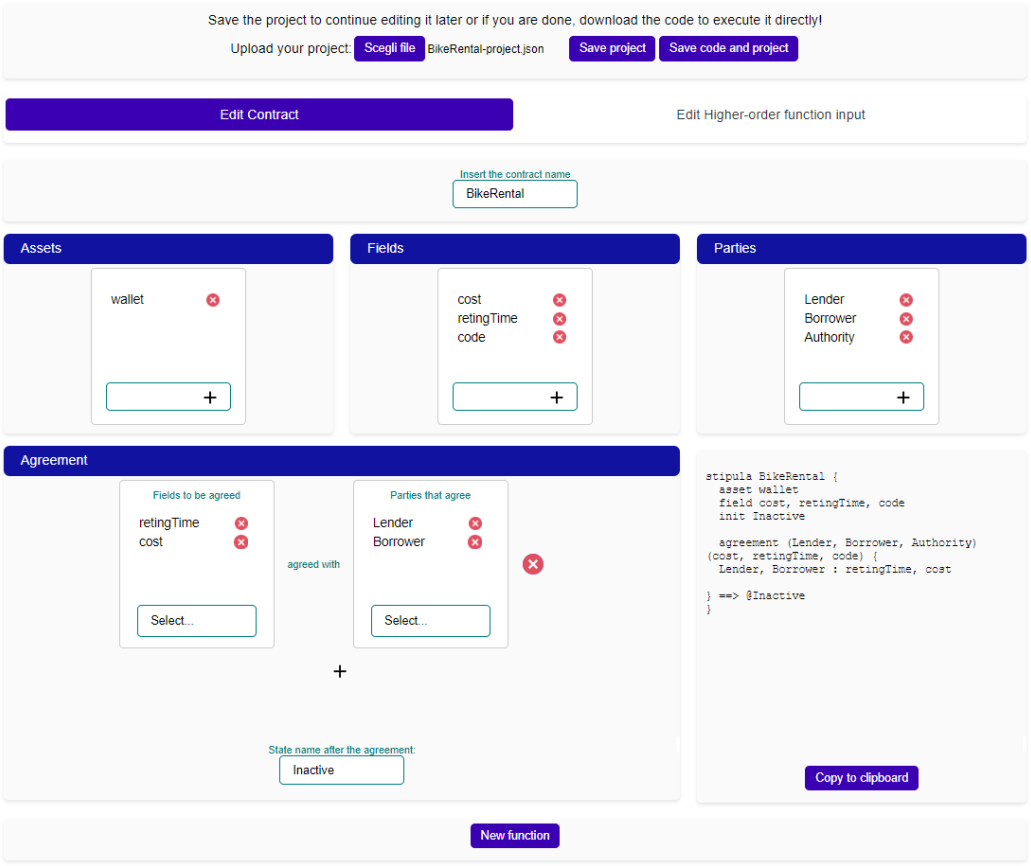

Figura 2.1: La prima parte dell'interfaccia.

il nostro attuale progetto.
Subito sotto possiamo inserire il nome del contratto. Ancora sotto i form per aggiungere le nostre dichiarazioni degli asset, dei field e dei party. Aggiungendo il nome del campo e premendo invio verrà aggiunto il campo alla lista sopra, di fianco ai campi è invece presente un bottone per eliminarli; questa soluzione grafica per aggiungere ed eliminare campi è coerentemente ricorrente in tutto il progetto.
Sotto a sinistra abbiamo l'agreement: è possibile aggiungere più accordi, ogni accordo è formato da una lista di field e una lista di party, che possiamo selezionare solo da quelli che abbiamo dichiarato sopra. Nell'ultimo campo dell'agreement impostiamo lo stato iniziale.
Sulla destra invece abbiamo il nostro contratto in linguaggio Stipula che verrà aggiornato automaticamente ad ogni modifica, in basso il bottone per copiarlo.
In fondo, la possibilità di aggiungere una funzione.

2.4.2 Le funzioni

L'interfaccia delle funzioni possiamo vederla divisa in due parti, nella parte superiore possiamo impostare:

- il nome
- da quali stati è possibile chiamare la funzione
- quali party possono chiamare la funzione
- verso quale stato transitare
- i parametri field e asset
- le condizioni necessarie ad eseguire la funzione

Ovvero tutto quello che in una funzione Stipula viene specificato fuori dal corpo.

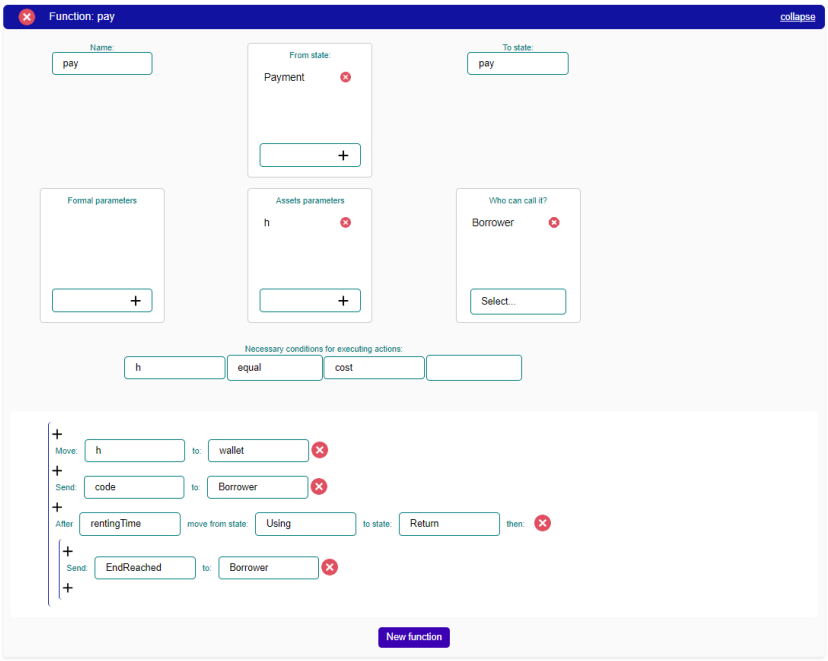

Figura 2.2: Funzione pay del contratto BikeRental.

Figura 2.3: La funzione può essere ridotta visualizzando solo le informazioni fondamentali per facilitare la navigazione.

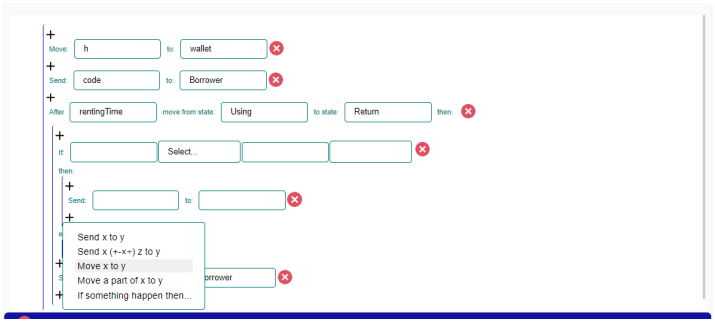

Figura 2.4: Funzione in dettaglio, con menù per aggiungere le azioni.

Nella seconda parte abbiamo il corpo della funzione, la scelta è stata di non utilizzare un sistema drag and drop come si avvalgono molti linguaggi di programmazione visuale; invece per aggiungere un'azione possiamo cliccare sul `+` e scegliere il tipo di azione da un menu pop-up che mostra solo le azioni che si possono aggiungere in quella riga. Ho giudicato questa modalità più intuitiva e veloce, anche dato il numero limitato del tipo di azioni che si possono aggiungere.
Dall'immagine 2.4 possiamo vedere come è possibile aggiungere un azione e come vengono visualizzati costrutti che creano blocchi annidati come `if then` e gli eventi.
L'azione che aggiungeremo avrà solamente dei campi di testo da riempire in modo da ridurre i possibili errori dell'utente, di fianco è ovviamente presente il bottone per eliminare la riga.

2.5 Implementazione

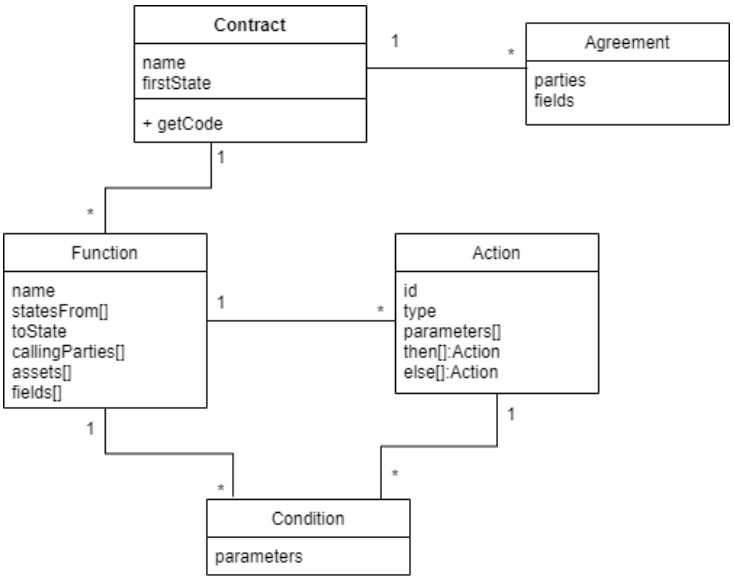

Figura 2.5: Struttura dell'oggetto contratto.

L'interfaccia grafica sviluppata è una webapp statica: viene eseguita completamente sul client, quindi il codice viene eseguito nel browser dell'utente, senza richiedere alcuna comunicazione aggiuntiva con il server. Il vantaggio di una webapp è sicuramente garantire la massima compatibilità con qualsiasi sistema operativo, senza la necessità di installare software aggiuntivo. Fattori che ho considerato importanti per uno strumento che dovrebbe aiutare la diffusione di un nuovo linguaggio.
L'applicazione web è stata sviluppata utilizzando React[6], una libreria JavaScript ampiamente utilizzata per lo sviluppo di interfacce utente.
React è stato scelto in quanto offre un approccio dichiarativo alla creazione di componenti UI, semplificando la creazione di interfacce complesse, con molti componenti che possono essere aggiunti in modo dinamico e che interagiscono tra di loro. I vari componenti possono essere riutilizzati ed aggiornati singolarmente, è quindi risultato utile per creare un'interfaccia molto dinamica, dove i vari componenti interagiscono tra loro e possono essere aggiunti e modificati dall'utente.
Ho utilizzato semplice CSS per lo stile, mantenendo l'aspetto minimale e senza elementi superflui.

2.5.1 Generazione del codice

La parte centrale del funzionamento dell'interfaccia grafica che ho sviluppato si basa sull'oggetto Contratto, figura 2.5. Questo oggetto contiene tutte le informazioni necessarie per definire un contratto in linguaggio Stipula, inclusi gli accordi e le funzioni. Questo permette inoltre di salvarlo in modo semplice come file `json` e di ricarlicarlo in un secondo momento.
Ogni volta che l'utente effettua una modifica o aggiunta tramite l'interfaccia grafica, l'oggetto Contratto viene modificato di conseguenza. In altre parole, l'oggetto è il tramite tra l'utente e il codice in linguaggio Stipula che viene generato.
Per trasformare l'oggetto Contratto in codice in linguaggio Stipula, ho sviluppato una funzione apposita che prende in input un oggetto contratto e

per ogni suo campo traduce i dati contenuti in codice Stipula ben formato con la corretta sintassi, per poi unire i codici dei vari componenti e creare l'intero contratto.

Capitolo 3

Higher-order Stipula

3.1 Introduzione al problema

Il presente capitolo si basa sulle informazioni riportate e analizzate nell'articolo [7].
I contratti nel linguaggio Stipula per come è stato descritto fino a questo momento sono immutabili. Quindi al momento della contrattazione di questo dovrebbero essere considerate tutte le variabili e imprevisti. Chiaramente è improbabile riuscire a prevedere tutte le possibili modifiche che un contratto potrebbe dover subire.
Una modifica può essere necessaria per vari motivi, ad esempio: il contratto potrebbe essere dichiarato nullo da un giudice perché viola la legge, oppure eventi imprevisti determinano la necessità di cambiare l'equilibrio economico del contratto.
Per permettere le modifiche è stata pensata un'estensione di Stipula, che ammette funzioni che prendono in input del codice che può modificare il contratto.

3.2 Le funzioni higher-order

La soluzione trovata è stata quella di introdurre un nuovo tipo di funzioni, che a differenza di quelle presentate nel capitolo 1.2.3, non presentano un corpo con le azioni da eseguire e uno stato dove transitare dopo l'esecuzione. Presentano invece un nuovo campo di input, dove al richiamo della funzione possiamo passare del codice che farà aggiunte e modifiche al contratto esistente.

```
@Payment Lender : hardship()[]([ X ])
```

Aggiungendo questa funzione al contratto, Lender potrà chiamare la funzione `hardship()[]([ codice... ])` passando nel campo `X` del codice per modificare il contratto.

3.3 Richiamare le funzioni

Quando si richiama una funzione higher-order, possiamo passare oltre ai parametri asset e field, un input di codice. Questo ci permette di modificare sostanzialmente il contratto: possiamo sovrascrivere o dichiarare nuovi asset, field, party e funzioni. Inoltre possiamo far eseguire delle operazioni nel momento della chiamata.
Vediamo un esempio di codice che può essere preso in input, che modificherà il contratto di esempio che abbiamo visto nel capitolo 1.2 in modo da aggiungere una nuova funzione per pagare, con la quale verrà pagata una tassa del 20% al governo:

```
1 parties Government
2 asset wallet2
3 field tax
4
5 @Payment Borrower : pay2()[h](h==cost){
6   (tax*h) -o h, Government
```

```
7     h -o wallet
8     code -> Borrower;
9     now + rentingTime >> @Using {
10      EndReached -> Borrower
11    } ==> @Return
12  } ==> @Using
13
14  {
15    0.2 -> tax;
16  _
17  } ==> @Payment
```

Nelle prime tre righe vediamo come vengono dichiarati nuovi party, asset e field.
Successivamente viene dichiarata una nuova funzione `pay2`, la quale sarà simile alla funzione `pay` che abbiamo visto al capitolo 1.2.3, ma una parte `tax` di `h` sarà spostata verso il nuovo party dichiarato `Government`.
Sotto alle funzioni abbiamo le azioni che dobbiamo svolgere quando la funzione `hardship` viene richiamata, in questo caso inviamo il valore 0.2 al field `tax` che come abbiamo appena visto servirà nella funzione `pay2`.

3.4 Implementazione nell'interfaccia grafica

Per poter scrivere i codici di input, ho pensato di aggiungere una nuova scheda, e creare un nuovo editor che permette di scrivere gli input. È possibile spostarsi tra l'editor del contratto e quello degli input attraverso una barra di navigazione a tab in alto, come si pùo osservare nell'immagine 3.2. In questo modo per chi non ha l'esigenza di scrivere un contratto con funzioni higher-order, l'esperienza è praticamente immutata.

In questo nuovo editor possiamo aggiungere nuovi input di codice, che saranno poi gli input che passeremo alle funzioni. La visualizzazione di essi

è molto simile a quella del contratto, ovviamente a differenza di questo, non è presente la parte di agreement, in più sono presenti le azioni che dovranno essere svolte alla chiamata. Per ogni nuovo input verrà visualizzato a destra il codice relativo che viene generato e modificato ad ogni cambiamento.
Ogni input è visualizzato all'interno di una scheda, ed è possibile ridurne la visualizzazione per facilitare la navigazione, in modo del tutto simile e coerente con quanto visto nelle funzioni.

Per quanto riguarda la dichiarazione di funzioni, è bastato aggiungere una casella di selezione che permette di decidere se una funzione è higher-order o meno, in caso non venga selezionata l'interfaccia rimarrà identica a quella presentanta nel capitolo 1.2.3. In caso selezioniamo la casella non verranno più mostrati i campi relativi alle azioni e allo stato da assumere dopo la funzione. Verrà invece data la possibilità di scegliere il nome dell'input tra gli input che abbiamo creato nell'altra scheda.

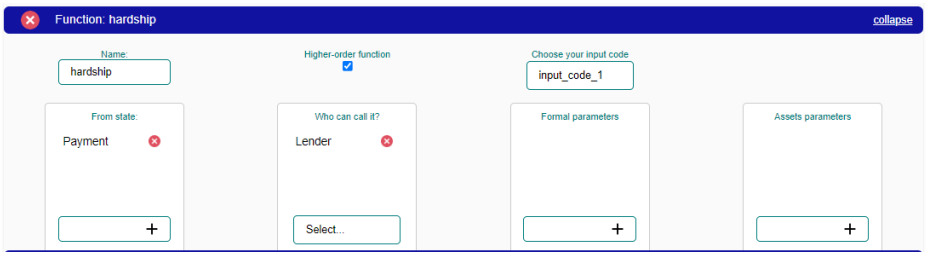

Figura 3.1: Funzione hardship nell'interfaccia grafica.

La generazione del codice è del tutto simile a quella dei contratti ed è stato possibile riutilizzare molte delle stesse funzioni. Il download del codice non restituisce più un solo file, ma una cartella compressa con all'interno il documento del contratto principale e uno per ogni input.

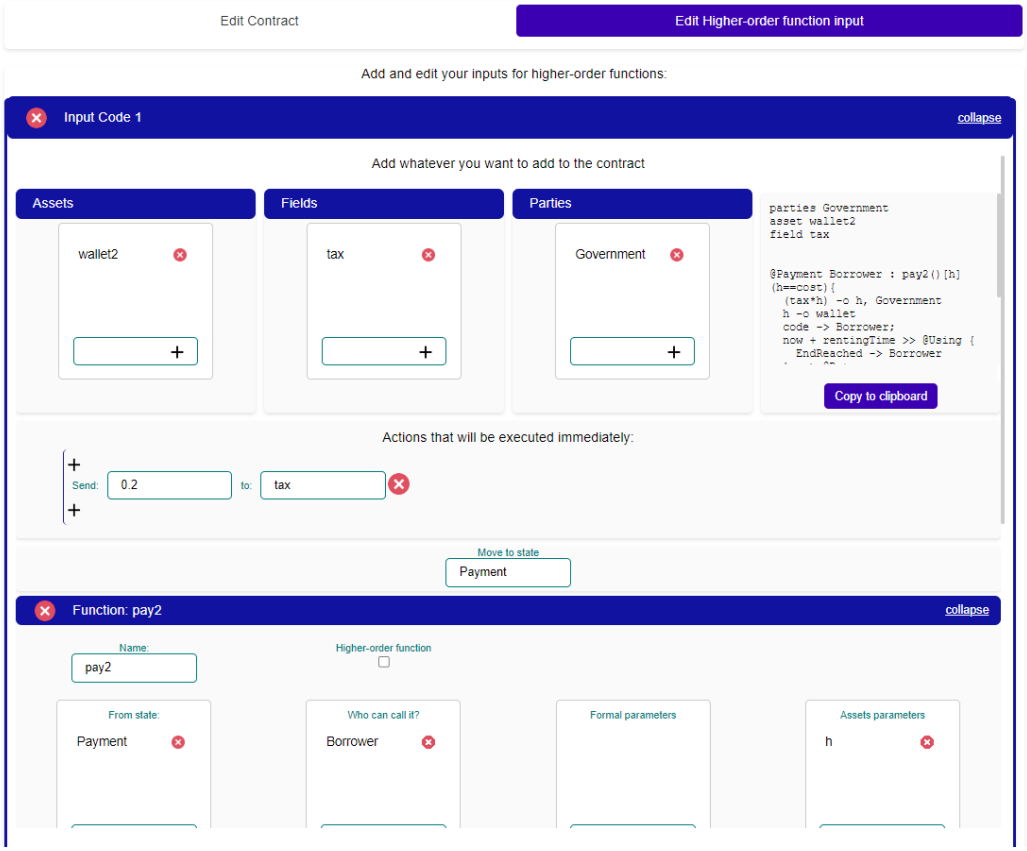

Figura 3.2: L'editor degli input.

Conclusioni

Dopo aver testato l'interfaccia sviluppata per il linguaggio di programmazione Stipula, si può affermare che il prototipo funziona in modo adeguato e consente di creare contratti legali effettivamente eseguibili. Questo rappresenta il risultato principale della tesi e dimostra l'efficacia dell'utilizzo di un'interfaccia grafica come metodo alternativo di scrittura di codice. Anche se l'utilizzo di questa facilita la stesura, è comunque necessario acquisire una conoscenza di base della struttura e il funzionamento a stati di un contratto Stipula. Ma sicuramente le conoscenze necessarie sono minori ed acquisibili in meno tempo.
Certamente, l'interfaccia potrebbe sempre essere ulteriormente integrata e migliorata. Ad esempio, sarebbe utile implementare un sistema in grado di rilevare eventuali errori di battitura e segnalare all'utente la loro presenza. Inoltre, potrebbe essere utile integrare una funzione che consente di ottenere un riassunto del contratto in un linguaggio naturale.

In conclusione, la ritengo sicuramente una buona base che soddisfa i suoi obiettivi, l'interfaccia è utile e funzionante per il fine di scrivere contratti eseguibili e ritengo abbia le potenzialità per rendere più accessibile il linguaggio.

Bibliografia

[1] Silvia Crafa, Cosimo Laneve, Giovanni Sartor, Adele Veschetti. *"Pacta sunt servanda: Legal contracts in Stipula"*, Science of Computer Programming, 2023, 225, pp. 1 - 21

[2] Repenning, Alexander. *"Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets"*, Journal of Visual Languages and Sentient Systems, 2017. pp. 68–91

[3] K.N. Whitley, *"Visual Programming Languages and the Empirical Evidence For and against"*, Journal of Visual Languages Computing, 1997, pp. 109-142

[4] *Blockly*, url:"https://developers.google.com/blockly"

[5] *Scratch*, url:"https://scratch.mit.edu/about"

[6] *React*, url:"https://it.reactjs.org/"

[7] Cosimo Laneve, Alessandro Parenti, Giovanni Sartor. *"Legal Contracts amending with Stipula"*, non ancora pubblicato