POLITECNICO DI TORINO

Laurea Magistrale in INGEGNERIA INFORMATICA

Tesi di Laurea Magistrale

Traduzione di contratti in smart contract tramite NLP

Relatori

Prof. VALENTINA GATTESCHI

Prof. FABRIZIO LAMBERTI

Candidato

ALBERTO BUTERA

APRILE 2022

Ringraziamenti

Questo spazio vorrei dedicarlo alle persone che, con il loro supporto, mi hanno aiutato a raggiungere questo traguardo.

In primis, vorrei ringraziare in modo particolare la mia relatrice, la professoressa Valentina Gatteschi, per i suoi indispensabili consigli e per la sua immensa pazienza e disponibilità.

Ringrazio infinitamente i miei genitori e mia sorella che mi hanno sempre sostenuto, appoggiando ogni mia decisione e permettendomi di affrontare serenamente tutto il mio percorso di studi.

Vorrei ringraziare Ilaria per essermi stata sempre accanto, per avermi dato supporto morale e dedicato il suo tempo quando ho avuto bisogno. Ci tengo a ringraziare il mio amico Giacomo per tutte le avventure che abbiamo condiviso in questi anni, per esserci sempre stato e per avermi sopportato in questi ultimi mesi difficili.

Un ultimo ringraziamento lo dedico agli amici e ai compagni di studi, per essermi stati vicini sia nei momenti difficili, sia nei momenti felici.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Oggi si sente sempre più spesso parlare di "Blockchain", una tecnologia innovativa per implementare un registro di contabilità condiviso e immutabile. Questa tecnologia permette inoltre di semplificare il processo di registrazione delle transazioni e la tracciabilità dei beni in una rete commerciale.
Come il termine stesso suggerisce, una Blockchain è formata da una catena di blocchi all'interno dei quali vengono registrate le transazioni e le informazioni relative ad esse (es. mittente, destinatario, data, etc.). Alla generazione di un nuovo blocco, questo viene prima verificato e successivamente collegato all'ultimo blocco della catena consolidandone la struttura. La verifica di un blocco viene effettuata tramite metodi specifici e serve a controllare che non vi siano errori che potrebbero compromettere le informazioni archiviate.
Tutti i partecipanti alla rete hanno accesso al registro distribuito, ma nessuno può modificare il valore di un record una volta inserito. Queste due caratteristiche permettono di garantire trasparenza e sicurezza dei dati inseriti rendendoli affidabili.
Ad oggi esistono diverse tipologie di reti blockchain, ognuna con alcune caratteristiche proprie che la differenziano dalle altre. Tra le reti blockchain più famose troviamo quelle pubbliche di Bitcoin e di Ethereum. La rete Bitcoin è famosa poiché è stata la prima in assoluto ad implementare questa tecnologia. La rete Ethereum, invece, è diventata famosa per aver introdotto il concetto di "Smart Contract".
Gli smart contract (in italiano: contratto intelligente), come qualsiasi tipo di contratto, specificano le condizioni di un accordo tra parti diverse. Tuttavia, vengono definiti "Intelligenti" perché le condizioni sono specificate ed eseguite sotto forma di codice anziché su un foglio di carta (o in formato digitale) conservato da un notaio.
Uno smart contract, prima di essere eseguito, va registrato all'interno di una blockchain. Questo gli permette di sfruttare i vantaggi che le blockchain offrono. Per esempio, il contratto una volta stipulato e registrato non può più essere modificato

e ciò permette una maggiore fiducia tra le parti. Inoltre, essendo la blockchain distribuita e accessibile a tutti i partecipanti della rete, il codice del contratto è sempre visibile e analizzabile. A questi vanno aggiunti anche i vantaggi introdotti dagli smart contract stessi. Ovvero, poiché il codice è pubblico ed eseguito in modo automatico, in teoria è possibile escludere l'intervento di un intermediario (ad esempio avvocati e notai). La relazione diretta comporta un risparmio dal punto di vista economico (oneri da versare all'intermediario) e di tempo.
La scrittura del codice di uno smart contract avviene tramite l'utilizzo di linguaggi di programmazione. Per la blockchain di Ethereum il più utilizzato è "Solidity", un linguaggio non troppo complesso e che permette di realizzare contratti sicuri.
A differenza dei linguaggi di programmazione popolari (C++, Java, etc.), Solidity soffre delle limitazioni imposte dalla blockchain. Prima di programmare uno smart contract, quindi, è necessario apprendere delle conoscenze specifiche che permettono di conoscere i limiti e le tecniche per superarli.
In generale quindi, anche se potenzialmente chiunque può programmare uno smart contract, la sua creazione prevede la padronanza di competenze tecniche che pochi possiedono. In ambito legale, ad esempio, non esistono (o sono casi rari) figure che hanno competenze informatiche avanzate.
Questo è uno dei motivi principali per i quali, ancora oggi, gli smart contract vengono utilizzati poco in campo legale nonostante il grande potenziale che offrono.
Il mondo delle blockchain e degli smart contract è un mondo nuovo, in costante crescita e miglioramento. Per incrementarne la crescita però bisogna renderlo accessibile a tutti, anche a chi non ha le giuste competenze.
Esistono diverse soluzioni già implementate che hanno provato a risolvere il problema della programmazione di uno smart contract. Tra queste, alcune prevedono la collaborazione tra figure dell'ambito IT (Information Technology) e dell'ambito legale, altre offrono dei metodi alternativi all'interfacciamento diretto con linguaggio di programmazione.
L'obiettivo di questa tesi è stato quello di effettuare un primo passo verso la creazione automatica di smart contract tramite la conversione diretta da linguaggio naturale a linguaggio formale.
In particolare, nel corso della tesi è stato realizzato uno strumento che sfrutta tecniche di "Natural Language Processing" per la comprensione e l'estrazione di elementi chiave di un contratto. Questi elementi vengono poi utilizzati per la generazione di parti di codice che, assemblate, vanno a creare uno smart contract. Il risultato è uno smart contract funzionante ed eseguibile che rispecchia nel miglior modo possibile le azioni descritte all'interno del contratto di partenza.
Il resto della tesi è strutturato come segue:

- nel Capitolo 2 sono stati analizzati gli strumenti attualmente disponibili per la generazione automatica di smart contract;

- il Capitolo 3 si occupa della descrizione delle tecnologie utilizzate per lo sviluppo di questa tesi;
- l'implementazione dello strumento è stata descritta al Capitolo 4;
- nel Capitolo 5 sono state fatte le valutazioni in base ai risultati ottenuti;
- infine le conclusioni sono riportate al Capitolo 6;

Capitolo 2

Stato dell'Arte

Come già menzionato nel capitolo introduttivo, uno degli aspetti fondamentali da tenere in considerazione quando si parla di smart contract è la scrittura del codice. Per permettere a questa tecnologia di espandersi e di essere utilizzata da più persone possibili bisogna abbattere il limite imposto dalle competenze tecniche. In generale tutte le persone conoscono ed utilizzano il linguaggio naturale per scrivere e comunicare. Quindi la soluzione ideale sarebbe quella di implementare una conversione automatica da linguaggio naturale a linguaggio formale.
In questo capitolo verrà fatta una panoramica delle soluzioni già esistenti ed infine verrà introdotta la soluzione proposta da questa tesi.

2.1 Contracts in code

Nel paper "Contracts in code?" [1] non viene presentata una soluzione concreta al problema, ma vengono affrontate le diverse difficoltà che subentrano quando si parla di conversione del linguaggio naturale in linguaggio machina. In particolare, l'articolo si concentra sulla traduzione di contratti ed infine suggerisce quale potrebbe essere l'approccio migliore al problema.
Il paper comincia analizzando le differenze tra i due linguaggi. Il linguaggio naturale, come l'inglese, è molto espressivo, ma a volte potrebbe risultare ambiguo o vago. Il significato delle parole o delle frasi spesso viene dedotto dal contesto in cui vengono inserite poiché ad una parola possono corrispondere significati diversi. Al contrario, il linguaggio formale è decisamente più semplice e preciso, creato in modo tale da eliminare le ambiguità. Questo però lo rende innaturale ed inespressivo.
Inoltre, il linguaggio formale è stato creato per assegnare determinati compiti ai computer, in modo tale da renderne automatica l'esecuzione. Quindi, prima di effettuare una conversione, è necessario comprendere quali frasi possono essere tradotte e quali, invece, non hanno il corrispettivo in linguaggio formale.

Dopo aver trattato l'argomento generale, il paper passa ad un'analisi più specifica della conversione di un contratto in smart contract.
All'interno di un contratto è possibile distinguere due tipologie principali di frasi:

- *Istruzioni primarie* che possono essere scomposte in sequenze di azioni e di risultati da raggiungere, ad esempio clausole contenenti termini di pagamento o obblighi di consegna;

- *Istruzioni secondarie* che descrivono come comportarsi nel caso in cui si presenti una disputa tra le parti, ad esempio clausole contenenti le leggi da seguire al verificarsi di determinati eventi.

In termini di esecuzione, soltanto le istruzioni primarie possono essere automatizzate poiché quelle secondarie richiedono l'intervento di un giudice che stabilisca come procedere. Tuttavia, non tutte le frasi che appartengono alla prima categoria possono essere tradotte perché "imprecise" per natura. Per esempio, alcune frasi possono contenere elementi impliciti che non permettono ad un computer di comprenderne il significato.
Alla luce delle precedenti considerazioni, il paper sostiene che una conversione diretta del contratto da un formato all'altro risulta molto complessa se non impossibile. Per questo motivo suggerisce la collaborazione di figure con competenze tecniche di entrambi gli ambiti, legale e IT, per la creazione di un livello intermedio che sia allo stesso tempo più semplice da comprendere e più preciso nel supportare il processo di codifica.
La rappresentazione intermedia potrebbe essere effettuata tramite l'utilizzo di linguaggi speciali, creati apposta, che siano più "formali" del linguaggio naturale ma anche più "naturali" rispetto ai linguaggi formali.
Sebbene l'obiettivo di questa soluzione sarebbe quello di rimuovere le ambiguità e di effettuare una conversione il più possibile realistica del contratto originale, rimane il problema della fiducia. Infatti, un programmatore potrebbe interpretare in modo errato la rappresentazione intermedia.
Eventuali errori generati al momento della programmazione rimarrebbero così ignoti fino all'esecuzione del programma o all'utilizzo di tecniche aggiuntive per l'individuazione di malfunzionamenti.

2.2 ADICO

Una delle prime soluzioni sviluppate per convertire in modo automatico i contratti in smart contract è spiegata nel paper di Franz et al. [2].
Questa fonda le basi su cinque componenti principali, le cui iniziali vanno a formare l'acronimo "ADICO". Di seguito sono descritti i componenti:

- **A**ttributes: è formato da caratteristiche e attributi di un attore;
- **D**eontic: individua la tipologia della frase identificando se quest'ultima è un obbligo, un permesso o un divieto;
- A**I**m: descrive l'azione o l'esito che caratterizza la frase;
- **C**onditions: elenca le condizioni alle quali deve sottostare l'azione. Se non sono specificate allora essa può avvenire in qualsiasi circostanza;
- **O**rElse: presenta un'azione alternativa nel caso in cui quella principale non può essere eseguita;

Tutte le frasi che descrivono un'azione possono essere scomposte in uno o più dei componenti citati sopra. Per esempio la frase *People must vote every four years, or else they face a fine.* può essere scomposta nel seguente modo:

People **(A)** *must* **(D)***vote* **(I)** *every four years* **(C)**, *or else they face a fine* **(O)**.

Annidando più strutture ADICO tramite operatori logici (*AND*, *OR*, *XOR*), si dà origine ad una struttura "Nested ADICO" che permette di rappresentare anche le frasi più complesse.
Per ogni componente, il paper descrive il costrutto corrispettivo in Solidity. Per esempio, il componente *Attributes* può essere rappresentato tramite *struct*. *Aims* viene rappresentato dalle *functions* e dagli *events*. Invece *Deontic* combinato con *Conditions* possono essere rappresentati tramite i *modifiers*. Scendendo di un gradino nel dettaglio, il componente *Aims* è composto da due sotto componenti, *object* e *target* che, in Solidity, corrisponderanno ai parametri delle funzioni.

Componente ADICO	**Costrutto Solidity**
Attributes	Structs
Deontic	Function modifiers
Aim	Functions, Events
Conditions	Function modifiers
OrElse	throw/codice alternativo

Tabella 2.1: Componenti ADICO mappati in Solidity

Tramite l'utilizzo del DSL (Domain Specific Language), mostrato in figura 2.1, vengono definite le regole per convertire una struttura ADICO in codice Solidity. Questo linguaggio associato al rispettivo algoritmo permette di automatizzare il processo di conversione.
L'algoritmo utilizzato può essere suddiviso in quattro passaggi. Al primo passo

```
actor      ⇐ <String literal> ;                                (* Attributes actor *)
prop       ⇐ property(<String literal>[, <String literal>]) ; (* Attributes property and
                                                                    optional type definition *)
deontic    ⇐ "may" | "must" | "must not" ;                      (* Deontic value *)
action     ⇐ <String literal> ;                                (* Aim action *)
obj        ⇐ object(<String literal>[, <String literal>]) ;    (* Aim object *)
tgt        ⇐ target(<String literal>[, <String literal>]) ;    (* Aim target *)
lOperator  ⇐ "AND" | "OR" | "XOR" ;                            (* Logical operators *)
condition  ⇐ IF(<Boolean expression>) |
               (condition lOperator condition) ;               (* Individual condition or
                                                                    condition combination *)
conditions ⇐ condition {lOperator condition} ;                 (* Conditions component *)
adico      ⇐ ADICO(
               A(actor[{, prop}]),
               D(deontic),
               I(action[, object][, target])
               [, C(conditions)]
               [, O(adico)]) ;                                 (* Individual ADICO statement *)
statement  ⇐ (adico {lOperator adico}) ;                       (* Complete statement *)
```

Figura 2.1: Domain specific language scritto in notazione EBNF

aggrega tutte le proprietà associate allo stesso attributo in una struttura. Successivamente estrae tutte le condizioni e le trasforma in *modifiers*, utilizzando il componente *deontic* inverte, se necessario, la condizione. Al terzo passaggio genera le funzioni associandogli i rispettivi *modifiers* generati al passo due ed infine genera gli eventi dai componenti *aims* che contengono la parola chiave "notify".
Il risultato ottenuto dalla conversione automatica è un template che contiene tutte le funzioni e gli elementi principali, ma che risulta essere incompleto. Per questo motivo non può essere eseguito direttamente senza prima essere stato revisionato e completato da uno sviluppatore.
Nella figura 2.2, è stato riportato un frammento del codice ottenuto dall'esecuzione dell'algoritmo. In questo caso, per esempio, manca la definizione del tipo del parametro passato alla funzione.

```
// TODO: Doublecheck parameter type definitions for function send
function send(*Define type* funds) msgValue$greaterEqualsPrice {
  seller.send(funds);
}
```

Figura 2.2: Frammento di codice estratto dall'output dell'algoritmo di conversione

2.3 Towards Declarative Smart Contracts

Prunell et al. [3] propongono una soluzione basata sull'utilizzo di template.
Il codice di uno smart contract può essere scomposto in fatti, eventi e logica di programmazione. L'idea di utilizzare dei template nasce dal fatto che per tutti i contratti appartenenti alla stessa tipologia, la logica di programmazione rimane la stessa, gli unici elementi che cambiano sono i fatti. Quindi, sulla base di questo principio, vengono realizzati degli "Standard Form Contract Template" (SFCT) ai quali viene affiancata la logica di programmazione corrispondente.
Per la generazione di un SFCT si parte da un documento esistente e se ne genera una versione digitale. A questa vengono aggiunti dei "markup" per identificare i campi che l'utente dovrà compilare. In pratica ad ogni campo da compilare viene associato un identificativo, il tipo di variabile attesa, le azioni collegate e altre informazioni. In figura 2.3, è stato rappresentato un esempio che mostra quanto descritto sopra. Nella parte destra della figura si notano una lista di opzioni selezionabili per la

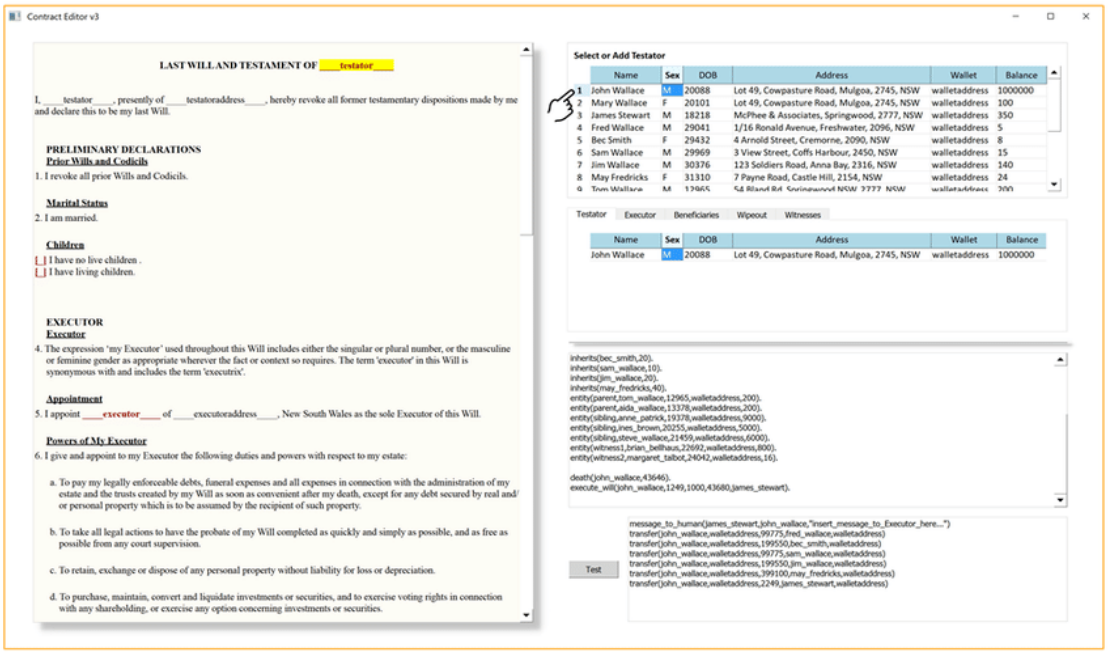


Figura 2.3: A sinistra il template con i campi da compilare e a destra un esempio di compilazione

compilazione del campo. Per questa parte, il paper prevede l'utilizzo di un "hardwer wallet", ovvero un'unità di memoria fisica, dove conservare tutte le informazioni personali e dal quale recuperarle al momento della compilazione.
Infine, il contratto digitale viene collegato alla logica di programmazione ad esso associata. Gli identificatori aggiunti permettono, durante la generazione dello

smart contract, di ottenere il tipo di informazione richiesto, dove trovarla e come processarla.
L'intero processo di creazione di uno smart contract si riduce così, dal punto di vista dell'utente, a semplici click del mouse.

2.4 Visual Smart Contract Design

La soluzione proposta nel paper di Mao et al. [4] si basa sull'utilizzo di blocchi contenenti parti di codice già implementate.
Questa soluzione incrementa il livello di astrazione rendendo più semplice la programmazione a chi non ha le competenze tecniche richieste.

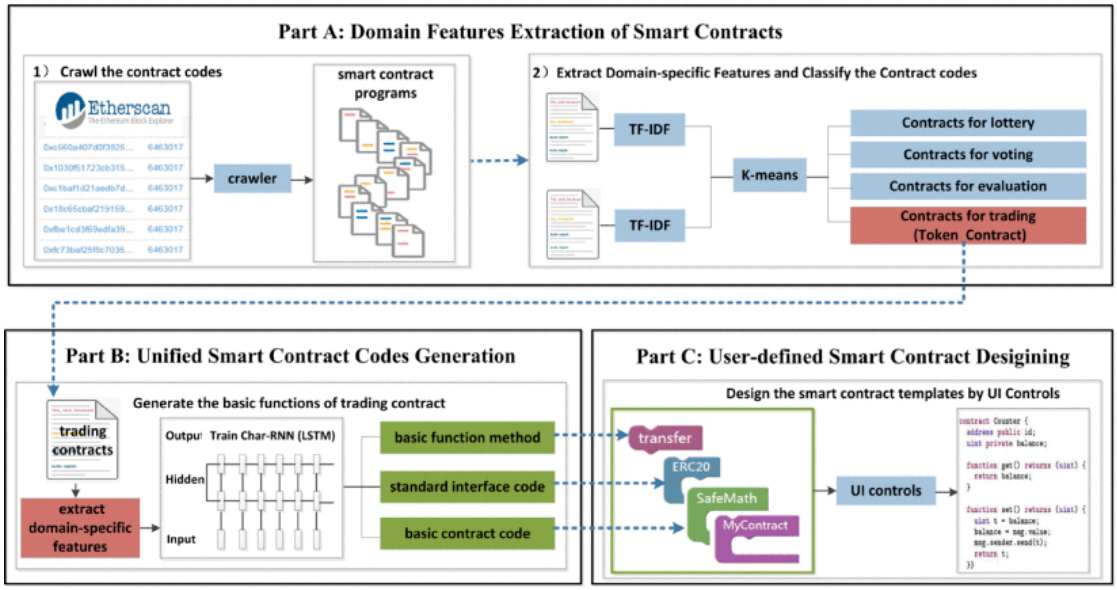


Figura 2.4: Architettura del sistema proposto

Il processo di generazione dello smart contract può essere suddiviso in tre punti:

1. Il sistema implementa un metodo per l'estrazione delle caratteristiche principali. Dopo aver recuperato una collezione di smart contract da Etherscan [5], ad essa vengono applicati gli algoritmi TF-IDF e K-means clustering per suddividere i contratti in categorie differenti in base alle caratteristiche estratte.

2. Tramite una versione migliorata dell'algoritmo Char-RNN, il sistema genera automaticamente, a partire dalle caratteristiche principali, le funzioni base di ogni categoria ottenuta al punto precedente. Mettendo insieme queste funzioni viene realizzato un template per ciascuna categoria. Il template può essere utile per evitare di dimenticarsi l'inserimento di funzioni principali.

3. Il sistema sfrutta la piattaforma Blockly [6], messa a disposizione da Google, per creare un'interfaccia web che permetta all'utente di personalizzare il template tramite "drag and drop" di blocchi. L'interfaccia viene creata in modo automatico e all'interno dei blocchi citati sopra vengono inserite le funzioni Solidity (figura 2.5). Una volta creata la struttura a blocchi, questa viene convertita in codice pronto per essere registrato su blockchain ed eseguito.

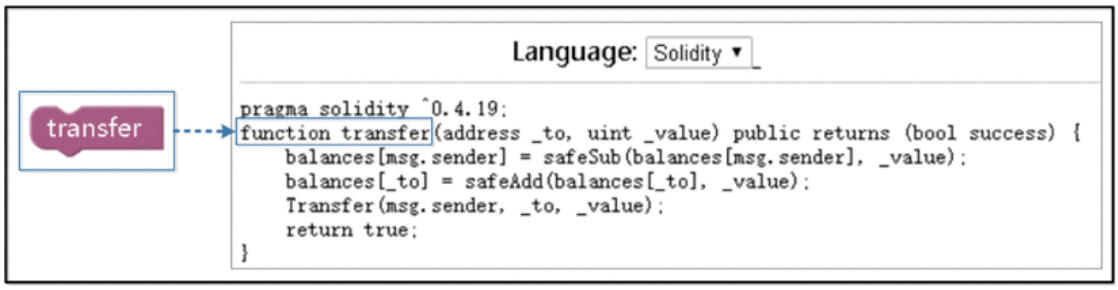


Figura 2.5: Contenuto del blocco "transfer"

2.5 FSolidM

Mavridou et al. [7] presenta lo strumento "FSolidM". Lo scopo dello strumento, oltre alla generazione automatica di uno smart contract, è quello di garantire lo sviluppo di un codice sicuro.
Solidity è un linguaggio di programmazione che può nascondere delle insidie anche per i programmatori più esperti. Molti studi hanno investigato sulle vulnerabilità degli smart contract individuandone diversi tipi e proponendo varie tecniche per risolverle.
Il paper si concentra sulla risoluzione di due tipologie di vulnerabilità:

- **Reentrancy Vulnerability**: questa è una tra le più famose tipologie di vulnerabilità, sfruttata anche nel famoso attacco "The DAO hack". In Ethereum, quando un contratto invoca una funzione che si trova all'interno di un altro contratto, il chiamante va in stato di attesa mentre aspetta che la chiamata termini. Questo permette al contratto chiamato, che potrebbe essere malevolo, di sfruttare lo stato di attesa per effettuare, a sua volta, una chiamata al contratto chiamante, ad esempio invocando una funzione *withdraw* più volte.

- **Transaction-Ordering Dependence**: questa vulnerabilità si basa sull'ordine delle transazioni effettuate. Se più utenti invocano delle funzioni all'interno dello stesso contratto, l'ordine con il quale queste vengono eseguite non può essere predetto. Di conseguenza, l'utente non può conoscere con certezza

lo stato in cui si trova il contratto quando verrà eseguita la sua funzione chiamata.

L'approccio proposto si basa sull'utilizzo del costrutto *state* di Solidity, il quale permette di definire degli stati (es. *Started*, *Finished*) che scandiscono un ordine cronologico con il quale eseguire le funzioni. Gli stati possono essere modificati tramite azioni invocate dagli utenti o da altri contratti stessi.
Per questo motivo gli smart contract possono essere rappresentati attraverso "Finite State Machines" (FSMs). Una FSM ha un numero finito di stati e un numero finito di transizioni che possono avvenire tra due stati. Il passaggio da uno stato all'altro garantisce che solo le funzioni ammesse dallo stato attuale possono essere eseguite.

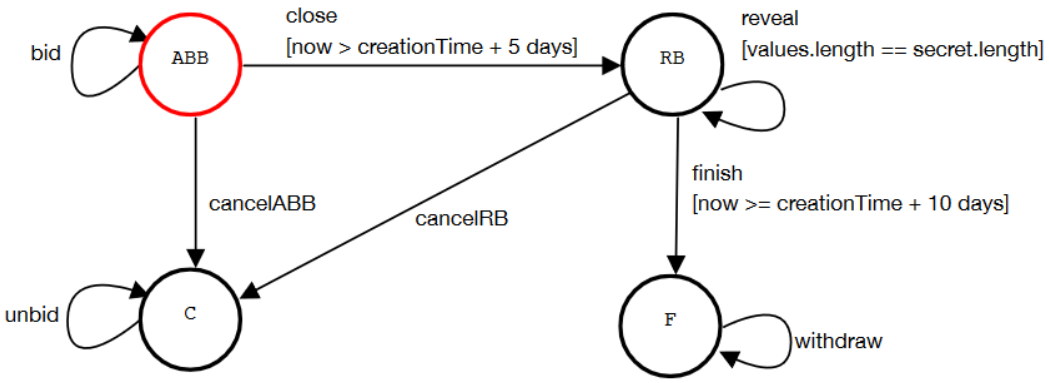


Figura 2.6: Esempio di FSM per un'asta.

Per generare in modo automatico lo smart contract basta che l'utente fornisca una FSM in forma grafica, come nell'esempio in figura 2.6. Ogni transizione viene implementata tramite una funzione in Solidity, gli stati vengono inseriti in una variabile di tipo *state* e vengono definite le variabili. Infine, il codice viene generato seguendo lo schema in figura 2.7. Le parti *StatesDefinition*, *VariablesDefinition*, *Plugins* e *Transition* verranno ricorsivamente sostituite con le parti di codice rispettive.
L'elemento *Plugins* contiene i plugins realizzati tramite il costrutto *modifier* per eliminare le vulnerabilità descritte sopra. Tra questi troviamo:

- **Locking**: sfrutta una variabile booleana per riprodurre la stessa logica utilizzata dai *lock* in altri linguaggi di programmazione. Permette l'esecuzione della funzione alla quale è associato solo nel caso in cui la variabile è impostata a *false*. Mentre una funzione è in esecuzione nessun'altra funzione potrà essere invocata.

- **Transition Counter**: sfrutta una variabile contatore per attribuire un ordine di esecuzione alle funzioni. In questo modo l'utente avrà la certezza che quando invocherà una funzione, questa verrà eseguita al momento stabilito e non in ordine casuale.

$$
\begin{aligned}
\mathit{Contract} ::= \ & \texttt{contract } \mathit{name} \ \{ \\
& \quad \mathit{StatesDefinition} \\
& \quad \texttt{uint private creationTime = now;} \\
& \quad \mathit{VariablesDefinition} \\
& \quad \mathit{Plugins} \\
& \quad \mathit{Transition}(t_1) \\
& \quad \ldots \\
& \quad \mathit{Transition}(t_{|\to|}) \\
& \}
\end{aligned}
$$

Figura 2.7: Struttura per la generazione del contratto

Lo strumento è stato messo a disposizione di tutti al link `https://cps-vo.org/group/SmartContracts`. Il framework mette a disposizione un editor grafico, per la realizzazione dell'FSM, e un editor Solidity per la scrittura diretta del codice.

2.6 LATTE

LATTE [8] è uno strumento che genera in modo automatico smart contract semplici senza dover scrivere codice Solidity. L'interfaccia sfrutta degli elementi grafici che verranno manipolati dagli utenti per la creazione del contratto.
Una caratteristica unica di questo strumento è quella di tracciare in tempo reale la somma di "Gas" da pagare per registrare il contratto sulla blockchain.
L'architettura dello strumento è così composta:

- **Visual Interface**: è composta *connection* e *build*, le due pagine principali. La prima è la pagina iniziale attraverso la quale l'utente si connette ad una blockchain valida. Una volta connesso, viene reindirizzato alla seconda pagina. La pagina *build* è composta da due componenti, *global state tab* (GST) e *initial state tab* (IST).
  Il componente GST permette la creazione dei dettagli di uno smart contract che non fanno parte di nessuna funzione (es. eventi e entità). Il componente IST, invece, si occupa della composizione delle funzioni. Per ogni funzione

vengono generati tre tab. Nel primo tab vanno inseriti i parametri della funzione, nel secondo vanno inserite le condizioni che devono essere soddisfatte prima dell'esecuzione e nel terzo viene inserito il corpo della funzione. Il corpo della funzione deve essere generato tramite un sistema "drag and drop" di componenti messe a disposizione dallo strumento.

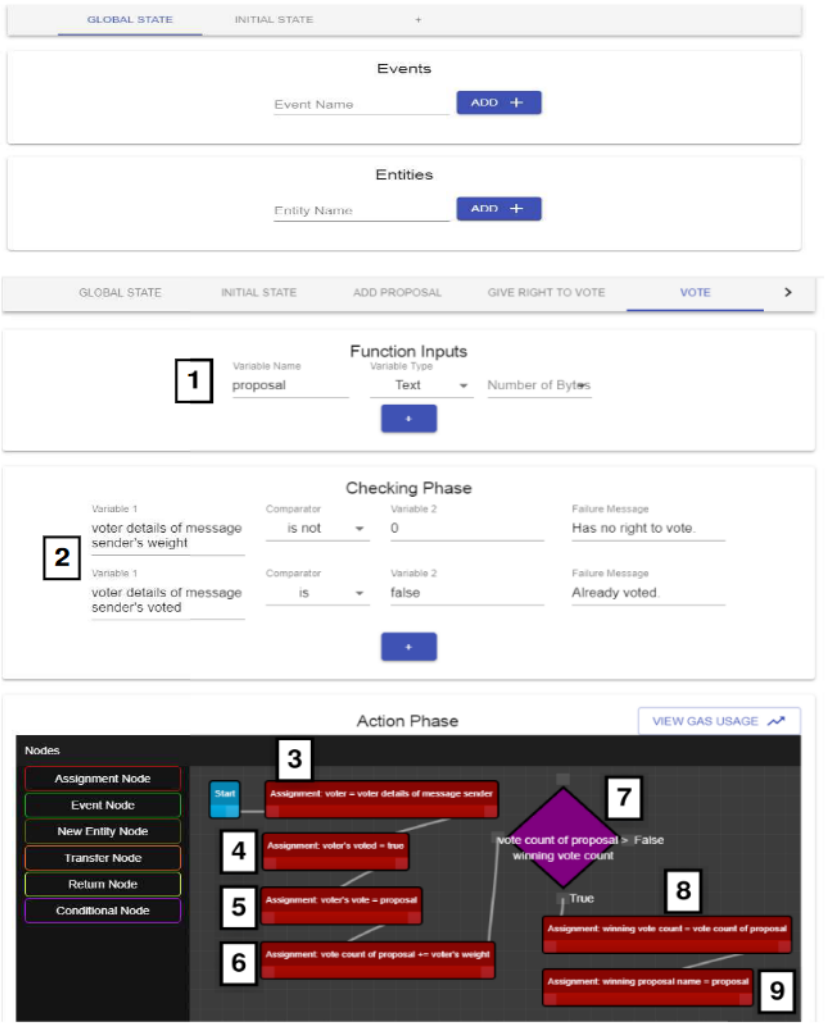


Figura 2.8: Interfaccia grafica di LATTE

- **Contract Manager**: questo modulo permette di salvare il contenuto del pagina *build* in un file JSON e di ricaricarlo in qualsiasi momento. Inoltre, questo modulo è responsabile della registrazione dello smart contract sulla blockchain. Se la registrazione è avvenuta con successo, vengono restituiti i valori di riferimento del contratto (es. indirizzo).

- **Code Generator**: questo è il modulo che si occupa della generazione del codice Solidity. Esso è formato da due componenti, *BuildParser* e *CodeBuilder*. Il *BuildParser* è responsabile della traduzione del grafo, generato tramite l'interfaccia grafica. Con una prima scansione si selezionano le variabili e si inseriscono all'interno di una *lookup table*. Successivamente viene fatta una seconda scansione convertire i nodi presenti in codice Solidity. Il *CodeBuilder* utilizza i valori ottenuti dal *BuildParser* per la generazione del codice completo.
- **Storage Advisor**: questo modulo avvisa gli utenti quando è possibile cambiare il tipo di variabile per risparmiare sulla quantità di "Gas" richiesto. Permette all'utente di definire quanti byte o bit devono essere utilizzati per memorizzare una variabile.
- **Gas Tracker**: questo modulo si occupa di tracciare, durante la generazione del funzioni, la quantità di "Gas" necessaria.

2.7 CML

Wohrer et al. [9] propongono un "Domain Specific Language" (DSL), per la programmazione degli smart contract, chiamato "Contract Modeling Language" (CML). Un DSL è un linguaggio di programmazione che ha espressività limitata e che viene utilizzato in un ambito specifico.
Gli obiettivi del CML sono:

1. Permettere la definizione degli elementi principali di un contratto.
2. Essere facile da comprendere attraverso l'utilizzo di elementi simili al linguaggio naturale.
3. Migliorare la produttività e semplificare la codifica di un contratto.
4. Poter essere convertito in codici eseguibili su diverse blockchain.

Il costrutto principale di questo linguaggio è la struttura *clause*. Una *clause* permette di definire le azioni principali descritte nei contratti ed è definita da almeno tre parti: attore, azione e deontica. Le *clause* possono essere definite anche da elementi opzionali quali oggetto dell'azione, condizioni, e trigger.
Le condizioni possono essere di tipo generale o temporale. Quelle temporali sono indicate dalla parola "due" seguita da una preposizione temporale (es. after, before) e da un trigger. In figura 2.10 è rappresentata la struttura di una *clause* in CML.
Ogni costrutto di CML viene poi mappato in Solidity per poter generare il contratto finale in modo automatico. Le variabili sono mappate in *struct*, funzioni e condizioni invece sono mappate rispettivamente in funzioni Solidity e *modifier*.

clause ID
[due [within RT] **[every** RT **from** AT **to** AT] **(after|before)** TRIGGER]
[given CONDITION]
party ACTOR
(may|must) ACTION {**(and|or|xor)** ACTION}

Trigger: AT | ClauseTrigger | EventTrigger | ActionTrigger
ClauseTrigger: **clause** ID **(fulfilled|failed)**
ActionTrigger: ACTOR **did** ACTION
EventTrigger: **event** ID

RT...Relative Time, AT...Absolute Time

Figura 2.9: Struttura di *clause* in CML

Dopo aver convertito tutti gli elementi, per completare la generazione dello smart contract viene aggiunto del codice di supporto (es. librerie) che renderà il contratto funzionante ed eseguibile.

```
namespace cml.examples

import cml.generator.annotation.solidity.*

@PullPayment
contract SimpleAuction
   Integer highestBid
   Party currentLeader
   Party beneficiary
   Duration biddingTime

   clause Bid
      due within biddingTime after contractStart
      party anyone
      may bid

   clause AuctionEnd
      due after contractStart.addDuration(biddingTime)
      party beneficiary
      may endAuction

   action init(Duration _biddingTime, Party _beneficiary)
      biddingTime = _biddingTime
      beneficiary = _beneficiary

   action bid(TokenTransaction t)
      ensure(t.amount > highestBid, "There already is a higher bid.")
      caller.deposit(t.amount)
      if (highestBid != 0)
         transfer(currentLeader, highestBid)
      currentLeader = caller
      highestBid = t.amount

   action endAuction()
      transfer(beneficiary, highestBid)
```

Figura 2.10: Esempio di contratto in CML

2.8 SmaCoNat

Anche il paper Regnath et al. [10] propone "SmaCoNat", un DSL per semplificare la programmazione degli smart contract.
Il linguaggio SmaCoNat è stato realizzato cercando di riprodurre una sintassi che rimandasse al linguaggio naturale. Per esempio, è stato limitato l'utilizzo di simboli e parentesi, elementi che complicano la comprensione del codice ad un utente alle prime armi.
A differenza di linguaggi di programmazione come il C, la struttura di SmaCoNat è stata suddivisa in più sezioni:

- **Heading**: contiene il linugaggio utilizzato (SmaCoNat) e la sua versione.
- **AccountSection**: contiene tutto ciò che riguarda gli account, ad esempio il nome, alias, etc.
- **AssetSection**: contiene tutto ciò che riguarda gli asset.
- **AgreementSection**: qui si definiscono le azioni che verranno eseguite una volta che il contratto è stato firmato da tutte le parti.
- **EventSection**: qui si definiscono gli eventi da generare quando un asset viene trasferito.

Lo scopo della suddivisione in sezioni serve a rendere più ordinato, e quindi più comprensibile il codice. Inoltre, sapere come sono composte le sezioni, aiuta nella realizzazione di regole più semplici per la conversione in Solidity.
Poiché gli smart contract permettono di registrare account e scambiare asset in maniera globale, sono necessari degli identificatori globali. In generale, le blockchain utilizzano degli "hash" unici ottenuti tramite operazioni di crittografia. Questi identificatori, però, sono facilmente confondibili e troppo complessi per essere ricordati da un essere umano.
L'idea proposta dal paper è quella di rappresentare questi identificatori seguendo lo stesso principio dei domini nella rete Internet. Infatti, è più semplice ricordare un nome che un indirizzo IP.
Un account o un asset viene, quindi, identificato tramite un nome e una catena di nomi di chi possiede quell'informazione. Ad esempio, "licensekey.alice.company" identificherebbe la chiave di licenza emessa da dall'account "alice" emesso a sua volta dall'account "company" noto a livello globale.
In SmaCoNat, account ed asset vengono definiti tramite la parola chiave rispettiva seguita da un nome, un alias ed una catena di alias di account che conoscono quell'informazione. La catena si interrompe al raggiungimento di un alias noto a livello globale.

Il linguaggio definisce tre alias speciali per gli account *Self*, *Genesis*, *Anyone* e uno per gli asset *Input*. *Self* si riferisce all'account del contratto stesso. *Genesis* è l'account che conosce qualsiasi altra entità. *Anyone*, invece, si riferisce a qualsiasi entità ignota. Per quanto riguarda l'asset *Input*, questo si riferisce all'asset mandato al contratto.

```
Contract in SmaCoNat version 0.1.

§ Involved Accounts:
Account 'BarrierIn' by 'AComp' by Genesis alias 'BarrierIn'.
Account 'BarrierOut' by 'AComp' by Genesis alias 'BarrierOut'.

§ Involved Assets:
Asset 'TheCoin' by Genesis alias 'TheCoin'.
Asset 'ParkTicket' by Self alias 'Ticket'.
Asset 'OpenBarrier' by Self alias 'Open'.

§ Agreement:
Self issues 'Ticket' with value 42.
Self issues 'Open' with value 1.

§ Input Event:
if Input is equal to 'TheCoin' from Anyone
and if value of Input is equal to 0.3
then
   Self transfers 'Ticket' with value 1 to owner of Input.
   Self transfers 'Open' with value 1 to 'BarrierIn'.
   Self issues 'Open' with value 1.
endif

if Input is equal to 'Ticket' from Anyone then
   Self transfers 'Open' with value 1 to 'BarrierOut'.
   Self issues 'Open' with value 1.
endif
```

Figura 2.11: Esempio di contratto in SmaCoNat

2.9 Problematiche riscontrate e soluzione proposta

Nei paragrafi precedenti, sono stati descritti gli strumenti disponibili per la generazione automatica di uno smart contract. Essi possono essere divisi principalmente in due categorie.

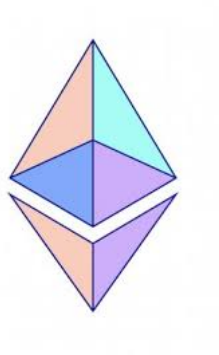
Nella prima rientrano le soluzioni che propongono di rimuovere la fase di programmazione attraverso l'uso di interfacce grafiche. Questa tipologia di soluzioni ha il vantaggio di eliminare il limite imposto dalla carenza di competenze tecniche. Allo stesso tempo, però, potrebbero vincolare l'utente nel caso in cui si faccia uso di template prestabiliti.
Nella seconda categoria, invece, vengono inseriti quegli strumenti che sfruttano dei linguaggi specifici creati per essere comprensibili sia alle macchine che agli esseri umani. In questo modo si garantisce agli utenti una maggiore libertà nella creazione del proprio smart contract rispetto alle soluzioni appartenenti alla prima categoria, ma si reintroduce la necessità di imparare un nuovo linguaggio.
Se esistesse uno strumento per convertire direttamente il linguaggio naturale in smart contract, verrebbero meno le problematiche riscontrate sopra. Il linguaggio naturale sostituirebbe i linguaggi intermedi e le interfacce grafiche non sarebbero più necessarie.
Questa tesi, come menzionato nel capitolo introduttivo, si pone come obiettivo quello di suggerire una possibile implementazione per uno strumento che sia in grado di eseguire questa conversione in modo automatico. In pratica, si propone una soluzione per eliminare al tempo stesso i vincoli generati dai template prestabiliti e le difficoltà legate alla programmazione.

Capitolo 3

Tecnologie utilizzate

In questo capitolo sono descritte le tecnologie utilizzate durante la fase di implementazione e di testing dello strumento.

3.1 Ethereum



Ethereum [11] è un universo basato su un unico computer (chiamato macchina virtuale Ethereum o EVM), sul cui stato è d'accordo tutta la rete Ethereum. Tutti i nodi della rete hanno una copia dello stato di questo computer. In più, ogni partecipante può effettuare delle richieste affinchè questo computer le esegua. Ogni volta che viene trasmessa una richiesta di questo tipo, gli altri partecipanti hanno il compito di verificarla, convalidarla ed eseguirla. Questa esecuzione porta ad un cambiamento di stato nell'EVM, che viene salvato e propagato su tutta la rete.
Le richieste di calcolo sono dette richieste di transazione. Il registro di tutte le transazioni e dello stato presente dell'EVM viene memorizzato sulla blockchain, che a sua volta è memorizzata e concordata da tutti i nodi.
I meccanismi crittografici assicurano che una volta verificate come valide e aggiunte

alla blockchain, le transazioni non possano essere successivamente manomesse. Gli stessi meccanismi assicurano inoltre che tutte le transazioni siano firmate ed eseguite con "permessi" appropriati (nessuno a parte Alice dovrebbe essere in grado di inviare asset digitali dal suo account).
La criptovaluta nativa di Ethereum è Ether (ETH). Lo scopo dell'ether è retribuire i nodi della rete per l'esecuzione delle transazioni. Questa ricompensa serve ad incentivare i partecipanti a verificare ed eseguire le richieste di transazione.
Ogni partecipante che manda una richiesta deve pagare un certo importo di ether (gas) che andrà a costituire la ricompensa. Tale ricompensa viene data a chiunque svolga la verifica, l'esecuzione e la propagazione del risultato.
L'importo di ether pagato corrisponde al tempo necessario per eseguire il calcolo. Questi costi, inoltre, servono ad impedire ai malintenzionati di intasare la rete di proposito richiedendo l'esecuzione di calcoli infiniti o di script che consumano una grande quantità di risorse.
Il codice utilizzato per la creazione delle richieste non viene scritto ad ogni transazione, viene inserito in programmi (smart contract) che vengono caricati sulla blockchain. Le transazioni all'interno di uno smart contract possono essere effettuate da chiunque, a patto che abbia i permessi per farlo. Con gli smart contract è possibile creare e distribuire delle app per gli utenti quali marketplace, strumenti finanziari, giochi, etc.

3.2 Solidity

Solidity [12] è un linguaggio di programmazione di alto livello per l'implementazione di smart contract. Questo linguaggio è ispirato ai linguaggi C++, Javascript e Python, utilizza le parentesi graffe per definire i blocchi di codice e ammette la tipizzazione delle variabili, l'ereditarietà delle classi e l'importazione di librerie.
Un contratto, in linguaggio Solidity, è l'insieme di codice (le *function*) e dati (lo *state*) che risiede ad un indirizzo specifico sulla blockchain di Ethereum.
I costrutti principali di questo linguaggio sono i tipi di variabile, gli *event*, i *modifier*

e le *function*.
In solidity vengono introdotti due tipi nuovi di variabile, *addresss* e *mapping*. Il primo tipo non permette nessuna operazione aritmetica, serve a memorizzare gli indirizzi dei contratti o degli account degli utenti. Il secondo tipo, invece, può essere visto come una tabella di hash virtualmente inizializzata cosi che possa esistere ogni genere possibile di chiave. Una variabile di tipo *mapping* richiede, al momento della definizione, un parametro che specifica il tipo di mappatura. Ad esempio,

mapping (address => uint) public balances;

la viariabile *balances* mappa il tipo *address* in variabili di tipo *uint*. Oltre alle variabili dichiarate dagli sviluppatori, esiste la variabile *msg*. *msg* è una variabile speciale con visibilità globale che permette di accedere alle informazioni della blockchain. Per esempio, *msg.sender* sarà sempre l'indirizzo di chi ha chiamato la funzione corrente.
Oltre alle variabili dichiarate dagli sviluppatori, esiste la variabile *msg*. *msg* è una variabile speciale con visibilità globale che permette di accedere alle informazioni della blockchain. Per esempio, *msg.sender* sarà sempre l'indirizzo di chi ha chiamato la funzione corrente.
Le *function* sono il costrutto che viene chiamato dagli utenti per l'esecuzione delle azioni, possono essere *public* (visibili dall'esterno) o *private* (visibili solo all'interno del contratto).
Una delle funzioni più utilizzate è la funzione *call* che permette di chiamare funzioni di altri contratti e di mandare ether agli indirizzi.

```
function sendViaCall(address payable _to) public payable {
    // Call returns a boolean value indicating success or failure.
    // This is the current recommended method to use.
    (bool sent, bytes memory data) = _to.call{value: msg.value}("");
    require(sent, "Failed to send Ether");
}
```

Figura 3.1: Esempio di funzione *call*

Il *constructor* è una funzione speciale che viene eseguita al momento della creazione del contratto e non può essere richiamata una seconda volta. Può essere utilizzato per inizializzare in modo permanente alcune variabili, ad esempio l'indirizzo di chi ha creato il contratto.
Un *event* è un messaggio di notifica che viene mandato tramite il metodo *emit* alle applicazioni. Un'applicazione, dopo aver effettuato una chiamata ad una funzione, può mettersi in ascolto di un di un determinato evento per avere conferma che l'operazione è avvenuta con successo. L' *event* può essere definito nel modo seguente,

event Sent(address from, address to, uint amount);

al quale vengono passati come parametri l'indirizzo di partenza della transazione, l'indirizzo di destinaizione e la somma mandata.
I *modifier* possono essere utilizzati per cambiare il comportamento di una funzione. Per esempio, un *modifier* può controllare in automatico una condizione prima che la funzione venga eseguita. Per utilizzare un *modifier*, il nome deve essere aggiunto agli argomenti della funzione da controllare.

```
// Modifiers are a convenient way to validate inputs to
// functions. `onlyBefore` is applied to `bid` below:
// The new function body is the modifier's body where
// `_` is replaced by the old function body.
modifier onlyBefore(uint time) {
    if (block.timestamp >= time) revert TooLate(time);
    _;
}
modifier onlyAfter(uint time) {
    if (block.timestamp <= time) revert TooEarly(time);
    _;
}
```

Figura 3.2: Due esempi di *modifier*

3.3 Remix

Remix [13] è un progetto per lo sviluppo di strumenti che usano un'architettura basata su plugin. Tra questi strumenti c'è Remix IDE, un'applicazione open source disponibile sia per web che per desktop. Questa applicazione, basata su javascript, è usata esclusivamente per lo sviluppo dei contratti ed è ideale per fare testing o per imparare a programmare in solidity e in vyper.
L'interfaccia principale è suddivisa in 4 moduli:

- **Icon Panel**: in questo modulo sono mostrate le icone dei plugin disponibili. Cliccando l'icona, verrà caricato il plugin all'interno del *Side Panel*.

- **Side Panel**: in questo modulo viene caricata la GUI della maggior parte dei plugin.
- **Main Panel**: in questo modulo viene scritto il codice del contratto.
- **Terminal Panel**: in questo modulo vengono mostrati i risultati delle interazioni con le GUI o possono essere eseguiti degli script.

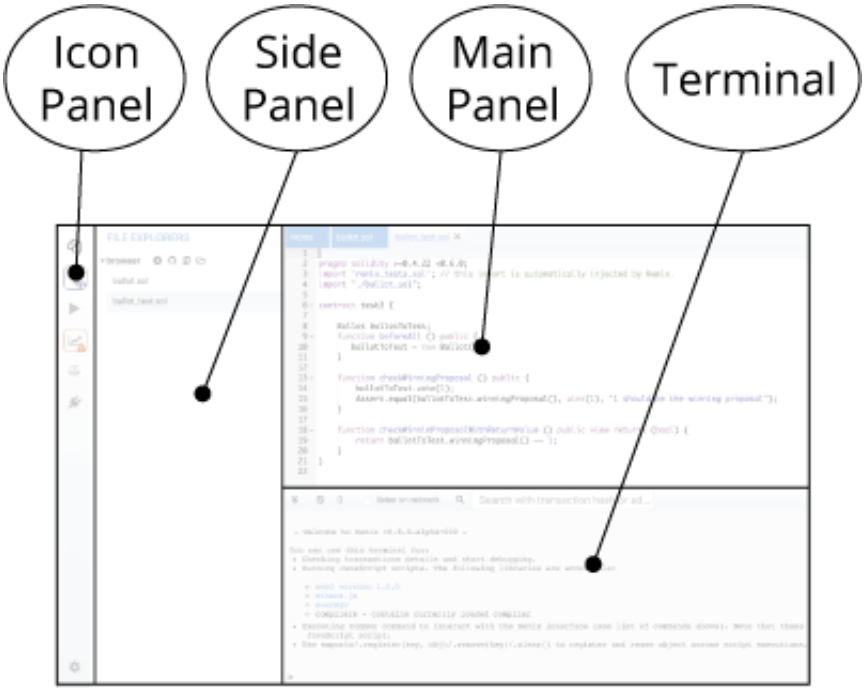


Figura 3.3: Interfaccia principale di Remix IDE

Come già detto, Remix IDE offre la possibilità di utilizzare due ambienti di sviluppo differenti, a seconda se si vuole utilizzare solidity o vyper. In base all'ambiente di sviluppo scelto, vengono caricati i rispettivi plugin.
Attraverso il **Plugin Manager** è possibile gestire i plugin in modo tale da caricare soltanto quelli necessari (mantenere dei plugin che non vengono utilizzati potrebbe rallentare l'applicazione).
I plugin più utilizzati sono il compilatore, il debugger, il Solidity Static Analysis e il plugin per la registrazione su blockchain e l'esecuzione.
Il compilatore segnala eventuali errori e warning all'interno del codice suggerendo, spesso, cosa modificare per risolverli.
Il plugin per la registrazione su blockchain può essere utilizzato solo se il contratto è stato compilato con successo. La prima operazione da fare è quella di scegliere l'ambiente nel quale registrare il contratto. Esistono tre ambienti disponibili:

- **JavaScript VM**: tutte le transazioni verranno eseguite in una blockchain di test all'interno del browser. Questo significa che ricaricando la pagina la blockchain verrà eliminata e ne verrà generata una nuova.
- **Injected Provider**: Remix si connette ad un provider web3 e utilizzerà gli account disponibili sul provider, per esempio Metamask.
- **Web3 Provider**: Remix si connette ad un nodo remoto. In questo caso sarà necessario fornire l'URL del provider (geth, parity, etc.).

Dopo aver scelto l'ambiente, appariranno gli account disponibili con la somma di ether disponibile per ciascuno. Prima di effettuare una transazione andrà selezionato l'account dal quale farla partire e andranno settati i campi *Value* (quantità di ether da mandare) e *Gas Limit* (la quantità massima di gas da utilizzare). Infine, la registrazione del contratto ritornerà l'indirizzo al quale è stato registrato ed un'interfaccia per utilizzare le funzioni implementate.
Il debugger mostra lo stato del contratto durante l'esecuzione di una funzione permettendo di muoversi in avanti e in dietro tra le istruzioni. Questo plugin è utile per rintracciare malfunzionamenti del codice.
Il plugin di analisi, infine, esegue delle analisi statiche, sul contratto compilato, per rintracciare tra i vari problemi, vulnerabilità o parti di codice sviluppate in modo errato.

3.4 Metamask

Metamask [14] è uno strumento per interagire in modo sicuro con applicazioni basate su Ethereum. In particolare, si occupa della gestione degli account e della connessione alla blockchain.
Metamask consente agli utenti di gestire i propri account e le loro chiavi in diversi modi, mantenendole isolate dal contesto del sito. Questa caratteristica offre una sicurezza maggiore rispetto al conservare le chiavi all'interno di un server centrale, o in un database locale, esponendole ad attacchi di massa.
Anche gli sviluppatori di applicazioni traggono vantaggio da questa caratteristica di sicurezza. Infatti, durante la programmazione non serve occuparsi della gestione utente, basta semplicemente utilizzare le API per connettersi con Metamask. Ogni volta che ad un utente viene richiesta la firma di una transazione, sarà Metamask

stesso ad occuparsene. Questo permette agli utenti di essere sempre informati sulle azioni che si effettuano con gli account e l'unico tipo di attacco possibile rimane il phishing individuale.
Per quanto riguarda la connessione alla blockchain, Metamask offre una connessione veloce sia con la rete di Ethereum sia con diverse reti di test. Questo consente agli utenti di iniziare ad effettuare transazioni senza dover sincronizzare un intero nodo. Metamask esiste sia come estensione per i browser sia come applicazione mobile. L'interfaccia principale (in figura 3.4) mostra l'account, il suo indirizzo e quanto ether ha a disposizione.

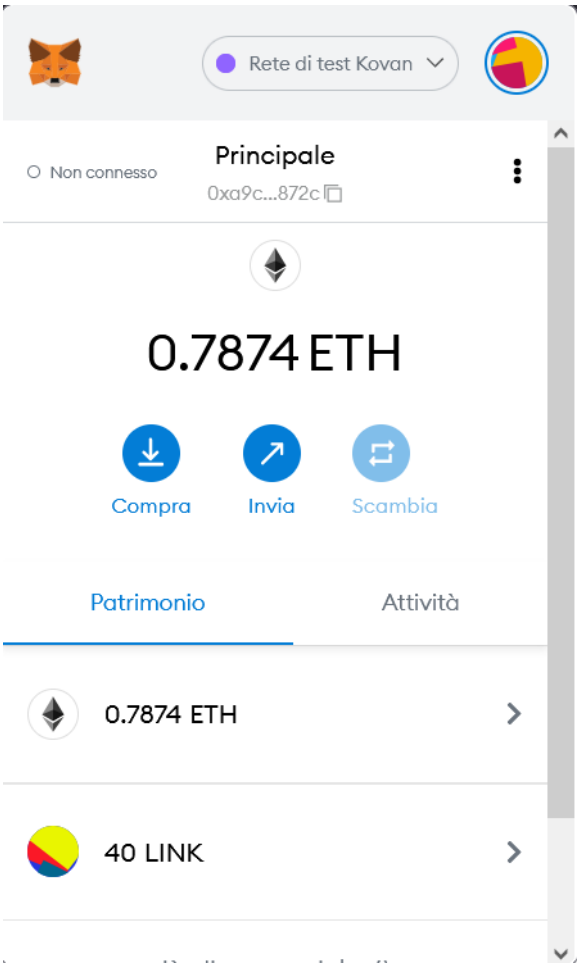


Figura 3.4: Interaccia principale di Metamask

Da qui è possibile ricevere ether, effettuare transazioni ed avere uno storico di tutte le transazioni effettuate. In alto a destra, invece, si possono selezionare il tipo di blockchain e l'account da utilizzare. Oltre allo scambio di ETH, Metamask offre la possibilità di acquistare ether direttamente, utilizzando altre valute (es. euro).

3.5 Python

Python [15][16] è un linguaggio di programmazione di alto livello, orientato a oggetti, adatto, tra gli altri usi, a sviluppare applicazioni distribuite, script o per connettere componenti diversi tra loro.
Python è un linguaggio multi-paradigma che ha tra i principali obiettivi quello di essere dinamico, semplice e flessibile.
Tra le caratteristiche principali, quelle che lo caratterizzano sono la tipizzazione dinamica e l'uso dell'indentazione per la sintassi dei blocchi di codice, al posto delle più comuni parentesi.
Il controllo dei tipi viene eseguito a runtime: una variabile è un contenitore a cui viene associata un'etichetta (il nome) che può essere associata a diversi contenitori anche di tipo diverso durante il suo tempo di vita. Un "garbage collector" si occuperà della liberazione e del recupero automatico della memoria di lavoro.
Python viene considerato come un linguaggio interpretato, ovvero non ha uno step di compilazione del codice. Questo velocizza le fasi di modifica, test e debug del codice sorgente.
Tutta questa versatilità e tutte queste agevolazioni di sviluppo, però, ne riducono di molto le prestazioni. Infatti, paragonato ai linguaggi compilati in modo statico (es. C) la velocità di esecuzione è nettamente inferiore, specialmente nel calcolo matematico.
Python supporta e usa estesamente la gestione delle eccezioni come mezzo per segnalare e controllare eventuali condizioni di errore, incluse le eccezioni generate dagli errori di sintassi.
Le eccezioni permettono un controllo degli errori più conciso e affidabile. Se l'interprete non rileva un'eccezione, stampa uno "stack trace" mostrando la parte di codice che ha generato un malfunzionamento.
Uno dei punti di forza di Python sono le librerie. Esso ha una vasta libreria standard e parecchie librerie sviluppate da terzi parti, le quali lo rendono adatto a molti impieghi.

3.6 Natural Language Toolkit

Natural Language Toolkit (NLTK) [17] è una delle librerie più importanti per la realizzazione di programmi che lavorano con il linguaggio naturale. Questa libreria è stata realizzata per Python e vanta più di cinquanta collezioni e risorse lessicali. Tra queste troviamo WordNet insieme ad una serie di librerie per la classificazione, tokenization, stemming, tagging, parsing e analisi della semantica. Molte librerie che implementano funzioni per processare il linguaggio naturale sfruttano NLTK. Graize alla guida pratica che spiega i fondamentali della computazione linguistica, più una dettagliata documentazione dell'API che spiega come interagire con le funzioni disponibili, NLTK è ideale per chiunque voglia sviluppare o imparare qualcosa inerente a questo mondo.

3.7 LexNLP

LexNLP [18] è una libreria Python open source che fornisce funzioni per il natural language processing e il machine learning di testo legale. La libreria include funzioni per le seguenti operazioni:

- Suddividere il testo in segmenti.
- Identificare parole chiavi, come titoli o intestazioni, all'interno dei documenti.
- Estrarre diversi tipi di informazioni come distanze e date.
- Estrarre entità nominali, come nomi di aziende, ed entità geopolitiche.
- Trasformare il testo in elementi per allenare modelli di machine learning.
- Costruire modelli "supervised" e "unsupervised".

Inoltre, LexNLP include dei modelli "pre-trained" basati su migliaia di testi presi da SEC EDGAR, un databse di documenti reali. Questa libreria è stata sviluppata sia per la ricerca accademica che per le applicazioni industriali.

3.8 Spacy

Spacy [19] è una libreria open source, di Python, per il Natural Language Processing (NLP). Una delle caratteristiche principali di questa libreria è la velocità con la quale riesce a processare testi di grandi dimensioni. Può essere utilizzata per l'estrazione di informazioni, per le principali tipologie di analisi che fanno parte di NLP o per processare testi per il deep learning.
Mentre alcune funzionalità di Spacy possono lavorare in maniera indipendente, altre richiedono delle pipeline già allenate da caricare, ad esempio per predire le annotazioni linguistiche (nome, verbo, etc.).
Per questa ragione, Spacy offre una serie di modelli, contenenti delle pipeline già pronte, che possono essere scaricati ed utilizzati.
I modelli differiscono tra loro per lingua, dimensione, velocita e memoria utilizzata. Non esiste un modello migliore rispetto ad un altro, la scelta è condizionata dalle esigenze, a seconda se si punta ad una maggiore efficienza o ad una maggiore precisione.
Una pipeline è composta da una serie di elementi in sequenza. In genere, le pipeline dei modelli disponibili includono un tagger, un parser, un lemmatizer e un entity recognizer. La pipeline, in input, accetta un testo che verrà processato in ordine dagli elementi che la compongono, e restituisce in output un Doc.

3.9 VerbNet

VerbNet [20] è la più grande collezione on-line attualmente disponibile di lessico dei verbi. Questa collezione è organizzata in modo gerarchico in classi e sottoclassi di verbi. Ogni classe è completamente descritta da ruoli tematici, dalla sintassi

delle frasi e da eventi che caratterizzano quella classe.
Una classe è composta da un gruppo di verbi che condividono la stessa struttura sintattica e gli stessi ruoli tematici. Ogni classe può contenere una o più sottoclassi per coprire al meglio più semantiche possibili. Una sottoclasse, infatti, contiene quei verbi che ereditano le caratteristiche della classe padre, arricchendole con eventuali informazioni aggiuntive.
L'obiettivo di VerbNet è quello di comprendere il senso delle frasi a partire dal verbo. I verbi appartenenti alla stessa classe danno origine a frasi con stessa sintassi e stesso senso. Per esempio, la classe "addict-96", rappresentata in figura 3.5, racchiude i verbi *to addict*, *to bias*, *to dispose*, *to incline*, *to predispose* e *to slant*. Questi verbi saranno caratterizzati dai ruoli tematici *Agent*, *Patient* e *Theme* e dalla stessa sintassi. Di conseguenza condivideranno lo stesso senso logico descritto dagli eventi semantici *Desire*, *Do* e *Cause*.

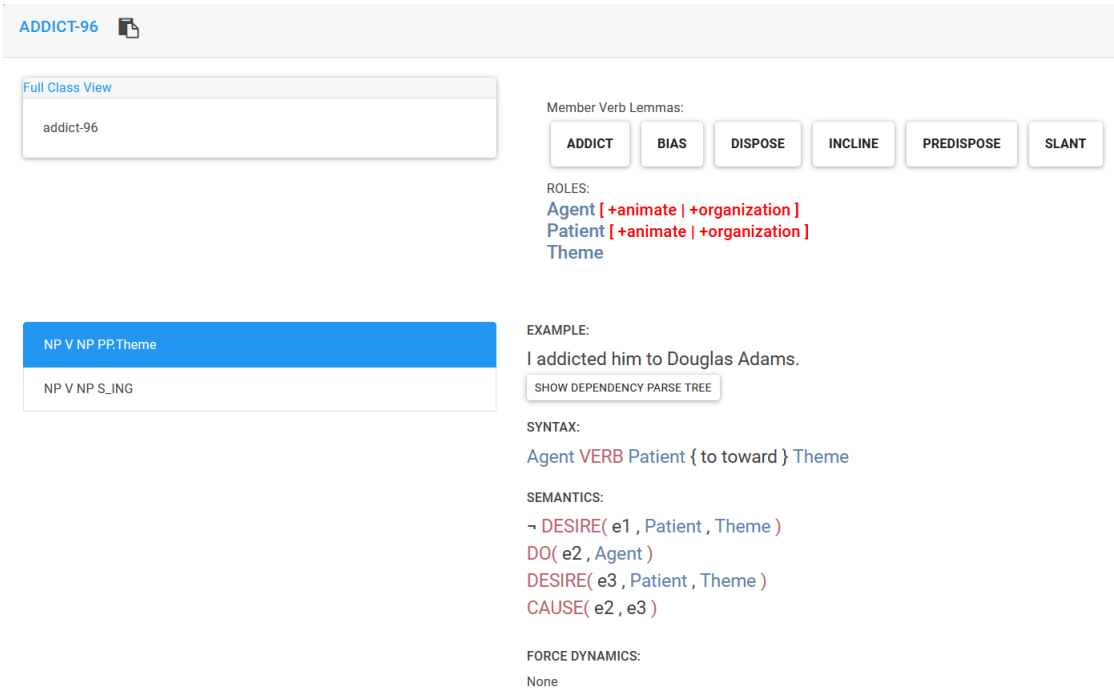


Figura 3.5: Classe "addict-96" di VerbNet

I ruoli tematici identificano gli elementi partecipanti all'azione descritta dal verbo definendone il modo in cui ne prendono parte. Per esempio, *Agent* identifica il soggetto del verbo, o *Patient* identifica colui che subisce l'azione. In caso di più partecipanti con lo stesso ruolo, viene affiancato ad esso un numero crescente, per esempio *Agent1* e *Agent2*. Ai ruoli possono essere applicate delle restrizioni per definire la tipologia del partecipante. Nell'esempio in figura 3.5, al ruolo *Agent*

vengono attribuite le restrizioni *+animate* e *+organization* per definire che quel ruolo verrà assegnato soltanto ad esseri animati o organizzazioni.

3.10 SemParse

SemParse [21] è uno strumento per l'analisi semantica del testo. Questo strumento fa parte del progetto SemLink [22], un progetto che ha come scopo quello di collegare insieme una serie di risorse lessicali, attraverso delle mappature. Questi collegamenti permettono di combinare informazioni differenti, fornite da risorse differenti, per ottenere dei risultati più accurati e dettagliati.
Attualmente il progetto collega le seguenti risorse:

- **PropBank**: Una collezione di circa un milione di parole in lingua inglese, annotate per l'identificazione dei ruoli.
- **VerbNet**: Un lessico che raggruppa i verbi in base alla loro sintassi/semantica.
- **FrameNet**: Un lessico basato sulla semantica delle parole.
- **WordNet**: Un lessico che descrive le relazioni semantiche (es. sinonimi) tra le parole individuali.

Quindi SemParse sfrutta tutte le risorse sopra elencate per produrre un'analisi semantica nel modo più accurato e dettagliato possibile.
Gli sviluppatori hanno messo a disposizione una demo dello strumento che implementa sia un'interfaccia grafica che una API per integrare lo strumento in altri progetti.
L'interfaccia grafica [23], una volta inserito il testo da analizzare, mostra tutte le informazioni ottenute. Il risultato può essere diviso in tre parti.
Nella prima parte viene mostrato il testo con le parole classificate come verbi evidenziate in blu. In questa parte è possibile selezionare uno degli elementi evidenziati per mostrare la rispettiva analisi semantica nelle parti seguenti.
Nella seconda parte sono elencate tutte le parti del testo con le rispettive etichette assegnate dal parser. Le etichette possono essere di quattro tipologie differenti:

- Etichetta classe: l'etichetta classe viene assegnata esclusivamente ai verbi e rappresenta la classe di appartenenza.
- Etichette blu: in blu sono rappresentate le etichette che definiscono i ruoli in base alla nomenclatura di VerbNet.
- Etichette viola: in viola sono rappresentate le etichette che definiscono i ruoli in base alla nomenclatura di PropBank e FrameNet.

- Etichette grigie: in grigio sono rappresentate le etichette che definiscono la tipologia dei così detti "modifier", ovvero quelle parti di testo che modificano il senso della frase. Possono essere di tipo temporale, modale, etc.

Nella terza parte si trovano gli eventi semantici, ovvero quegli eventi che caratterizzano il senso del verbo e più in generale della frase. Gli eventi si suddividono in due categorie: eventi principali ed eventi secondarti. Un evento principale è composto dall'insieme di uno o più eventi secondari ordinati in ordine cronologico. Per esempio, l'evento *CAUSE(e2, e3)* indica che l'evento secondario *e2* è la causa dell'evento secondario *e3*. Un evento secondario, invece, è composto dai ruoli visti nella parte precedente. Un esempio potrebbe essere l'evento *SPEND(Agent, Asset, Goal)* dove l'*Agent* spende un determinato *Asset* per il raggiungimento del *Goal*.

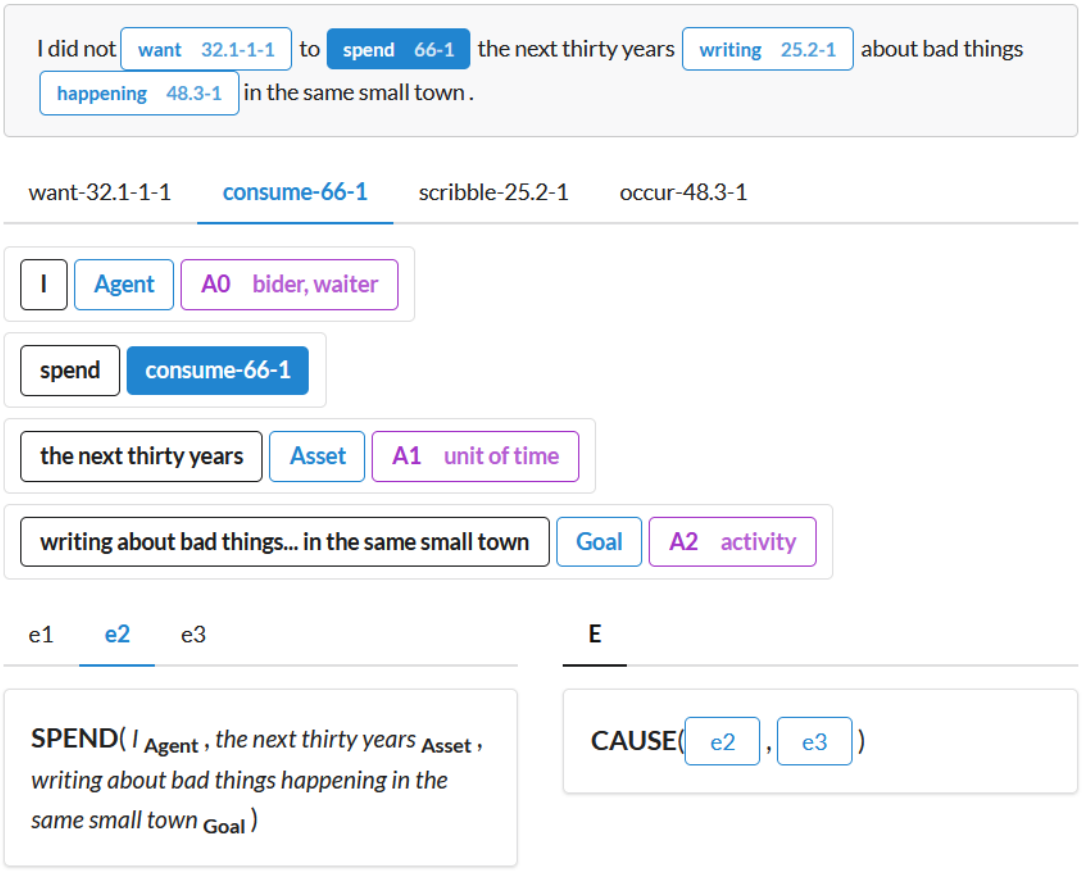


Figura 3.6: Interfaccia grafica di SemParse

Per utilizzare SemParse all'interno di un progetto bisogna fare riferimento alla API. Utilizzare la API è semplice, basta mandare una richiesta http contenente la frase

da analizzare e la demo restituirà, in risposta, un JSON contenente il risultato dell'analisi. Di seguito è stato riportato lo scheletro del JSON ritornato:

```
{
    "tokens": [...],                    # lista token della frase
    "props": [                          # lista verbi identificati
        0: {
            "sense":"",                 # categoria del verbo
            "mainEvent":                # evento principale
            {
                "eventIndex": ,
                "name": "",
                "predicates": [...],
            },
            "events": [...],            # lista eventi secondari
            "spans": [...],             # lista etichette attribuite
        }
    ]
}
```

Il risultato contiene, praticamente, gli stessi elementi descritti nella parte dell'interfaccia grafica, l'unico elemento del JSON non ancora visto è la lista "tokens". In questa lista vengono elencati i "token", ovvero gli elementi in cui è stato scomposto il testo, ancora privi di etichette.

Capitolo 4

Realizzazione dello strumento

Il linguaggio di programmazione scelto per la realizzazione dello strumento è stato Python. Come mostrato al capitolo 3.5, Python è un linguaggio molto versatile che si adatta bene a gran parte delle situazioni. In questo caso, per i problemi che si basano sul "Natural Language Processing" (NLP), Python è il linguaggio più utilizzato e quello con il maggior numero di librerie a disposizione.
Sebbene le librerie più famose offrano supporto per lingue differenti, tra le quali anche l'italiano, è stato scelto di utilizzare la lingua inglese. La lingua inglese, oltre ad offrire una libertà più ampia nella scelta delle librerie da usare, permette di ottenere dei risultati più accurati rispetto alla lingua italiana.
Il processo di traduzione (vedi figura 4.1) può essere suddiviso in due sottoprocessi. Il primo sottoprocesso si occupa di scomporre ed analizzare un contratto, scritto in inglese, per estrapolare i componenti principali. I componenti che vengono estratti sono gli stessi definiti dalla soluzione ADICO, descritta al capitolo 2.2, la quale stabilisce che una frase può essere scomposta in *attributes*, *deontic*, *aim*, *condition* e *orelse*. L'individuazione di questi componenti avviene tramite la scomposizione e l'analisi delle frasi.
Una volta che gli elementi sono stati estratti si passa al secondo sottoprocesso, ovvero la generazione dello smart contract. Per questa fase viene prima realizzato un template contenente il codice standard, comune a tutti i contratti. Successivamente vengono generate le funzioni e le condizioni sfruttando la mappatura proposta da ADICO per la traduzione dei componenti in costrutti Solidity. Durante questa fase, inoltre, vengono identificati gli attori del contratto per generare la parte di codice relativa alla loro registrazione. Infine, dall'unione di tutte le parti di codice generate, si ottiene uno smart contract completo e funzionante.
Nella prima parte di questo capitolo saranno spiegate le tecniche utilizzate per

l'elaborazione del linguaggio naturale, fondamentali per la comprensione della seconda parte. Nella seconda parte, invece, verranno spiegate nel dettaglio tutte le fasi che costituiscono il processo di traduzione.

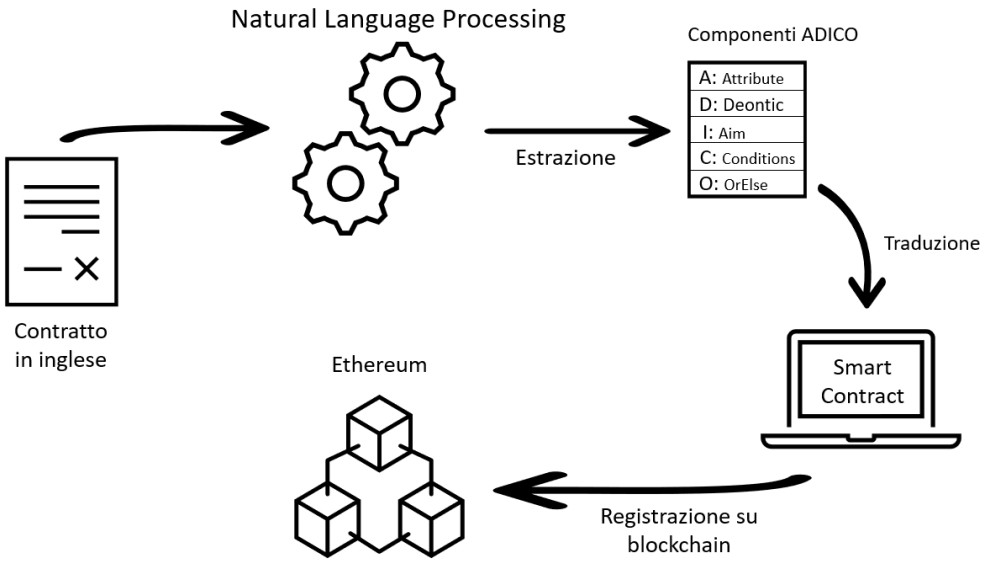


Figura 4.1: Architettura dello strumento

4.1 Elaborazione del linguaggio naturale

Il testo che deve essere elaborato, in questo caso, deriva da contratti reali scritti in lingua inglese.
Un contratto è un accordo tra due o più parti per costituire, regolare od estinguere un rapporto giuridico patrimoniale. In genere, nella prima sezione del contratto vengono identificate le parti (tramite nome, indirizzo, etc.) e vengono stabilite delle parole di riferimento agli elementi principali. Per esempio, spesso la parola "Term" viene utilizzata come riferimento alla data di fine contratto. Nella seconda sezione del contratto, invece, sono elencate le clausole contenenti obblighi, diritti e altre informazioni di tipo descrittivo.
Un contratto, dal punto di vista strutturale, può quindi essere inteso come un insieme di clausole che a loro volta sono formate da frasi. La frase, di conseguenza, è l'elemento base che dovrà essere processato.

4.1.1 Difficoltà nella comprensione del linguaggio naturale

Il linguaggio naturale ha come obiettivo quello di essere compreso dagli esseri umani, ha un vocabolario ricco e una quantità indefinita di modi per esprimere i concetti. Al contrario, i linguaggi formali, creati per essere compresi dalle macchine, hanno una struttura e un set limitato di parole chiave che limitano l'espressività, ma al tempo stesso rimuovono ogni sorta di ambiguità. Di conseguenza, permettere ad una macchina di comprendere il linguaggio naturale risulta uno dei compiti più difficili da eseguire.
Oltre all'ambiguità, anche la complessità delle frasi contribuisce a complicarne la comprensione. Il compito di una macchina è quello di eseguire delle azioni in modo automatico; quindi, necessita di istruzioni chiare e ben definite. Nel linguaggio naturale le frasi vengono arricchite con informazioni descrittive, che non alterano il senso generale della frase, per semplificare la comprensione agli esseri umani. Dal punto di vista delle macchine, però, queste informazioni aumentano le probabilità di commettere errori.
Per esempio le frasi,

Bob plays the piano.
Bob, who is the best musician in the world, plays the piano.

esprimono entrambe lo stesso concetto, nonostante la seconda contenga delle informazioni in più. La frase di esempio risulta essere ancora "semplice", ma nel linguaggio naturale una frase può essere complicata a piacere, mantenendo il significato generale inalterato.

4.1.2 Difficoltà nella traduzione dei contratti

Il paper "Contracts in code?", introdotto al capitolo 2.1, fa una panoramica delle difficoltà legate alla traduzione di un contratto in smart contract. Queste difficoltà nascono dall'utilizzo dei linguaggi di programmazione. Un linguaggio di programmazione serve ad implementare degli algoritmi che verranno eseguiti in modo automatico dalle macchine; quindi, possono essere programmate solo quelle parti del contratto che descrivono azioni e condizioni, per esempio pagamenti, consegne, invio di notifiche, etc.
Dunque, prima di procedere con la traduzione, un contratto deve essere filtrato in modo tale da rimuovere tutte quelle frasi che non potranno essere tradotte.

4.1.3 Scomposizione delle frasi

Una frase per essere tradotta va prima scomposta in modo tale da estrapolare i componenti ADICO. La scomposizione viene fatta sfruttando operazioni di elaborazione del linguaggio naturale quali analisi morfologica e lessicale ("part-of-speech"),

analisi sintattica, analisi semantica e diverse altre operazioni che agevolano la comprensione del testo. La collaborazione di queste operazioni permette di ottenere dei risultati soddisfacenti.
Per spiegare in modo chiaro i diversi tipi di analisi, però, è bene iniziare da frasi semplici.

Frase "semplice"

Nella lingua inglese, una frase viene considerata semplice quando questa è indipendente, priva di subordinate e composta esclusivamente da soggetto, verbo e complemento oggetto (se richiesto). Inoltre, la frase può essere espressa in forma passiva, ovvero il soggetto non è più colui che compie l'azione, ma colui che la subisce. Ritornando all'esempio citato prima, la forma passiva diventa,

Bob **(Attore/Soggetto)** *plays* **(Verbo)** *the piano* **(Oggetto)**.
The piano **(Soggetto)** *is played* **(Verbo Passivo)** *by Bob* **(Attore)**.

In questa tipologia di frasi, individuare i componenti ADICO presenti è molto semplice. A seconda della forma, se attiva o passiva, il soggetto può essere inteso come *attribute* o come *object* del componente *aim*, il verbo sarà la parte principale del componente *aim* e il complemento oggetto (forma attiva) o di agente (forma passiva) sarà il componente opposto del soggetto.

Analisi morfologica e lessicale

L'analisi morfologica e lessicale, anche nota come "part-of-speech", è l'analisi più di basso livello tra quelle di elaborazione del linguaggio naturale. Il suo obiettivo è quello etichettare le parole classificandole come nome, pronome, verbo e così via.
Prima di essere analizzata, una frase va scomposta attraverso un processo noto come "tokenizzazione". Tokenizzare un testo significa suddividerlo in unità minime di analisi (chiamate *token*), che possono essere costituite da caratteri, cifre, parole o porzioni di frasi. La suddivisione in token rende possibile tutti gli ulteriori step di analisi.
Il pacchetto Spacy, descritto al capitolo 3.8, offre la possibilità di utilizzare una pipeline già pronta per eseguire in modo automatico la tokenizzazione e la part-of-speach. Mandando in input la parte di testo da analizzare, il risultato ottenuto sarà un "Doc", ovvero una lista di token etichettati. Nella tabella 4.1 è stato riportato il risultato ottenuto dall'analisi della frase *Bob plays the piano.*. Questo tipo di analisi, preso singolarmente, aiuta poco nell'identificazione delle componenti ADICO, ma è fondamentale per poter effettuare l'analisi sintattica.

Token	Part-of-speech Tag
Bob	PROPN
plays	VERB
the	DET
piano	NOUN
.	PUNCT

Tabella 4.1: Risultato prodotto dall'analisi lessicale

Analisi sintattica

L'analisi sintattica si basa sul risultato ottenuto dall'analisi lessicale per individuare le dipendenze logiche tra i token. Queste dipendenze possono essere di tipo *nsubj* (soggetto), *dobj* (complemento oggetto), *aux* (ausiliare), etc.
Il risultato che si ottiene da questo tipo di analisi è un grafo di dipendenze che, nella maggior parte dei casi, ha come punto di partenza il verbo. In generale, l'analisi sintattica attribuisce ai token la tipologia di dipendenza che questi hanno con il padre (nel caso della radice la dipendenza sarà proprio *root*) e una lista di token figli. Tornando all'esempio di Bob, il risultato è mostrato in figura 4.2. In

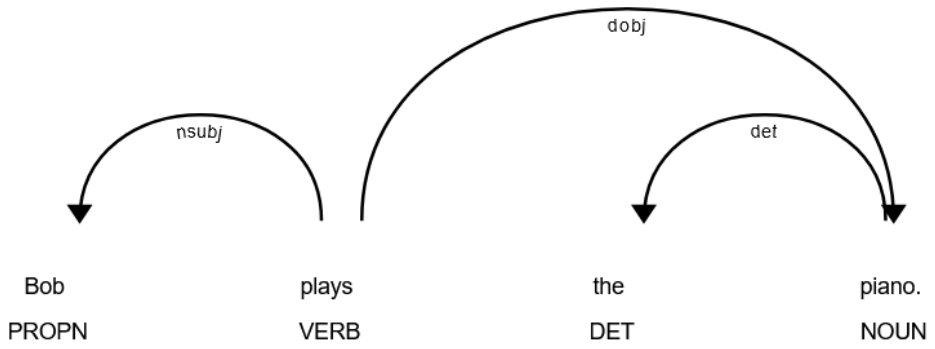


Figura 4.2: Grafo delle dipendenze

questo caso la radice del grafo è il verbo "pays" e ad esso sono collegati "Bob" con dipendenza *nsubj* e "piano" con dipendenza *dobj*. Il token "the", invece, è collegato a "piano" con dipendenza *det*. Se le frasi fossero tutte così semplici, questo tipo di analisi basterebbe ad individuare i componenti ADICO. Nel caso reale, però, è più probabile trovare frasi complesse, soprattutto all'interno dei contratti, e il risultato ottenuto dall'analisi sintattica diventa difficile da interpretare in modo automatico

o addirittura potrebbe risultare errato.
Per esempio, prendendo la frase

Bob shall pay a rent of $ 600.

L'analisi sintattica permetterebbe di identificare il soggetto "Bob", il verbo modale "shall", che andrà a definire il componente ADICO *deontic*, il verbo "pay" e l'oggetto "rent" (vedi figura 4.3). La frase, però, contiene anche il valore dell'oggetto, "$ 600". Automatizzare l'identificazione di questo valore tramite il grafo di dipendenze risulta complicato. In questi casi si può sfruttare un'altra tecnica, la "Named Entity Recognition".

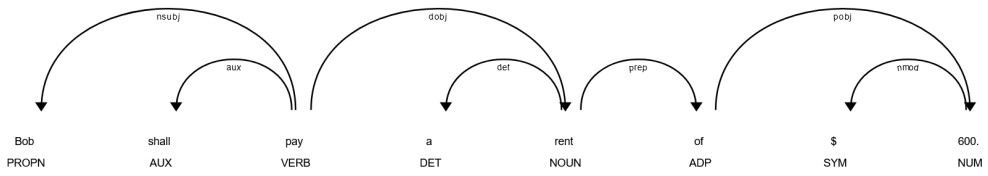


Figura 4.3

La "Named Entity Recognition" (NER) è uno dei metodi più efficaci per estrarre informazioni chiave dai testi. Tramite questo metodo si possono estrarre persone, luoghi, organizzazioni, espressioni temporali, somme di denaro e molto altro.
Tornando alla frase di esempio, se si prendesse l'oggetto e si unisse ai token dipendenti, il risultato sarebbe *a rent of $ 600*. Applicando la NER a questa parte della frase verrà evidenziata la somma di denaro (vedi figura 4.4), identificata come tale, e potrà essere estratta automaticamente.

Figura 4.4: Risultato ottenuto dalla NER

Esistono casi in cui l'analisi della sintassi potrebbe condurre ad interpretazioni errate del grafo. Infatti, se una frase contenesse un verbo che ammette più sintassi differenti, verrebbe meno la regola vista prima per la quale il soggetto equivale al componente *attribute* e l'oggetto al componente *object* di *aim*. Per esempio se la frase fosse

Bob shall pay Alice in installments of $ 600.

il grafo risultante (vedi figura 4.5) indica "Alice" come oggetto diretto del verbo "pay". In questa frase, però, il *dobj* equivale al componente *target* di *aim*. In un

sistema di interpretazione automatica del grafo risulterebbe complicato distinguere i due casi. Dunque, bisogna appoggiarsi ad un'altra tecnica, l'analisi semantica.

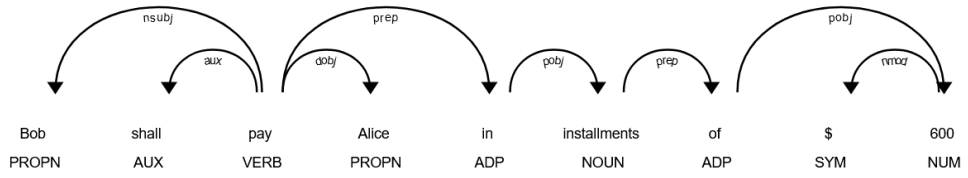


Figura 4.5

Analisi semantica

L'analisi semantica o "Semantic Role Labeling" (SRL) serve per comprendere il senso di una frase partendo dal suo predicato. In pratica, sfruttando l'analisi sintattica, vengono individuati gli elementi dipendenti dal predicato e classificati in base al ruolo che ricoprono all'interno della frase. Il risultato sarà una lista di elementi etichettati con la classe corrispondente. Degli esempi di etichette possono essere *agent*, *goal* o *result*.
Per eseguire l'analisi semantica serve un modello allenato su testo annotato. On-line esistono diverse risorse lessicali che mettono a disposizione frasi già annotate, come ad esempio VerbNet.
VerbNet, introdotto al capitolo 3.9, è la più grande collezione on-line di verbi in inglese che collega sintassi e semantica delle frasi. Questa collezione, organizzata in classi e sottoclassi, raggruppa insieme tutti i verbi che condividono la stessa sintassi e la stessa semantica. Per quanto riguarda il modello, invece, esistono degli strumenti open-source che offrono dei modelli già pronti. SemParse, descritto in modo dettagliato al capitolo 3.10, è uno di questi. Questo strumento costruisce il proprio modello basandosi su più risorse lessicali contemporaneamente, in modo tale da ridurre la probabilità di etichettatura errata.
Riprendendo l'esempio critico dell'analisi sintattica, questa volta il risultato che si ottiene è decisamente più semplice da interpretare.
In figura 4.6 viene mostrata la prima parte del risultato ottenuto tramite SemParse. Innanzitutto, lo strumento ha riconosciuto e classificato come verbi le parole "pay" e "installments". Sebbene la seconda sia errata, il valore è facilmente scartabile poiché dall'analisi lessicale soltanto "pay" risulta essere un verbo. Dunque, escludendo "installments" e considerando il verbo effettivo, lo strumento ha riconosciuto "Bob" come colui che compie l'azione (*agent*), "shall" come verbo modale (*AM-MOD modal*),"Alice" come colei che riceve l'azione (*recipient*) e "in installments of $ 600" come oggetto dell'azione (*asset*).
Nella seconda parte del risultato, mostrata in figura 4.7, sono stati riportati gli eventi

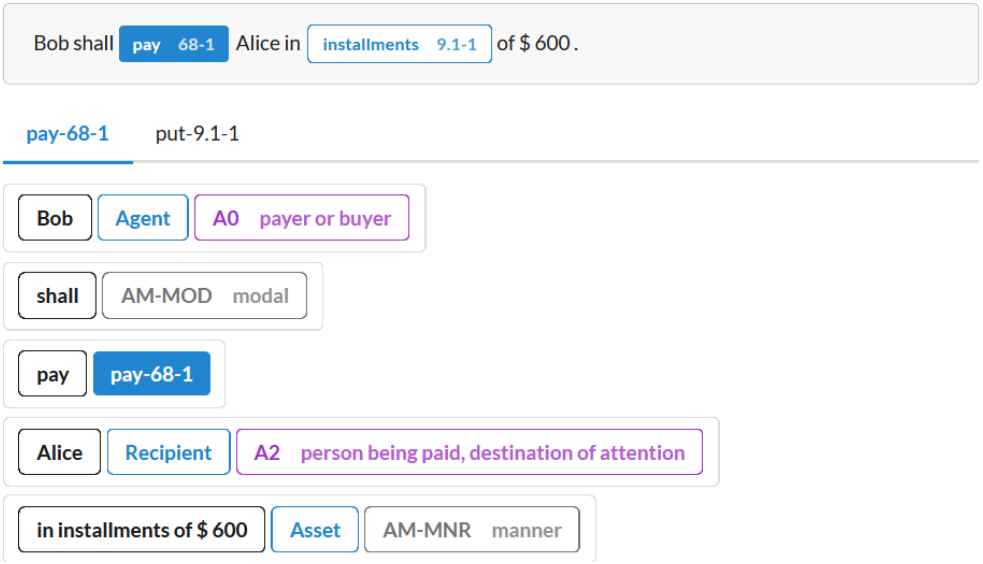


Figura 4.6: Prima parte del risultato ottenuto con SemParse

associati al verbo. A destra sono elencati gli eventi principali, mentre a sinistra quelli secondari. Tra gli eventi secondari, quello di maggiore interesse è l'evento *Transfer* poiché indica che la frase sta descrivendo un'azione di trasferimento. Grazie agli eventi è possibile stabilire il senso delle frasi, elemento fondamentale per selezionare le frasi che esprimono un'azione eseguibile in uno smart contract.
Attraverso l'analisi semantica è possibile identificare anche il componente ADICO *condition*. Quando nella frase è presente una condizione, questa viene etichettata con *AM-TMP temporal*, se è una condizione temporale, o con *AM-ADV adverbial*, in caso contrario.
Le condizioni temporali iniziano con una preposizione temporale, ad esempio "after", "before", "in", etc. Queste condizioni esprimono un vincolo cronologico stabilito da una data, dall'esecuzione di un'azione o dall'avvenimento di un evento.
L'altra tipologia di condizioni, invece, è caratterizzata da congiunzioni condizionali quali "if", "in case", "unless", etc.

4.2 Implementazione dello strumento

Dopo aver compreso quali sono le operazioni da eseguire su una frase per poter essere scomposta in componenti ADICO, si può passare all'implementazione dello strumento.
Come introdotto ad inizio capitolo, l'intero processo può essere suddiviso in due sottoprocessi, uno per l'estrazione dei componenti e l'altro per la generazione del

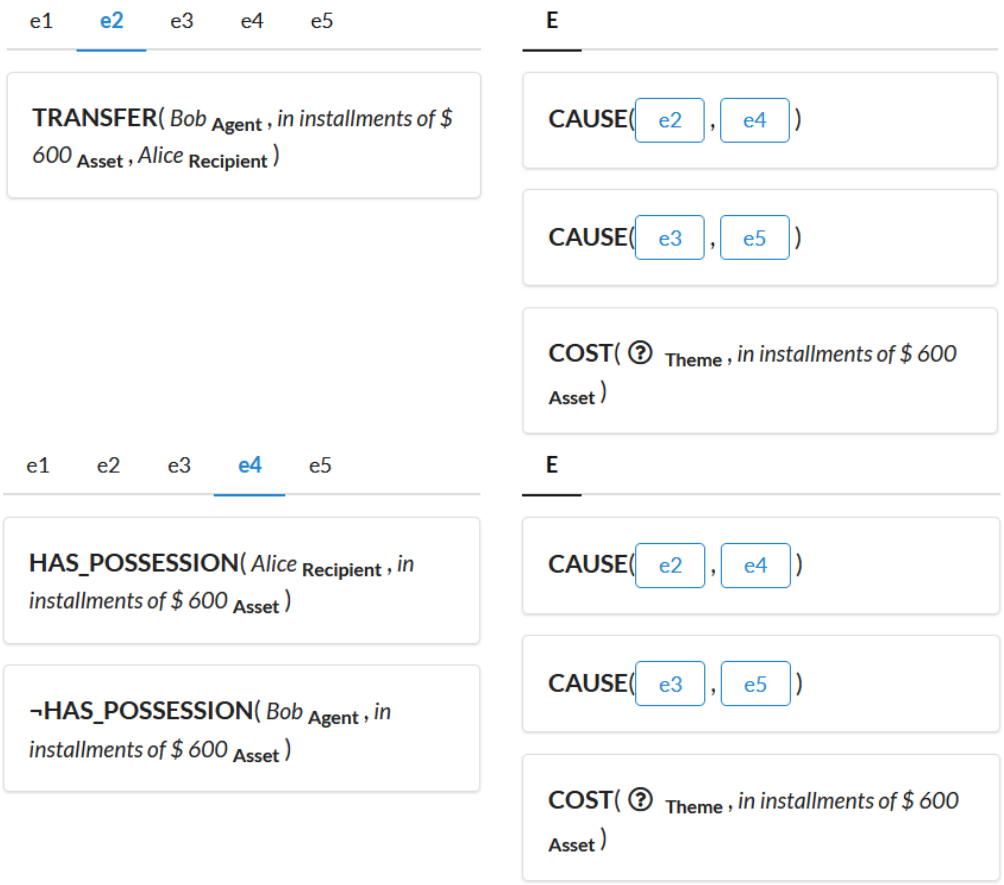


Figura 4.7: Seconda parte del risultato ottenuto con SemParse

codice.
In questa sezione verranno spiegati nel dettaglio entrambi i sottoprocessi e verrà mostrato come sono stati implementati all'interno dello strumento.

4.2.1 Estrazione dei componenti

L'estrazione dei componenti si basa, principalmente, sulle operazioni di elaborazione del linguaggio naturale viste nella sezione precedente.
Per l'implementazione di questa parte, oltre alla libreria Spacy e allo strumento SemParse, è stata utilizzata la libreria LexNLP. Questa libreria, descritta al capitolo 3.7, offre parecchie funzioni utili per la comprensione di testo legale. In particolar modo, è stata la libreria che ha permesso di ottenere i risultati migliori per quanto riguarda la Named Entity Recognition e la scomposizione del testo in frasi
Il processo di estrazione inizia con la lettura del file .txt contenente il contratto

scritto in lingua inglese (si è deciso di non utilizzare il formato PDF poiché la conversione di alcuni caratteri avveniva in modo errato, generando malfunzionamenti nelle fasi successive). Per avere un'idea più chiara della tipologia di testo sulla quale si andrà a lavorare, di seguito è stata riportata la parte iniziale di un contratto di affitto (i dati sensibili sono stati inventati):

STANDARD LEASE AGREEMENT

This Agreement, dated , 20. , by and between an
individual known as David Miller of 324 W Gore St, Orlando, Florida, 32806, hereinafter known as
the "Landlord",

AND

An individual known as Richard Garcia, hereinafter known as the "Tenant(s)", agree to the
following:

OCCUPANT(S): The Premises is to be occupied strictly as a residential dwelling with only the
Tenant(s) mentioned above as the Occupant(s).

OFFER TO RENT: The Landlord hereby rents to the Tenant(s), subject to the following terms and
conditions of this Agreement, an apartment with the address of 7000 NW 27th Ave, Miami, Florida,
33147 consisting of 1 bathroom(s) and 2 bedroom(s) hereinafter known as the "Premises". The
Landlord may also use the address for notices sent to the Tenant(s).

PURPOSE: The Tenant(s) and any Occupant(s) may only use the Premises as a residential
dwelling. It may not be used for storage, manufacturing of any type of food or product, professional
service(s), or for any commercial use unless otherwise stated in this Agreement.
... ...

Una volta letto, il file deve essere suddiviso in frasi. Questo procedimento viene effettuato tramite la funzione *get_sentence_list()*, messa a disposizione da LexNLP. La funzione riceve come parametro il testo da scomporre e ritorna la lista delle

frasi ricavate.

Selezione delle frasi valide

L'esempio sopra mostra chiaramente come la formattazione del testo non agevola la scomposizione. Le frasi sono spezzate su più righe e sono presenti degli elementi che disturbano questo procedimento. Di conseguenza, la lista potrebbe contenere alcune frasi non valide che devono essere rimosse.
Per stabilire la validità di una frase, ognuna di esse viene tokenizzata e analizzata sintatticamente. Tutte quelle che non hanno un verbo come radice del grafo di dipendenze vengono rimosse in quanto non riconosciute come valide.
Le frasi rimanenti vengono poi analizzate con SemParse per comprendere il tipo di azione che descrivono. Gli sviluppatori di SemParse hanno messo a disposizione, oltre all'interfaccia grafica, una API per comunicare con il server. Effettuando una richiesta http GET tramite l'URL

https://verbnetparser.com/predict/semantics?utterance=

e aggiungendo al parametro *utterance* la frase da analizzare, questo ritornerà un JSON contenente le informazioni ricavate dall'analisi (vedi sezione 3.10).
SemParse impone un limite massimo di 256 caratteri per frase; quindi, tutte le frasi che superano quella lunghezza devono essere scartate poiché non possono essere analizzate.
Ottenuto il JSON, dalla chiave "props" si ricava la lista dei verbi e delle rispettive informazioni. In questa lista, però, potrebbero esserci delle parole identificate erroneamente come verbi. Questo accade quando all'interno della frase sono presenti parole che derivano da un verbo, ad esempio i participi passati. Dunque, gli elementi di questo tipo devono essere rimossi in quanto non descrivono nessuna azione.
La rimozione viene fatta basandosi su due regole: un verbo deve essere riconosciuto come tale anche dall'analisi lessicale e se il tempo verbale è un participio passato allora non deve essere preceduto da un nome. Tutti gli elementi della lista che non le rispettano vengono rimossi. A questo punto, per ogni frase, si avranno le informazioni relative alla classe del verbo e le etichette dei ruoli identificati.
Prima di passare alle fasi successive, le frasi vengono raggruppate in classi, in modo tale da effettuare delle selezioni basate su di esse.
All'interno delle classi "begin-55.1-1-1" e "stop-55.4-1-1" si trovano le frasi che contengono rispettivamente le date di inizio e fine del contratto. Queste date vengono estratte tramite la funzione *get_raw_dates()* che andrà ad applicare la Named Entity Recognition (vedi figura 4.8). Per tutte le altre classi, invece, viene fatta una selezione basata sugli eventi.

```
from lexnlp.extract.en.dates import get_raw_dates

b = all_dict["begin-55.1-1-1"]
e = all_dict["stop-55.4-1-1"]

def extract_date(d):
  for element in d:
    snt = ""
    for span in element['spans']:
      snt += span['text']+" "
    dates = list(get_raw_dates(snt))
    if len(dates) > 0:
      print(dates, snt)

extract_date(b)
extract_date(e)
```

```
[datetime.date(2022, 4, 1)] a term beginning on April 01 , 2022
[datetime.date(2023, 4, 1)] a term ending on April 01 , 2023 ( the " Term '
```

Figura 4.8: Frammento di codice per l'estrazione di date

Come già menzionato nella sezione 4.1.3, ad ogni classe sono associati degli eventi. Questi eventi sono fondamentali per la comprensione delle frasi e permettono di stabilire se una frase può essere tradotta o meno.
Gli eventi principali, che descrivono azioni traducibili, sono l'evento *Transfer* e l'evento *Transfer_info*. Quindi vengono selezionate tutte le classi che possiedono questi due eventi.
Le classi selezionate, però, oltre a contenere le frasi candidate alla traduzione, contengono delle frasi incomplete. Per frase incompleta si intende una frase alla quale manca uno dei ruoli principali perché non presente o implicito. Per esempio, i ruoli principali dell'evento *Transfer* sono *agent*, *recipient* e *asset*. Se uno di questi dovesse mancare, la traduzione genererebbe una funzione che non può essere eseguita. Una volta scartate tutte le frasi incomplete, si può passare alla fase principale, ovvero all'estrazione dei componenti ADICO.

Preparazione della struttura ADICO

I ruoli dell'analisi semantica non vengono attribuiti a parole singole, ma a parti della frase. Quindi tramite operazioni basate sull'analisi sintattica e sulla Named Entity Recognition si vanno a rimuovere le parole superflue. Ad esempio, spesso la parte di frase etichettata con *recipient*, oltre all'attore, contiene anche la preposizione (figura 4.9). In questi casi, l'analisi sintattica permette di estrarre soltanto le parole utili e di scartare il resto.
Applicando lo stesso ragionamento al ruolo *agent*, si ottengono i due attori della frase e possono essere trasposti nei componenti ADICO *Attribute* e *Target*.

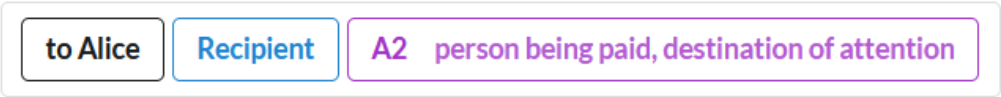


Figura 4.9

Tutti gli attori estratti dalle frasi saranno fondamentali per la generazione del codice; quindi, devono essere salvati all'interno di una lista.
Per quanto riguarda il ruolo *asset*, invece, la somma di denaro viene estratta tramite NER. In alcuni casi potrebbe capitare che invece del ruolo *asset* (che garantisce la presenza di una somma di denaro), la frase abbia il ruolo *theme*. Il ruolo *theme* identifica l'oggetto dell'azione, ma non da nessuna informazione sul valore o sulla quantità. Questo capita quando la definizione dell'oggetto avviene in una frase differente, che quindi deve essere rintracciata.
Generalmente, le frasi di questo tipo rientrano nella classe "seem-109-1-1". Di conseguenza, applicando la NER alle frasi di questa classe che contengono quell'oggetto, si estrare la quantià o la somma di denaro.
Estratto questo valore, esso verrà trasposto nel componente ADICO *Object* il quale, insieme a *Target* e al verbo, andrà a comporre il componente *Aim*.
Il componente *Deontic* è quello più semplice da estrarre poiché il ruolo *AM-MOD modal* è composto da una singola parola, il verbo modale.
Infine, rimane da preparare il componente *Condition*. Per la creazione di questo componente i ruoli da utilizzare sono due: *AM-TMP temporal* e *AM-ADV adverbial*.
Le condizioni temporali introducono un vincolo temporale traducibile in un confronto tra due elementi (nel caso di date) o nella verifica di una variabile booleana (nel caso di azioni o eventi).
Per effettuare il confronto, le preposizioni temporali più comuni sono mappate con degli operatori di confronto (tabella 4.2) e la data stabilita dalla condizione viene estratta tramite NER (generalmente il confronto avviene tra questa data e la data corrente).

Preposizione temporale	**Operatore di confronto**
After	>
Before	<
On	==
Until	<
Within	<=
From	>=

Tabella 4.2: Operatori di confronto relativi alle preposizioni temporali

Per quanto riguarda le variabili booleane, invece, il procedimento da seguire è lo stesso di quello per le condizioni *AM-ADV adverbial*.
Gli scenari possibili sono due:

- La condizione si basa sul verificarsi di un'azione da eseguire tramite lo smart contract.
- La condizione si basa su un evento/azione non eseguibile dallo smart contract.

Nel primo caso basta creare una variabile booleana da settare ad esecuzione avvenuta dell'azione. Nel secondo caso, invece, non potendo controllare in modo automatico l'avvenimento dell'evento, bisogna introdurre un'azione fittizia che avrà il solo compito di settare una variabile booleana quando chiamata.
Estratti gli elementi principali dalle condizioni e composto il componente ADICO *Condition*, è tutto pronto per poter passare alla realizzazione del codice.

4.2.2 Traduzione da ADICO a smart contract

Il codice dello smart contract è stato scritto tramite la versione più recente di Solidity. Questo linguaggio, descritto al capitolo 3.2, è quello più utilizzato sulla piattaforma Ethereum (vedi 3.1) per lo sviluppo degli smart contract.
Il codice è stato composto partendo da un template, contenente la struttura comune a tutti gli smart contract, al quale sono state aggiunte le parti generate dalla traduzione delle strutture ADICO.

Costruzione del template

Il template, implementato come stringa Python, è composto dallo scheletro dello smart contract al quale sono state aggiunte una serie di funzioni e modificatori, necessari per il funzionamento generale del contratto e che verranno spiegati in questo paragrafo.
Il template può essere scomposto in quattro parti: importazioni, variabili, modificatori e funzioni.
Nella prima parte del template, vengono importati tre smart contract:

- **DateTimeContract.sol**: in solidity una data viene definita attraverso un *timestamp*; questo smart contract [24] offre una serie di funzioni per l'estrazione di informazioni temporali (es. giorno, mese e anno) dai *timestamp*.
- **FeedRegistryInterface.sol**:questo smart contract [25] permette di ottenere il valore di conversione corrente della maggior parte di valute e criptovalute.
- **Denominations.sol**: questa è una libreria [26] di supporto per utilizzare le funzioni dello smart contract introdotto al punto precedente.

Nella parte dedicata alle variabili, sono state inserite *state*, *Actor* e *actors*. La variabile *state* serve per tenere traccia dello stato corrente dello smart contract, in modo tale da garantire un ordine cronologico di esecuzione tra le funzioni. La struttura *Actor* raggruppa tutti gli attributi che ogni attore possiede. Invece, *actors* di tipo *mapping* serve ad identificare una struttura *Actor* attraverso una stringa. Questa decisione è stata presa poiché dalle strutture ADICO gli attori vengono ricavati come stringhe.
In merito ai modificatori, è stato aggiunto *inState(State _state)*. Questo modificatore permette di controllare, prima che una funzione venga eseguita, se lo stato attuale dello smart contract è quello richiesto dalla funzione.
Infine, nella parte dedicata alle funzioni, sono state aggiunte *allRegistered()*, *allConfirmed()*, *allSetup()* e *terminate()* per permettere al proprietario dello smart contract di modificare lo stato, *actorRegistration()* per permettere di registrare gli attori e *getLatestPrice()* e *fromUsdToEth* che permettono di convertire un importo da dollari a ether (la criptovaluta di Ethereum).
Di seguito è stato riportato il template utilizzato:

```
sol_template = """
pragma solidity^0.8.7;

import "https://github.com/Quant-Finance-HQ/solidity-datetime/
    contracts/DateTimeContract.sol";
import "@chainlink/contracts/src/v0.8/interfaces/
    FeedRegistryInterface.sol";
import "@chainlink/contracts/src/v0.8/Denominations.sol";

contract MyContract {

  enum State {Created, Registered, Confirmed, Started, Terminated}
  State public state;
  DateTimeContract dateTime = new DateTimeContract();
  FeedRegistryInterface internal registry;

  struct Actor {
    string name;
    address addr;
    bool isRegistered;
    bool hasConfirmed;
  }

  mapping(string => Actor) public actors;

  address public owner;"""+variables+"""

  modifier isOwner() {
```

```
27      require(msg.sender == owner, "Caller is not owner");
28      _;
29    }
30
31    modifier inState(State _state) {
32      require(state == _state, "This function cannot be executed in the
        current state");
33      _;
34    }
35    """+modifiers+isAllRegistered+isAllConfirmed+"""
36    constructor() {
37      owner = msg.sender;
38      state = State.Created;
39      registry = FeedRegistryInterface(0
       xAa7F6f7f507457a1EE157fE97F6c7DB2BEec5cD0);
40      """+add_to_contructor+"""
41    }
42
43    function getLatestPrice() public view returns (int) {
44        (
45            /*uint80 roundID*/,
46            int price,
47            /*uint startedAt*/,
48            /*uint timeStamp*/,
49            /*uint80 answeredInRound*/
50        ) = registry.latestRoundData(Denominations.ETH, Denominations.
       USD);
51        return price;
52    }
53
54    function fromUsdToEth(int256 dollars) public view returns (int) {
55        int256 converted = dollars * (10 ** 18) * (10  ** 8);
56        int256 price = getLatestPrice();
57        converted = usd / price;
58        return converted;
59    }
60
61    function actorRegistration(string memory name, address addr) public
        inState(State.Created) isOwner {
62      require(bytes(actors[name].name).length != 0, "No actor with this
        name");
63      actors[name].addr = addr;
64      actors[name].isRegistered = true;
65    }
66
67    function allRegistered() public isAllRegistered inState(State.
       Created) isOwner {
68      state = State.Registered;
69    }
```

```
70
71   function allConfirmed() public isAllConfirmed inState(State.
       Registered) isOwner {
72     state = State.Confirmed;
73   }
74
75   function allSetup() public inState(State.Confirmed) isOwner {
76     state = State.Started;
77   }
78
79   function terminate() public inState(State.Started) isOwner {
80     state = State.Terminated;
81   }
82   """+confirm_functions+functions+"""
83 }
84 """
```

Registrazione degli attori e conferma

Il contratto di partenza non è stato concepito con lo scopo di essere tradotto in smart contract. Quindi, non esiste nessuna identificazione tramite chiavi pubbliche per attori. DI conseguenza serve una parte di codice dedicata alla registrazione delle chiavi pubbliche.
Durante l'estrazione dei componenti ADICO, tutti gli attori del contratto sono stati salvati in una lista. Ciclando gli elementi della lista, per ogni attore viene inizializzata *Actor* all'interno del costruttore,

```
1 act = """
2     actors["""+'"'+actor+'"'+"""] = Actor("""+'"'+actor+'"'+""",
      address(0), false, false);
3 """
```

viene generato un modificatore che permetterà di limitare l'esecuzione delle funzioni agli attori autorizzati

```
1 mod = """
2   modifier is"""+actor.capitalize()+"""() {
3     require(msg.sender == actors["""+'"'+actor+'"'+"""].addr, "Caller
       is not """+actor+"""");
4     _;
5   }
6 """
```

e viene creata una funzione che permette all'attore di confermare la sua partecipazione al contratto.

```
cf = """
  function confirm"""+actor.capitalize()+"""() public is"""+actor.
    capitalize()+""" inState(State.Registered) {
     actors["""+'"'+actor+'"'+"""].hasConfirmed = true;
  }
"""
```

Infine, vengono creati due modificatori, uno per verificare che tutti gli attori sono stati registrati e l'altro per verificare che tutti gli attori hanno accettato di prendere parte al contratto.

Costruzione delle variabili

Durante la fase di estrazione, sono stati individuati alcuni elementi (es. date di inizio e fine contratto) che per poter essere sfruttati, vanno memorizzati all'interno del codice sotto forma di variabili.
In questa fase, per ognuno di questi elementi, viene ricavato il tipo di variabile solidity (es. int, string, etc.) con cui costruire la parte di codice relativa alla dichiarazione di essa.
Infine, tutte le variabili generate vengono raggruppate e aggiunte al template nella sezione apposita.

```
import numpy as np
import decimal

solidity_variables = ""
for k,v in variables.items():
  if isinstance(v, (int, np.uint)):
    tmp = """
  uint public """+k+""" = """+str(v)+""";"""
  elif isinstance(v, decimal.Decimal):
    tmp = """
  int256 public """+k+""" = """+str(v)+""";"""
  elif isinstance(v, datetime.date):
    tmp = """
  uint public """+k+""" = """+str(v)+""";"""
  elif isinstance(v, str):
    tmp = """
  string public """+k+""" = """+str(v)+""";"""
  solidity_variables += tmp
```

Costruzione dei modificatori per le azioni

I modificatori servono per rappresentare le condizioni. Per ciascuna azione vengono generati tanti modificatori quante sono le condizioni presenti nel componente *Condition*.
Il componente, oltre a contenere l'espressione da verificare, contiene il tipo della condizione. Ad esempio, se questa fosse un confronto tra date, il tipo sarebbe *checkDate*, invece, se fosse un controllo per verificare che l'evento sia avvenuto, il tipo sarebbe *checkEvent*. In base alla condizione, il modificatore viene implementato in modo differente. Nel caso delle date, a seconda se il confronto richiesto è basato sul giorno, sul mese o sull'intera data, viene ricavato il giorno, il mese o la data corrente dalla variabile globale *block.timestamp*.

```
mod = """
  modifier """+v['prep']+v['date']+"""() {
    uint today = dateTime."""+v['dateType']+"""(block.timestamp);
    require(today """+v['comparator']+""" """+str(int(v['date']))+"""
    , "Date error");
    _;
  }
"""
```

Il nome di questo modificatore viene creato unendo la preposizione temporale e il valore della data. Per esempio, se la condizione fosse "before April 5, 2022" il nome sarebbe *before20220405()* Per quanto riguarda gli eventi, il controllo viene effettuato sulla variabile booleana corrispondete (aggiunta all'elenco delle variabili); se uguale a *true* allora l'evento è già avvenuto, altrimenti ancora deve avvenire. Per questi modificatori il nome è generato utilizzando un contatore. Di cosnegueza, al primo viene associato il nome *mod1()*, al secondo *mod2()* e così via.

```
mod = """
  modifier mod"""+str(mod_counter)+"""() {
    require("""+v['var']+""" == true, "Event error");
    _;
  }
"""
```

Costruzione delle funzioni

Gli attori utilizzano le funzioni per interagire con lo smart contract. Quindi, in fase di costruzione bisogna generare un nome che permetta di capire quale azione è stata implementata al loro interno. Per questo, si è deciso di utilizzare il verbo del

componente *Aim* accompagnato dall'oggetto, se presente. Ad esempio, se l'azione fosse "send money", il nome della funzione diventerebbe *sendMoney()*.
Dopo aver creato il nome, vengono aggiunti i modificatori relativi alle condizioni. Inoltre, viene aggiunto un modificatore per limitare la chiamata agli attori autorizzati e uno per controllare che lo stato attuale dello smart contract sia quello corretto.
Passando alla costruzione del corpo della funzione, per prima cosa viene invocata *fromUsdToEth()* che converte in ether il valore da pagare. Successivamente, attraverso il costrutto *require()*, si controlla che il valore mandato dall'utente sia uguale al valore ottenuto all'istruzione precedente; in caso contrario l'esecuzione fallisce ritornando un messaggio di errore.
Infine, tramite il metodo *call*, il valore viene trasferito al destinatario, ottenuto dal componente *target*.

```
function = """
  function """+verb+asset.capitalize()+"""() public payable inState(
    State."""+vps['state']+""") is"""+agent.capitalize()+""" """+
    function_modifiers+"""{
    int256 value = fromUsdToEth("""+asset.lower()+""");
    require(msg.value == value, "Wrong input value");
    (bool sent, bytes memory data) = actors["""+''''+recipient+''''+"""
    ].addr.call{
       value: msg.value}("");
    require(sent, "Failed to send Ether");
  }
"""
```

Capitolo 5

Valutazioni

Lo strumento è stato valutato su contratti presi dai seguenti siti: “Jotform” [27], “LawDepot” [28] e “SignWell” [29]. Sono state analizzate tre tipologie differenti di contratti: quattro contratti di affitto, tre contratti di vendita e tre contratti di prestito. Per ognuno di questi, è stato verificato se, all’interno del risultato prodotto, fossero presenti i componenti più importanti e se questi fossero stati generati correttamente. Infine, gli smart contract creati sono stati compilati e registrati su una blockchain di test per verificarne la corretta esecuzione.

5.1 Verifica dei componenti

Per ciascuna tipologia di contratto sono stati elencati i componenti principali. Inoltre, è stata effettuata un’analisi dei risultati con lo scopo di ripotare quante volte tali componenti sono stati estratti e tradotti con successo.

5.1.1 Contratti di affitto

I componenti principali di un contratto di affitto sono:

- Gli attori.
- Le date di inizio e di fine del contratto.
- Il pagamento dell’affitto.
- La data entro la quale pagare l’affitto.
- Il pagamento della cauzione.
- La restituzione della cauzione.

Dall'analisi dei contratti di affitto sono emersi i seguenti risultati. Gli attori sono stati estratti in modo corretto da tutti e quattro i contratti. Per quanto riguarda le date di inizio e fine del contratto, soltanto in uno l'estrazione è fallita poiché la data di fine non è stata espressa in modo esplicito. Il pagamento dell'affitto è stato tradotto con successo per tutti i contratti nonostante in uno l'importo si trovasse all'interno di una frase differente. La data di pagamento dell'affitto è stata estratta e tradotta in modo soddisfacente soltanto in un caso, negli altri non è stata rilevata in quanto indicata in frasi diverse dalla principale. Per quanto riguarda il pagamento della cauzione, in un contratto non è stato rilevato perché il verbo apparteneva ad una classe senza evento *transfer*. Infine, il ritorno della cauzione è stato tradotto in modo corretto per tutti i contratti, ma ai fini di un corretto funzionamento dello smart contract, uno è stato scartato in quanto mancava la parte del versamento.

Componenti	**N. corretti**
Attori	4/4
Date inizio/fine	3/4
Pagamento affitto	4/4
Data Pagamento affitto	1/4
Pagamento cauzione	3/4
Restituzione cauzione	3/4

Tabella 5.1: Numero di contratti di affito in cui i componenti sono stati tradotti correttamente

Esempi di funzioni

Di seguito vengono riportati degli esempi tratti dai contratti analizzati.

Pagamento dell'affitto: *Tenant(s) shall pay the Landlord in equal monthly installments of $850.00 (US Dollars) hereinafter known as the "Rent".*

```
function payRent() public payable inState(State.Started) isTenant {
    int256 value = fromUsdToEth(rent);
    require(msg.value == uint256(value), "Wrong input value");
    (bool sent, ) = actors["landlord"].addr.call{
      value: msg.value}("");
    require(sent, "Failed to send Ether");
  }
```

Pagamento della cauzione: *On execution of this Lease, the Tenant will pay the Landlord a security deposit of $2,000.00 (the "Security Deposit").*

```
function payDeposit() public payable inState(State.Confirmed)
    isTenant {
     int256 value = fromUsdToEth(deposit);
     require(msg.value == uint256(value), "Wrong input value");
     (bool sent, ) = actors["landlord"].addr.call{
       value: msg.value}("");
     require(sent, "Failed to send Ether");
   }
```

Restituzione della cauzione: *after the termination of this Agreement, Landlord will return the security deposit to Tenant*

```
function returnDeposit() public payable inState(State.Started)
    isLandlord afterTermination {
     int256 value = fromUsdToEth(deposit);
     require(msg.value <= uint256(value), "Wrong input value");
     (bool sent, ) = actors["tenant"].addr.call{
       value: msg.value}("");
     require(sent, "Failed to send Ether");
   }
```

5.1.2 Contratti di vendita

Gli elementi principali all'interno di un contratto di vendita sono:

- Gli attori.
- La data entro la quale deve avvenire il pagamento.
- Il pagamento dei beni.
- L'invio dei beni.
- La data di consegna dei beni.

Dall'analisi dei contratti di vendita sono emersi i seguenti risultati. Anche questa volta gli attori sono stati ricavati in modo corretto da tutti i contratti. La data entro la quale deve avvenire il pagamento è stata estratta con successo in un caso su tre, nei casi in cui l'estrazione è fallita la data si trovava in frasi differenti da quella del pagamento ed era dichiarata in modo implicito. Il pagamento dei beni è stato tradotto correttamente per tutti i contratti. Per quanto riguarda l'invio dei

beni, sebbene lo strumento abbia estratto correttamente le informazioni necessarie, non è riuscito a tradurre l'azione in quanto attualmente non è stato implementato l'invio di beni differenti da somme di denaro. Infine, l'estrazione della data entro la quale consegnare i beni è fallita in un solo caso, poiché non presente all'interno del contratto.

Componente ADICO	Costrutto Solidity
Attori	3/3
Data pagamento	1/3
Pagamento beni	3/3
Invio beni	0/3
Data consegna	2/3

Tabella 5.2: Numero di contratti di vendita in cui i componenti sono stati tradotti correttamente

Esempi di funzioni

Di seguito vengono riportati degli esempi tratti dai contratti analizzati.

Pagamento dei beni: *The Buyer will accept the Goods and pay the Seller for the Goods with the sum of $5,000.00 (USD) on July 15, 2020 (the "Purchase Price").*

```
modifier on20200715() {
    uint today = dateTime._now();
    require(today == 1594771200, "Date error");
    _;
}

function payPrice() public payable inState(State.Started) isBuyer
    on20200715 {
    int256 value = fromUsdToEth(price);
    require(msg.value == uint256(value), "Wrong input value");
    (bool sent, ) = actors["seller"].addr.call{
      value: msg.value}("");
    require(sent, "Failed to send Ether");
  }
```

5.1.3 Contratti di prestito

Gli elementi principali all'interno di un contratto di prestito sono:

- Gli attori.
- La data entro la quale saldare il debito.
- L'erogazione del prestito.
- Il pagamento del debito.

Dall'analisi dei contratti di prestito sono emersi i seguenti risultati. Come per le altre tipologie, anche in questa gli attori sono stati riconosciuti ed estratti correttamente da tutti i contratti. Per quanto riguarda la data entro la quale saldare il debito, l'estrazione è avvenuta con successo per un solo contratto, gli altri avevano la data dichiarata in parti differenti del codice rispetto al pagamento. L'erogazione del prestito è stato tradotto in modo corretto per tutti e tre i contratti. Infine, il pagamento del debito è stato tradotto con successo in due casi su tre poiché, in un contratto, il verbo utilizzato apparteneva ad una classe priva dell'evento *transfer*.

Componente ADICO	Costrutto Solidity
Attori	3/3
Data debito	1/3
Effettuare prestito	3/3
Pagamento debito	2/3

Tabella 5.3: Numero di contratti di prestito in cui i componenti sono stati tradotti correttamente

Esempi di funzioni

Di seguito vengono riportati degli esempi tratti dai contratti analizzati.

Effetuare il prestito: *The Lender agrees to lend the Borrower $1500 (the "Loan Amount") on the terms and conditions set forth in this Personal Loan Agreement.*

```
function lendLoan() public payable inState(State.Started) isLender {
    int256 value = fromUsdToEth(loan);
    require(msg.value == uint256(value), "Wrong input value");
    (bool sent, ) = actors["borrower"].addr.call{
```

```
5         value: msg.value}("");
6     require(sent, "Failed to send Ether");
7   }
```

Pagare il debito: *The Borrower promises to repay this principal amount to the Lender, without interest payable on the unpaid principal, on or before January 24, 2022.*

```
modifier onBefore20222401() {
    uint today = dateTime._now();
    require(today <= 1642982400, "Date error");
    _;
}

function repayAmount() public payable inState(State.Started)
    isBorrower onBefore20222401 {
    int256 value = fromUsdToEth(amount);
    require(msg.value == uint256(value), "Wrong input value");
    (bool sent, ) = actors["lender"].addr.call{
      value: msg.value}("");
    require(sent, "Failed to send Ether");
  }
```

5.2 Test su blockchain

Dopo le valutazioni, gli smart contract completi sono stati compilati per verificare che la generazione automatica non avesse introdotto degli errori ed infine sono stati caricati su una blockchain di prova per testarne il funzionamento.
Per le fasi di compilazione e di testing è stato utilizzato Remix. Questa applicazione, introdotta al capitolo 3.3, offre dei plugins (es. compilatore, debugger, etc.) che permettono di programmare e testare gli smart contract in modo semplice ed efficace.
Di tutti i contratti compilati, nessuno presentava errori o warnings, quindi, si è passati alla fase di testing delle funzioni. Gli smart contract sono stati caricati sulla rete di prova "Kovan" utilizzando Metamask (vedi 3.4) e sono stati testati sfruttando tre account differenti (owner, attore1, attore2). Alla fine della fase di testing, tutti gli smart contract si sono dimostrati completamente funzionanti. Al seguente link è possibile reperire un esempio dei contratti generati `https://kovan.etherscan.io/address/0xc7cb74986f8faace5232913c20ca975bc7f844e7`

Capitolo 6

Conclusioni

In questa tesi è stata presentata una possibile soluzione alle problematiche che riguardano la scrittura degli smart contract. In particolare, è stato proposto uno strumento per permettere a tutti di sfruttare le potenzialità di questa nuova tecnologia senza dover apprendere nozioni nuove o rivolgersi ad esperti del settore. Basandosi sulla comprensione del linguaggio naturale, è stato possibile creare un metodo innovativo per la traduzione automatica dei contratti in smart contract. Il metodo si basa sulla combinazione di tecniche di analisi del testo per l'estrazione degli elementi principali dai contratti, i quali vengono utilizzati per completare il template dello smart contract.
In generale, osservando le valutazioni, lo strumento ha ottenuto dei risultati soddisfacenti. Infatti, nella maggior parte dei casi, i componenti principali del contratto sono stati estratti e tradotti con successo. In alcuni casi, però, utilizzare contratti non predisposti ad essere tradotti in smart contract ha causato non poche difficoltà. Se un contratto venisse scritto nell'ottica di essere poi tradotto, sicuramente si otterrebbero dei risultati migliori. Ad esempio, seguendo uno schema per la formattazione del testo si potrebbero evitare errori in fase di lettura dal file. Un ulteriore aiuto si potrebbe avere dalla definizione di uno standard per la stesura dei contratti, stabilendo un modo di scrivere le frasi che faciliti lo strumento nella comprensione del testo. Ad esempio, sarebbe utile evitare l'uso di parole ambigue e la scrittura di frasi molto lunghe o troppo articolate.
Bisogna tenere presente che durante l'implementazione dello strumento di traduzione si è preferito focalizzare l'attenzione sulla creazione di un codice che fosse prima di tutto funzionale alla realizzazione dell'obiettivo di questa tesi. Per questo motivo, non ci si è concentrati su altri dettagli implementativi, come quelli relativi all'uso di tecniche utili a prevenire attacchi informatici. Inoltre, lo strumento è in grado di estrapolare e tradurre condizioni semplici, mentre in presenza di condizioni più articolate il codice generato non è funzionante.
Alla luce di queste ultime considerazioni, lo strumento proposto potrebbe in futuro

essere perfezionato generando un codice più sicuro e migliorando la traduzione delle condizioni.
Per concludere, nonostante i limiti evidenziati precedentemente, la tesi dimostra che è possibile generare uno smart contract a partire dal linguaggio naturale e propone uno strumento che potrebbe prestarsi a implementazioni future per essere esteso anche a casi più complessi.

Bibliografia

[1] Eliza Mik. «Contracts in code?» In: *Law, Innovation and Technology* 13.2 (2021), pp. 478–509 (cit. a p. 4).

[2] Christopher K Frantz e Mariusz Nowostawski. «From institutions to code: Towards automated generation of smart contracts». In: *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE. 2016, pp. 210–215 (cit. a p. 5).

[3] Kevin Purnell e Rolf Schwitter. «Towards declarative smart contracts». In: *Proc. Symp. Distrib. Ledger Technol.* 2019, pp. 18–21 (cit. a p. 8).

[4] Dianhui Mao, Fan Wang, Yalei Wang e Zhihao Hao. «Visual and User-Defined Smart Contract Designing System Based on Automatic Coding». In: *IEEE Access* 7 (2019), pp. 73131–73143. DOI: 10.1109/ACCESS.2019.2920776 (cit. a p. 9).

[5] *Etherscan.io*. URL: https://etherscan.io/ (cit. a p. 9).

[6] *Google Blockly*. URL: https://developers.google.com/blockly (cit. a p. 10).

[7] Anastasia Mavridou e Aron Laszka. «Designing secure ethereum smart contracts: A finite state machine based approach». In: *International Conference on Financial Cryptography and Data Security*. Springer. 2018, pp. 523–540 (cit. a p. 10).

[8] Sean Tan, Sourav S Bhowmick, Huey Eng Chua e Xiaokui Xiao. «LATTE: Visual construction of smart contracts». In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2713–2716 (cit. a p. 12).

[9] Maximilian Wöhrer e Uwe Zdun. «Domain specific language for smart contract development». In: *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE. 2020, pp. 1–9 (cit. a p. 14).

[10] Emanuel Regnath e Sebastian Steinhorst. «Smaconat: Smart contracts in natural language». In: *2018 Forum on Specification & Design Languages (FDL)*. IEEE. 2018, pp. 5–16 (cit. a p. 16).

[11] *Ethereum*. URL: https://ethereum.org/en/ (cit. a p. 19).

[12] *Solidity*. URL: https://docs.soliditylang.org/en/v0.8.13/index.html (cit. a p. 20).

[13] *Remix*. URL: https://remix.ethereum.org/ (cit. a p. 22).

[14] *Metamask*. URL: https://metamask.io/ (cit. a p. 24).

[15] *Python*. URL: https://www.python.org/ (cit. a p. 26).

[16] Wikipedia. *Python — Wikipedia, L'enciclopedia libera*. [Online; in data 22-marzo-2022]. 2022. URL: http://it.wikipedia.org/w/index.php?title=Python&oldid=126067358 (cit. a p. 26).

[17] *Natural Language Toolkit*. URL: https://www.nltk.org/ (cit. a p. 27).

[18] Michael J Bommarito II, Daniel Martin Katz e Eric M Detterman. «LexNLP: Natural language processing and information extraction for legal and regulatory texts». In: *Research Handbook on Big Data Law*. Edward Elgar Publishing, 2021 (cit. a p. 27).

[19] *Spacy*. URL: https://spacy.io/ (cit. a p. 28).

[20] Karin Kipper Schuler. *VerbNet: A broad-coverage, comprehensive verb lexicon*. University of Pennsylvania, 2005 (cit. a p. 28).

[21] *SemParse Github repository*. URL: https://github.com/jgung/verbnet-parser (cit. a p. 30).

[22] *SemLink Github repository*. URL: https://github.com/cu-clear/semlink (cit. a p. 30).

[23] *VerbNet Parser*. URL: https://verbnetparser.com/ (cit. a p. 30).

[24] *DateTimeContract.sol*. URL: https://github.com/RollaProject/solidity-datetime/blob/master/contracts/DateTimeContract.sol (cit. a p. 46).

[25] *FeedRegistryInterface.sol*. URL: https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/interfaces/FeedRegistryInterface.sol (cit. a p. 46).

[26] *Denominations.sol*. URL: https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/Denominations.sol (cit. a p. 46).

[27] *Jotform*. URL: https://www.jotform.com/pdf-templates/agreement (cit. a p. 53).

[28] *LawDepot*. URL: https://www.lawdepot.com/ (cit. a p. 53).

[29] *SignWell*. URL: https://www.signwell.com/contracts/ (cit. a p. 53).