Pacta sunt servanda: smart legal contracts in *Stipula*

Silvia Crafa
crafa@math.unipd.it
University of Padova
Italy

Cosimo Laneve
cosimo.laneve@unibo.it
University of Bologna – Inria FOCUS
Italy

Giovanni Sartor
Giovanni.Sartor@unibo.ti
University of Bologna – European University Institute
Italy

ABSTRACT

There is a growing interest in running legal contracts on blockchain systems to exploit the advantages of such systems in terms of disintermediation, transparency and immutability. At the same time, it is important to understand to what extent smart contracts programs may capture legal content. To fulfil these needs, we undertake a foundational study of legal contracts and we distill four main features: agreement, permissions, violations and obligations. We therefore design *Stipula*, a domain specific language that assists lawyers in programming legal contracts through specific patterns. The language is based on a small set of abstractions that express the features of legal contracts and that are amenable to be executed on blockchain systems. *Stipula* is equipped with a formal semantics and an observational equivalence, that provide for a clear account of the contracts' behaviour, and allows us to sketch an implementation on top of the Ethereum platform, pinpointing the critical differences between the two settings.

CCS CONCEPTS

• Theory of Computation; **• Models of Computation**; **• Concurrency**; **• Distributed computing models**;

KEYWORDS

Legal contracts, smart contracts, domain-specific language, operational semantics, legal bisimulation

ACM Reference Format:
Silvia Crafa, Cosimo Laneve, and Giovanni Sartor. 2021. Pacta sunt servanda: smart legal contracts in *Stipula*. In *???*. ACM, New York, NY, USA, 14 pages. https://doi.org/????

1 INTRODUCTION

A legal contract is "an agreement which is intended to give rise to a binding legal relationship or to have some other legal effect" [23]. The parties are in principle free to determine the content of their contracts (*party autonomy/freedom of contract*): the law recognizes their intention to achieve the agreed outcomes and secures the enforcement of such outcomes (*legally binding effect*). A contract produces the intended effects, declared by the parties, only if it is legally valid: the law may deny validity to certain clauses (*e.g.* excessive interests rate) and/or may establish additional effects that were not stated by the parties (*e.g.* consumer's power to withdraw from an online sale, warranties, etc.). The intervention of the law is particularly significant when the contractor (usually the weaker party, such as the worker in an employment contract or the consumer in an online purchase) agrees without having awareness of all clauses in the contract, nor having the ability to negotiate them, due to the existing unbalance of power.

Blockchain-based smart contracts have been advocated for digitally encoding legal contracts, so that the execution and enforcement of contractual conditions may occur automatically, without human intervention. This technology enables cutting performance costs and increasing outcomes certainty. Moreover, since the execution of smart contracts is trackable and irreversible, parties that to not trust each other can use them with no intermediation of a trusted third party. On this account, several governments have recently recognised that smart contracts and, more general, programs operating over distributed ledgers may indeed have legal value [2, 4, 6]. However, the assimilation of smart contracts to legally binding contracts, or rather the double nature of smart contracts as computational mechanisms and as legal contract raises both legal and technological issues.

In this paper, after discussing the main issues raised by the literature, we propose a technology that may contribute to addressing them. The overall aim is to facilitate the transparency of smart contracts as well as the mapping of computational operations into legal-institutional outcomes, thus limiting or mitigating the problems concerning the implementation of smart contracts and preventing disputes between the parties.

Specifically, we put forward *Stipula*, a new domain specific language for the creation of smart legal contracts operating over a blockchain. *Stipula* is pivoted on a small number of abstractions that are useful to capture the distinctive elements of legal contracts, that is *permissions*, *prohibitions*, *obligations*, fungible and non fungible *assets* exchanges, risk (viz. *alea*), *escrows* and *securities*. All these normative elements are expressed by a strictly regimented behaviour in legal contracts: permissions and empowerments correspond to the possibility of performing an action at a certain stage, prohibitions correspond to the interdiction of doing an action, while obligations are recast into commitments that are checked at a specific time limit. Moreover, the set of normative elements changes over time according to the actions have been done (or not). To model these changes, *Stipula* commits to a *state-aware programming* style, inspired by the state machine pattern widely used in smart contracts (*c.f.* Solidity [1] and Obsidian [3]). This technique allows one to enforce the intended behaviour by prohibiting, for instance, the invocation of a function before another specific function is called.

A second distinctive feature of *Stipula* is the *event* primitive, a programming abstraction that is used to issue an obligation and schedule a future statement that automatically executes a corresponding penalty, if the obligation is not met. This allows one to implement legal obligations and commitments in terms of the future execution of a (state-aware) computation at a specific point in time.

A third peculiarity of *Stipula* is the *agreement operator*, which marks that the contract's parties have reached a consensus on the contractual arrangement they want to create. In legal contracts, this phase corresponds to the subscription of the contract, where there are parties that are going to *set* the contractual conditions and others that *accept* them. Technically, the operation is a multiparty synchronization, which is implemented in current smart contract languages by ad-hoc protocols (see Section 6).

The fourth key feature regards *assets*, which are first-class linear concepts in *Stipula*. In particular, *Stipula* has an explicit, and thus conscious, management of linear resources, such as currency-like values and tokens. The transfer of such resources must preserve the total supply: the sender of the asset must always relinquish the control of the transferred asset. These assets, which are pervasive in the most successful blockchain applications and the induced economy, are necessary in legal contracts, as well: currency is required for payment but also for escrows, and tokens, both fungible and non-fungible, are useful to model securities and provide a digital handle on a physical good. The design choice of explicitly marking asset movements with an ad hoc syntax in *Stipula* promotes a safer, asset-aware, programming discipline that reduces the risk of the so-called double spending, the accidental loss or the locked-in assets. Therefore, without providing the flexibility/expressive power of resource-aware languages like Move [11] or Nomos [10], *Stipula* uses a simple and powerful core of primitives that support the writing of digital legal contracts even by non ICT experts.

In Section 2 we give an interdisciplinary discussion about smart legal contracts, focusing on the interlace between the on-chain and off-chain elements required by the digitalization of juridical acts. The syntax of *Stipula* is formally defined in Section 3 where a simple example – the Bike-Rental contract – is used to describe the concepts. The semantics of *Stipula* is defined in Section 4, sticking to an operational approach that specifies the runtime behaviour of a legal contracts by means of transitions. In Section 5, following a standard technique in concurrency theory [19], we develop an observational equivalence that provides for an equational theory of smart legal contracts. The equivalence is based on a notion of *bisimulation* that is suitable to blockchain-based observations, and that equates contracts differing for hidden elements, such as names of states, and singles out conditions for identifying contracts that send two assets in different order. The study of the implementation of the distinctive elements of *Stipula*, namely agreements, assets and events, on a blockchain systems using a Solidity-like target language is undertaken in Section 6. The specification in *Stipula* of a set of archetypal acts, ranging from renting to (digital) licenses and to bets, is reported in Section 7

We end our contribution by discussing the related work in Section 8 and delivering our final remarks in Section 9. The appendix, which has been added for reviewer sake, contains a technical part.

2 SMART LEGAL CONTRACTS

A substantial debate has taken place on whether the parties' decision to execute a smart contract having certain computational effects may count as legal contract establishing corresponding legal effect, *e.g.* [12, 16, 18, 20]. A simple answer to this question comes from the principle of "*freedom of form*" in contracts, which is shared by modern legal systems: *parties are free to express their agreement using the language and medium they prefer, including a programming language.* Therefore, by this principle, smart contracts may count as legal contracts.

However, the problem is whether smart contracts preserve the essential elements of legal ones. In this respect, a legal contract is meant to bring about the *institutional effects* intended by the parties, that is establishing new obligations, rights, powers and liabilities between them or to transfer rights (such as rights to property) from one party to the other. These institutional effects are guaranteed by the possibility of activating judicial enforcements. That is, each party may start a lawsuit if he believes that the other party has failed to comply with the contract. In this case, the judge will have to interpret the contract, ascertain the facts of the case, and determine whether there has indeed been a contractual violation. Accordingly, the defaulting party may be enjoined to comply or pay damages. Whether and to what extent we may consider that a smart contract produces judicially enforceable legal effects is more debatable, given that smart contract modify especially in absence of international technical standards and transnational legal frameworks.

We recognise that no easy and comprehensive solution is yet at hand for the issue we have just mentioned. However, we believe that it can be at least mitigated if a strict, and understandable mapping is established between executable instructions and institutional-normative effects. To this aim, we observe that the lifecycle of a legal contract goes through a number of phases: (*a*) formation and negotiation, (*b*) contract storage/notarizing, (*c*) performance, enforcement and monitoring, (*d*) possible modification (*e*) dispute resolution, and (*f*) termination. Software-based solutions can be valuable in all these phases, but the specific features of blockchain-based smart contracts make them convenient only in some of them. For example, the negotiation of contractual conditions might require a degree of privacy that conflicts with that of the blockchain, which naturally runs on transnational infrastructures, thus crossing several, possibly different, legal systems and jurisdictions (*c.f.* General Data Protection Regulation is valid only in Europe). Similarly the dispute resolution can take advantage of online services, but can hardly be fully ported on-chain. It is also very problematic to amend the behaviour of a running smart contract to match a change in the parties' will.

On the other hand, the blockchain is perfectly suited to phases (*b*) and (*c*). Both the content of the legal contract and the expression of agreement of the parties can be notarized and code can be used to express the contractual clauses and automatically enforce them through the runtime execution, which is verified and recorded by the nodes of the blockchain.

Nevertheless, if smart contracts are legally binding, then it is necessary to ensure that the parties are fully aware of the computational effects of their code. Only in this case, there may be a genuine agreement over the content of the contract. Thus transparency and some degree of readability by contractors that have no or little computer expertise becomes a key requirement. Relating to this point, we acknowledge that similar problems also exist in natural language contracts, which are usually signed without the parties being aware of all of their clauses and judicial enforcement being too costly or complex to be practicable. Usually, when concluding online purchases of goods or services, most consumers just click the "accept" button, without even trying to read the clauses (which are often lengthy and full of legal jargon).

We believe that *Stipula* provides some important advantages as a language for specifying smart legal contracts. Its primitives are concise and abstract enough to be easily accessible to lawyers. Its formal operational semantics promises that the execution of contracts does not lead to unexpected behaviours and is amenable to automatic verification. To mention a distinctive feature of *Stipula*, the `agreement` primitive helps to deal with some legal issues, since it clearly identifies the moment when (some) legal effects are triggered and the parties who are involved. For instance, the set of parties involved in the agreement might include an Authority that is charged to monitor off-chain constraints, such as obligations of diligent storage and care, or the obligations of using goods only as intended, taking care of litigations and dispute resolution. Moreover, untrusted players involved in a bet contract can rely on the agreement to explicitly define the data source providing the outcome of the aleatory value associated to the bet.

3 THE *STIPULA* LANGUAGE

Stipula features a minimal set of primitives that are primary in legal contracts, such as agreements, field updates, conditional behaviour, timed events, value and asset transfer, functions and states like Finite State Machines (FSMs).

We use countable sets of: *contract names*, ranged over by $\mathtt{C}, \mathtt{C}'$, $\cdots$; names of externally owned accounts, called *parties*, ranged over by $A, A', \cdots$; *function names* ranged over $\mathtt{f}, \mathtt{g}, \cdots$. Parties represent the users involved in the contract, *i.e.* addresses in blockchain systems. *Assets* and generic contract's *fields* are syntactically set apart since they have different semantics. Then we assume a countable set of *asset names*, ranged over by $\mathtt{h}, \mathtt{h}', \cdots$, and a set of *field names*, ranged over $\mathtt{x}, \mathtt{x}', \cdots$. We reserve $\mathtt{z}, \mathtt{z}', \mathtt{y}, \mathtt{y}'$, for function parameters and $\mathtt{A}, \mathtt{A}', \cdots$ for parameters that are parties. Finally, we will use $\mathtt{Q}, \mathtt{Q}', \cdots$, to range over contract states. A smart legal contract in *Stipula* is written

$$
\begin{array}{l}
\mathtt{legal\_contract\ C\ \{} \\
\quad \mathtt{assets}\ \mathtt{h}_1,\ldots,\mathtt{h}_k \\
\quad \mathtt{fields}\ \mathtt{x}_1,\ldots,\mathtt{x}_{k'} \\
\quad \mathtt{agreement(A_1,\ldots,A_n)\{} \\
\qquad \mathtt{A}_i\ \mathtt{=SET=>}\ \mathtt{x}_{i_1},\ldots,\mathtt{x}_{i_{r_i}} \qquad\qquad // \ i \in I \\
\qquad \mathtt{A}_j\ \mathtt{=OK=>}\ \mathtt{x}_{j_1},\ldots,\mathtt{x}_{j_{s_j}} \qquad\qquad // \ j \in J \\
\quad \mathtt{\}} \Rightarrow \mathtt{@Q} \\
\quad \mathtt{F} \\
\mathtt{\}}
\end{array}
$$

where `C` identifies the contract; its body contains assets and fields (without any typed information: *Stipula* is type-free), the *agreement code*, where I and J are subsets of $1..n$, and `F` is a sequence of functions.

The dichotomy between assets and fields is a key design choice of *Stipula*. Indeed, the relevance of first-class resources, *i.e.* linear values that cannot be copied nor dismissed, is widely acknowledged in DLTs (*c.f.* Move [11] and Nomos [10]) to support a safer asset-aware contract programming. Legal contracts as well manage assets, such as money and tokens granting a digital access to (possibly physical) goods or services. Henceforth the decision to syntactically highlight the differences between operations on values and on assets.

The definition of *who* is going to participate to the contract and *what* are the terms of the contract in an explicit abstraction – the *agreement* – is another distinctive feature of *Stipula*. Technically, the agreement is the *constructor* of the contract, that specifies which parties may set the values for fields – operation `=SET=>` – and which ones may agree on such values – operation `=OK=>`. It also specifies the initial state of the contract. We assume a consistency criterium for these operations: (*i*) a field may be set by at most one party, (*ii*) parties may agree on fields set by others, and (*iii*) every party involved in the agreement must either set or agree on a possible empty sequence of fields, *e.g.* $\mathtt{A}\ \mathtt{=OK=>} \varnothing$. Observe that no asset can be set during the agreement, only fields. It is assumed that fields, assets and parties' names do not contain duplicates.

Functions `F` and their bodies are written according to the syntax in Table 1. The function's syntax highlights the constraint that only the party `A` can invoke the function `f`, and only when the contract is in state `Q`. Function's parameters are split in two lists: the *formal parameters* $\mathtt{z}_1,\ldots,\mathtt{z}_m$ in brackets and the *asset parameters* $\mathtt{y}_1,\ldots,\mathtt{y}_{m'}$ in square brackets. The *precondition* `(B)` is a predicate on parameters of the functions and fields and assets of the contract that constrains the execution of the body of `f`. Finally the *body* $\mathtt{\{S\ \ W\}} \Rightarrow \mathtt{@Q}'$ specifies the *statement part* `S`, the *event part* `W`, and the state $\mathtt{Q}'$ of the contract when the function execution terminates. Function's parameters are assumed without duplicates, and empty lists of (asset) parameters are shortened by omitting empty parenthesis.

Statements `S` include the empty statement `_` and different types of assignments, followed by a continuation. Assignments use the two symbols $\multimap$ and $\rightarrow$ to differentiate updates of assets and of fields, respectively. The syntax of the two operators is taken from [10]. Assignments can be either *local*, that is referring to local fields or local assets, denoted by $\mathtt{E} \rightarrow \mathtt{x}$ and $\mathtt{E} \multimap \mathtt{h},\mathtt{h}'$, respectively, or they can be *remote*, denoted by $\mathtt{E} \rightarrow \mathtt{A}$ and $\mathtt{E} \multimap \mathtt{h},\mathtt{A}$, defining the sending of a value and an asset, respectively, to the address `A`. Asset assignments are ternary operations: the meaning of $\mathtt{E} \multimap \mathtt{h},\mathtt{h}'$ is that the value of `E` is subtracted to the asset `h` and added to the asset $\mathtt{h}'$ – *resources stored in assets can be moved but cannot be destroyed.* The operational semantics will ensure that asset assignments can at most drain an asset, preventing assets with negative values. In the rest of the paper we will always abbreviate assignments such as $\mathtt{h} \multimap \mathtt{h},\mathtt{h}'$ and $\mathtt{h} \multimap \mathtt{h},\mathtt{A}'$ (which are very usual, indeed) into $\mathtt{h} \multimap \mathtt{h}'$ and $\mathtt{h} \multimap \mathtt{A}'$, respectively. Statements also include *conditionals* $\mathtt{(B)\{\ S\ \}}\ \ \mathtt{S}'$ that executes `S` if the predicate `B` is true and continues as $\mathtt{S}'$.

Events `W` are sequences of *timed continuations*. A timed continuation is a term $\mathtt{E} \gg \mathtt{@Q\{S\}} \Rightarrow \mathtt{@Q}'$, which is triggered at a time $\mathbb{t}$ that is the value of `E`. When triggered, the continuation `S` will be executed only if the contract's state is `Q`. At the end of the execution of `S`, the contract transits to $\mathtt{Q}'$. That is, in *Stipula*, the programming abstractions of FSMs are used to schedule the future execution of

$$\begin{array}{lcl}
\mathtt{F} & ::= & \_ \mid \mathtt{@Q\ A:\ f(z_1,\ldots,z_m)[y_1,\ldots,y_{m'}]\ \ (B)\{\ S\ \ W\ \}} \Rightarrow \mathtt{@Q'}\ \ \mathtt{F} \\
\mathtt{S} & ::= & \_ \mid \mathtt{E} \rightarrow \mathtt{x\ \ S} \mid \mathtt{E} \multimap \mathtt{h,h'\ \ S} \mid \mathtt{E} \rightarrow \mathtt{A\ \ S} \mid \mathtt{E} \multimap \mathtt{h,A\ \ S} \mid \mathtt{(B)\{\ S\ \}\ \ S} \\
\mathtt{W} & ::= & \_ \mid \mathtt{E} \gg \mathtt{@Q\{\ S\ \}} \Rightarrow \mathtt{@Q'}\ \ \mathtt{W} \\
\mathtt{E} & ::= & \mathtt{now} \mid \mathtt{X} \mid v \mid \mathtt{E\ \ op\ \ E} \\
\mathtt{B} & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{E\ \ rop\ \ E} \mid \mathtt{B\ \&\&\ B} \mid \mathtt{B\ ||\ B} \mid \mathtt{\sim B}
\end{array}$$

Table 1: Syntax of *Stipula*.

a (state-aware) computation at a specific point in time. We will show that this notion of events is pivotal in the encoding of legal obligations and commitments.

Expressions E can be names of assets, fields and parameters, generically ranged over by X. The names v and u will be reserved for values; they include integer and boolean constants, asset constants like (non-fungible) tokens, strings, and a time datatype. We will be a little liberal with values and operations, generically denoted by E op E: they include standard arithmetic operations and operations on tokens, *e.g.* use_once(token) generates a usage-code providing a single access to the service or the good associated to the token asset. The operation $\mathbb{t}$ + n sums n seconds to the time value $\mathbb{t}$. The identifier now stores the present time (when the code is executed). *Boolean expressions* B are the standard ones, where the operations rop are the relational operations (==, >=, etc.) The set of names occurring in E will be noted by fv(E).

A *Stipula* program $\mathcal{P}$ is a sequence of smart legal contracts definitions. The contracts are inactive as long as no group of addresses has interest to run them, by invoking the agreement code. We remark that in *Stipula* the code of a contract cannot invoke another contract: we postpone to future work the study of language extensions allowing cross references between legal contracts using inheritance and composition. Therefore, at the moment, since legal contracts are independent, there is no loss of generality in considering a program to be composed by a single smart legal contract definition.

Example 3.1. Consider the following simple contract for renting bikes:

```
1  legal_contract Bike_Rental {
2    assets balance
3    fields cost, time_limit, use_code
4
5    agreement (Lender, Borrower) {
6       Lender =SET=> time_limit, cost
7       Borrower =OK=> time_limit, cost
8    } ⇒ @Inactive
9
10   @Inactive Lender : offer (z) {
11      z → use_code
12   } ⇒ @Proposal
13
14   @Proposal Borrower : accept [y]
15     (y == cost) {
16       y ⊸ balance
17       use_code → Borrower
18       now + time_limit ≫
19           @Using {               //end-of-time usage
20               "End_Reached" → Borrower
21               balance ⊸ Lender
22           } ⇒ @End
23   } ⇒ @Using
24
25   @Using Borrower : end {
26     balance ⊸ Lender
27   } ⇒ @End
28 }
```

Listing 1: The rent for free contract

The agreement code specifies that there are two parties – the Lender and Borrower – and that the Lender sets the values for the time of usage (time_limit) and the cost, while the Borrower has to agree on such values. Then Lender sends a use-code that is stored in the contract (in the use_code field) and is not accessible to Borrower till he pays for the usage. The transition from Inactive to Proposal at the end of the offer function enables the Borrower to pay for the usage – function accept that takes in input an asset – so that the contract sends him the use-code (line 17) and he can use the bike till the time limit. This constraint is expressed by the event in lines 18-22. We have two remarks: first, the payment is not made to Lender but the asset is stored in the contract (in balance); second, the event will be triggered when the time expires. In this case a message to Borrower is sent, the payment is transferred to Lender, which will change bike's use code so that the bike will be locked at the next Borrower's stop. The function end can be invoked by Borrower to terminate the renting before time expires. The legal issues involved in a rent contract will be discussed in Section 7.

4 SEMANTICS

The meaning of *Stipula* primitives is defined in an operational way by means of a transition relation. Let $\mathtt{C}(\Phi\ ,\ \ell\ ,\ \Sigma\ ,\ \Psi)$ be a *runtime contract* where

- C is the contract name;
- Φ is the current state of the contract: it is either _ (for no state) or a contract state Q;
- ℓ is a mapping from fields and assets to values;
- Σ is a possible residual of a function body or of an event handler, *i.e.* Σ is either _ or a term S W $\Rightarrow$ @Q;
- Ψ is a (possibly empty) multiset of *pending events* that have been already scheduled for future execution but not yet triggered. We let Ψ be _ when there are no pending events, otherwise $\Psi = \mathtt{W}_1 \mid \ldots \mid \mathtt{W}_n$ such that each $\mathtt{W}_i$ is a single event expression (not a sequence), and its *time guard* is an expression that has already been evaluated into a time value $\mathbb{t}_i$.

Runtime contracts are ranged over by $\mathbb{C}, \mathbb{C}', \cdots$. A *configuration*, ranged over by $\mathbb{S}, \mathbb{S}', \cdots$, is a pair $\mathbb{C}\ ,\mathbb{t}$, where $\mathbb{t}$ is a global clock. As anticipated, there is no loss of generality in considering programs made of a single running contract; a remark about the extension to configurations containing several contracts is reported at the end of the section.

The transition relation of a *Stipula* program $\mathcal{P}$ is $\mathbb{S} \xrightarrow{\mu}_{\mathcal{P}} \mathbb{S}'$, where the label μ is either empty, or $\mathsf{A} : \mathsf{f}(\overline{u})[\overline{v}]$, or $v \rightarrow \mathsf{A}$, or $v \multimap \mathsf{A}$, or α, where $\alpha = \mathsf{A}_1\ \mathsf{R}_1\ [\overline{x_1} \mapsto \overline{v_1}] \parallel \cdots \parallel \mathsf{A}_k\ \mathsf{R}_k\ [\overline{x_k} \mapsto \overline{v_k}]$, with $\mathsf{R}_1, \cdots, \mathsf{R}_k \in \{\mathsf{SET}, \mathsf{OK}\}$. In the following we will always omit the index $\mathcal{P}$ because it is considered implicit. The formal definition of $\mathbb{S} \xrightarrow{\mu} \mathbb{S}'$ is given in Tables 2 and 3, using the following auxiliary predicates and functions:

- $[\![\mathsf{E}]\!]_\ell$ is a function that returns the value of E in the memory ℓ. We omit the definition.
- $[\![\mathsf{W}]\!]_\ell$ is the multiset of scheduled events obtained from the sequence W by replacing every time expression by its value in the time guards. That is $[\![\_]\!]_\ell = \_$ and $[\![\mathsf{E} \gg @\mathsf{Q}\{\mathsf{S}\} \Rightarrow @\mathsf{Q}'\ \ \mathsf{W}']\!]_\ell = [\![\mathsf{E}]\!]_\ell \gg @\mathsf{Q}\{\mathsf{S}\} \Rightarrow @\mathsf{Q}' \mid [\![\mathsf{W}']\!]_\ell$.
- Let Ψ be a multiset of pending events, and $\mathbb{t}$ a time value, then the predicate $\Psi\ ,\ \mathbb{t} \nrightarrow$ is *true* whenever $\Psi = \mathbb{t}_1 \gg @\mathsf{Q}_1\{\mathsf{S}_1\} \Rightarrow @\mathsf{Q}'_1 \mid \cdots \mid \mathbb{t}_n \gg @\mathsf{Q}_n\{\mathsf{S}_1\} \Rightarrow @\mathsf{Q}'_n$ and, for every $1 \leqslant i \leqslant n$, $\mathbb{t}_i \neq \mathbb{t}$, *false* otherwise.

Rule [Agree] in Table 2 establishes the agreement of the parties involved in the contract. This operation is a multiparty synchronization and the label α is intended to highlight this point. Some parties, namely A_i with $i \in I$, set the initial values of the contract's fields $\overline{\mathsf{x}_i}$. At the same time the parties A_j, with $j \in J$, agree on this field initialisation – *c.f.* the last premise. The parties, represented in the legal contract by the parameters $\mathsf{A}_1, \cdots, \mathsf{A}_n$ are instantiated by the actual addresses $A_1, \cdots, A_n$. We recall the consistency criterium stating that a field may be set by at most one party, the parties may agree on fields that they do not set, and all the n parties appear in the label α.

Rule [Function] defines function invocations; the label specifies the address A performing the invocation and the function name f with the actual parameters. The transition may occur provided (*i*) the contract is the state Q that admits invocations of f from A and (*ii*) the contract is *idle*, *i.e.* the contract has no statement to execute – *c.f.* the left-hand side runtime contract – (*iii*) the precondition B is satisfied, and no event can be triggered – *c.f.* the premise $\Psi\ ,\ \mathbb{t} \nrightarrow$. In particular, this last constraint expresses that events *have precedence* on possible function invocations. For example, if a payment deadline is reached and, at the same time, the payment arrives, it will be refused in favour of the event managing the deadline.

Rule [State Change] says that a contract changes state when the statements execution terminates and the sequence of events W is added to the multiset of pending events, up to the evaluations of their time expressions, *i.e.* the occurrences of the identifier now are replaced by the current value of the clock.

Rule [Event Match] specifies that event handlers may run provided there is no statement to perform in the runtime contract, and the time guard of the event has exactly the value of the global clock $\mathbb{t}$. Observe that the timeouts of the events are evaluated in an eager way when the event is scheduled – *c.f.* rule [State Change] – not when the event handler is triggered. Moreover, the state change performed at the end of the execution of the event handler is carried over again by the rule [State Change], with an empty sequence W.

Rule [Tick] models the elapsing of time. This happens when the contract has no statement to perform and no event can be triggered. Intuitively, the implementation of *Stipula* on top of a blockchain will bind the global clock to the timestamp of the current block. Therefore a sequence of semantic transitions performed in the same unit of time will correspond to a set of transactions inserted into the same block to be appended to the blockchain.

It is worth to notice that the foregoing rules imply that the complete execution of a function call does not affect the global time. This admits the paradoxical phenomenon that an endless sequence of function invocations does not make time elapse. While this is possible in theory, it is not in practice, since blocks can only include a finite number of transactions. Additionally, all the legal contracts we have analyzed are finite state, each state admits a single function invocation, and function invocations update the state in a noncircular way, thus preventing infinite sequence of function calls. An alternative choice would be to adjust the semantics so to increment the clock every time a maximal number of functions has been evaluated, thus forcing each block to contain at most a limited number of function invocations. We have preferred to stick to the simpler semantics.

Table 3 defines transitions due to the execution of statements. All these transitions are local to the runtime contract and time does not change. Therefore, for simplicity, we always omit the clock. We only discuss [Asset_Send] and [Asset_Update] because the other rules are standard. Rule [Asset_Send] returns part of an asset h to the party A. This part, named v, is removed from the asset, *c.f.* the memory of the right-hand side runtime contract in the conclusion. In a similar way, [Asset_Update] moves a part v of an asset h to an asset h'. For this reason, the final memory becomes $\ell[\mathsf{h} \mapsto \ell(\mathsf{h}) - v, \mathsf{h}' \mapsto \ell(\mathsf{h}') + v]$. We observe that assets, representing physical entities (coins, houses, goods, etc.) are never destroyed. The condition $\ell(\mathsf{h}) \geqslant v$ in the premises ensures that assets can never become negative.

The semantics of *Stipula* does not consider runtime errors, for instance an attempt to drain too much value from an asset results in a stuck configuration. We postpone to future work a precise account of runtime failures and contract errors, since it requires a deep interdisciplinary analysis of the legal issues involved in the execution of the exceptional cases.

The initial configuration of a *Stipula* program $\mathcal{P}$ made of a single contract C is $\mathsf{C}(\_\ ,\ \varnothing\ ,\ \_\ ,\ \_)\ ,\ \mathbb{t}$. The contract is *inactive* as long as no group of addresses has interest to run it, *c.f.* rule [Agree]. The global clock can be any value, because it corresponds to the absolute time, defined by the timestamp of the current block in the blockchain system.

Example 4.1. Possible initial transitions of the `Bike_Rental` contract in Example 3.1 are reported in Table 4. We assume that the actual names of parties are the same as the formal names (therefore we omit the mappings in the memories).

Let be $\alpha = \mathsf{Lender\ \ SET}\ [\mathsf{time\_limit} \mapsto 3600, \mathsf{cost} \mapsto 2] \parallel \mathsf{Borrower\ \ OK}\ [\mathsf{time\_limit} \mapsto 3600, \mathsf{cost} \mapsto 2]$ and $\ell = [\mathsf{cost} \mapsto 2, \mathsf{time\_limit} \mapsto 3600]$ (*i.e.* the cost of renting a bike is 2 euro per hour – the time is measured in seconds. Let also be $\ell' = \ell[z \mapsto 123, \mathsf{use\_code} \mapsto 123]$ and $\mathsf{S}\ \ \mathsf{W}$ be the body of the function `accept`.

To sum up, a legal contract behaves as follows:

(1) the first action is always an agreement, which moves the contract to an idle state;

[Agree]

$$\frac{\begin{array}{c}\mathtt{agreement(A_1,\cdots,A_n)\{\ A_i\ {=}SET{=}{>}\ \overline{x_i}\ \cdots\ A_j\ {=}OK{=}{>}\ \overline{x_j}}\ ^{i\in I\ j\in J}\}\Rightarrow \mathtt{@Q}\ \in\ \mathtt{C}\\ \alpha = A_i\ \mathsf{SET}\ [\overline{\mathtt{x}_i}\mapsto\overline{v_i}] \parallel \cdots \parallel A_j\ \mathsf{OK}\ [\overline{\mathtt{x}_j}\mapsto\overline{v_j}] \qquad [\overline{\mathtt{x}_i}\ \mapsto\ \overline{v_i}]^{i\in I} = [\overline{\mathtt{x}_j}\ \mapsto\ \overline{v_j}]^{j\in J}\end{array}}{\mathtt{C}(\_\,,\ \varnothing\,,\ \_\,,\ \_)\ ,\ \mathbb{t} \xrightarrow{\alpha} \mathtt{C}(\mathtt{Q}\,,\ [\mathtt{A}_i\mapsto A_i, \overline{\mathtt{x}_i}\ \mapsto\ \overline{v_i}]^{i\in I}\,,\ \_\,,\ \_)\ ,\ \mathbb{t}}$$

[Function]

$$\frac{\begin{array}{c}\mathtt{@Q\ A:\ f(\overline{z})[\overline{y}]\ (B)\ \{\ S\ W\ \}}\Rightarrow\mathtt{@Q'}\in\mathtt{C}\\ \Psi\ ,\ \mathbb{t}\not\rightarrow\\ \ell(\mathtt{A}) = A \quad \ell' = \ell[\overline{\mathtt{z}}\mapsto\overline{u},\overline{\mathtt{y}}\mapsto\overline{v}] \quad [\![\mathtt{B}]\!]_{\ell'} = true\end{array}}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\,,\ \Psi)\ ,\ \mathbb{t} \xrightarrow{A:\mathtt{f}(\overline{u})[\overline{v}]} \mathtt{C}(\mathtt{Q};\ell'\,,\ \mathtt{S\,W}\Rightarrow\mathtt{@Q'}\,,\ \Psi)\ ,\ \mathbb{t}}$$

[State Change]

$$\frac{[\![\mathtt{W}\{^{\mathbb{t}}/_{\mathsf{now}}\}]\!]_\ell = \Psi'}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\mathtt{W}\Rightarrow\mathtt{@Q'}\,,\ \Psi)\ ,\ \mathbb{t}\longrightarrow\mathtt{C}(\mathtt{Q'}\,,\ \ell\,,\ \_\,,\ \Psi'\mid\Psi)\ ,\ \mathbb{t}}$$

[Event Match]

$$\frac{\Psi = \mathbb{t}\gg\mathtt{@Q}\ \{\ \mathtt{S}\ \}\Rightarrow\mathtt{@Q'}\mid\Psi'}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\,,\ \Psi)\ ,\ \mathbb{t}\longrightarrow\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \mathtt{S}\Rightarrow\mathtt{@Q'}\,,\ \Psi')\ ,\ \mathbb{t}}$$

[Tick]

$$\frac{\Psi\ ,\ \mathbb{t}\not\rightarrow}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\,,\ \Psi)\ ,\ \mathbb{t}\longrightarrow\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \_\,,\ \Psi)\ ,\ \mathbb{t}+1}$$

Table 2: The transition relation of *Stipula* – Part 1

[Value_Send]

$$\frac{[\![\mathtt{E}]\!]_\ell = v \quad \ell(\mathtt{A}) = A}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \mathtt{E}\rightarrow\mathtt{A}\,\Sigma\,,\ \Psi)\xrightarrow{v\rightarrow A}\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \Sigma\,,\ \Psi)}$$

[Asset_Send]

$$\frac{[\![\mathtt{E}]\!]_\ell = v \quad \ell(\mathtt{h})\geqslant v \quad \ell(\mathtt{A}) = A}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \mathtt{E}\multimap\mathtt{h,A}\,\Sigma\,,\ \Psi)\xrightarrow{v\multimap A}\mathtt{C}(\mathtt{Q}\,,\ \ell[\mathtt{h}\mapsto\ell(\mathtt{h})-v]\,,\ \Sigma\,,\ \Psi)}$$

[Field_Update]

$$\frac{[\![\mathtt{E}]\!]_\ell = v}{\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \mathtt{E}\rightarrow\mathtt{x}\,\Sigma\,,\ \Psi)\longrightarrow\mathtt{C}(\mathtt{Q}\,,\ \ell[\mathtt{x}\mapsto v]\,,\ \Sigma\,,\ \Psi)}$$

[Asset_Update]

$$\frac{[\![\mathtt{E}]\!]_\ell = v \qquad \ell(\mathtt{h})\geqslant v}{\begin{array}{l}\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \mathtt{E}\multimap\mathtt{h,h'}\,\Sigma\,,\ \Psi)\\ \quad\longrightarrow\mathtt{C}(\mathtt{Q}\,,\ \ell[\mathtt{h}\mapsto\ell(\mathtt{h})-v, \mathtt{h'}\mapsto\ell(\mathtt{h'})+v]\,,\ \Sigma\,,\ \Psi)\end{array}}$$

[Cond_true]

$$\frac{[\![\mathtt{B}]\!]_\ell = true}{\mathtt{C}(\mathtt{Q}\,,\ (\mathtt{B})\{\mathtt{S}\}\,\Sigma\,,\ \Psi)\longrightarrow\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \mathtt{S}\,\Sigma\,,\ \Psi)}$$

[Cond_false]

$$\frac{[\![\mathtt{B}]\!]_\ell = false}{\mathtt{C}(\mathtt{Q}\,,\ (\mathtt{B})\{\mathtt{S}\}\,\Sigma\,,\ \Psi)\longrightarrow\mathtt{C}(\mathtt{Q}\,,\ \ell\,,\ \Sigma\,,\ \Psi)}$$

Table 3: The transition relation of *Stipula* – Part 2

$$\begin{array}{lcll}
\mathtt{Bike\_Rental}(\_\,,\ \varnothing\,,\ \_\,,\ \_)\ ,\ \mathtt{0} & \xrightarrow{\alpha} & \mathtt{Bike\_Rental}(\mathtt{Inactive}\,,\ \ell\,,\ \_\,,\ \_)\ ,\ \mathtt{0} & [\textsc{Agree}]\\
 & \longrightarrow & \mathtt{Bike\_Rental}(\mathtt{Inactive}\,,\ \ell\,,\ \_\,,\ \_)\ ,\ 1 & [\textsc{Tick}]\\
 & \xrightarrow{\mathtt{Lender:offer(123)}} & \mathtt{Bike\_Rental}(\mathtt{Inactive}\,,\ \ell[z\mapsto 123]\,,\ z\rightarrow\mathtt{use\_code}\Rightarrow\mathtt{@Proposal}\,,\ \_)\ ,\ 1 & [\textsc{Function}]\\
 & \longrightarrow & \mathtt{Bike\_Rental}(\mathtt{Inactive}\,,\ \ell'\,,\ \_\Rightarrow\mathtt{@Proposal}\,,\ \_)\ ,\ 1 & [\textsc{Field\_Update}]\\
 & \longrightarrow & \mathtt{Bike\_Rental}(\mathtt{Proposal}\,,\ \ell'\,,\ \_\,,\ \_)\ ,\ 1 & [\textsc{State\_Change}]\\
 & \longrightarrow & \mathtt{Bike\_Rental}(\mathtt{Proposal}\,,\ \ell'\,,\ \_\,,\ \_)\ ,\ 2 & [\textsc{Tick}]\\
 & \longrightarrow & \mathtt{Bike\_Rental}(\mathtt{Proposal}\,,\ \ell'\,,\ \_\,,\ \_)\ ,\ 3 & [\textsc{Tick}]\\
 & \xrightarrow{\mathtt{Borrower:accept[2]}} & \mathtt{Bike\_Rental}(\mathtt{Proposal}\,,\ \ell'[y\mapsto 2]\,,\ \mathtt{S\ W}\Rightarrow\mathtt{@End}\,,\ \_)\ ,\ 3 & [\textsc{Function}]
\end{array}$$

Table 4: Initial transitions of `Bike_Rental`

(2) in an idle state, fire any ready event with a matching state. If there is one, execute its handler until the end, which is an idle state;

(3) if there is no event to be triggered in an idle state, *either* tick *or* call any permitted function (*i.e.* with matching state and preconditions). A function invocation amounts to execute its body until the end, which is again an idle state.

Therefore, we observe that *Stipula* has three sources of nondeterminism: (*i*) the order of the execution of ready event handlers, (*ii*) the order of the calls of permitted functions, and (*iii*) the delay of permitted function calls to a later time (thus, possibly, after other event handlers). For example, the contract `C` with two functions `@Q A:f{_}⇒@Q` and `@Q A′:g{_}⇒@Q` behaves as either $\xrightarrow{\mathtt{A:f}}\longrightarrow_n\xrightarrow{\mathtt{A':g}}$ or $\xrightarrow{\mathtt{A':g}}\longrightarrow_n\xrightarrow{\mathtt{A:f}}$, where $\longrightarrow_n$ are transitions that make the time elapse (rule [Tick]). As another example, consider a contract `C′` with a function `@Q A:f {_ now ≫@Q′{ "hello"→A }⇒@Q′ }⇒@Q` and a function `@Q A′:g {_}⇒@Q′`. Then it may behave as either $\xrightarrow{\mathtt{A:f}}\xrightarrow{\mathtt{A':g}}\xrightarrow{\mathtt{''hello''}\rightarrow\mathtt{A}}$ or as $\xrightarrow{\mathtt{A:f}}\longrightarrow_n\xrightarrow{\mathtt{A':g}}$, after which the action $\xrightarrow{\mathtt{''hello''}\rightarrow\mathtt{A}}$ is disabled, or as $\xrightarrow{\mathtt{A':g}}$, which precludes the call of `f`.

Remark. The semantics of *Stipula* may be easily extended to configurations with several smart legal contracts. It is sufficient to consider configurations as consisting of sets of runtime contracts and to change the rule [Tick]. To illustrate the general case, let $\mathbb{C} = \mathtt{C}_1(\Phi_1\ ,\ \ell_1\ ,\ \Sigma_1\ ,\ \Psi_1), \cdots, \mathtt{C}_n(\Phi_n\ ,\ \ell_n\ ,\ \Sigma_n\ ,\ \Psi_n)$. We define $Events(\mathbb{C}) \overset{\text{def}}{=} \Psi_1 \mid \cdots \mid \Psi_n$. The new rule [Tick] becomes

$$\text{[Tick+]}\quad \frac{Events(\mathbb{C})\ ,\ \mathbb{t} \not\rightarrow}{\mathbb{C}\ ,\ \mathbb{t} \longrightarrow \mathbb{C}\ ,\ \mathbb{t}+1}$$

5 *STIPULA* LAWS AND EQUATIONAL THEORY

The operational semantics of Tables 2 and 3 is too intensional because it defines the behaviour of a legal contract without showing any evidence of the differences between contracts. Nevertheless, it is the base for defining a more appropriate, extensional semantics using a standard technique based on observations [19]. According to this technique, two contracts cannot be separated if a party using them cannot distinguish one form the other. Said otherwise, a party can differentiate two contracts if he can *observe different interactions*. It turns out that defining the observations is a critical point of the overall technique, because it allows to fine-tune the discriminating power of the extensional semantics. The appropriate observations for smart legal contracts match the design principles of *Stipula*: let A be a party, then

- A should observe the *agreement* that he is going to sign, because stipulating a different agreement may be unsatisfactory; therefore we envisage the observations A SET $[\mathtt{x} \mapsto v]$ and A OK $[\mathtt{x} \mapsto v]$;
- A should also observe the *permission* or the *prohibition* to invoke a functionality at a given time $\mathbb{t}$, *i.e.* whether $A : \mathtt{f}(\overline{v})[\overline{w}]$ is possible at $\mathbb{t}$ or not;
- A should finally observe whether at $\mathbb{t}$ he can *receive* a value or an asset, *i.e.* whether $v \rightarrow A$ or $v \multimap A$ are possible at $\mathbb{t}$ or not.

The ordering of invocations and receives can be safely overlooked, as long as they belong to the same block of transactions, that is they are executed at the same global time. Notice also that the above observations allow a party to observe contract's *obligations*. Indeed, by shifting the observation at a specific point in time, one can observe the effects of executing the event that encodes a legal commitment, such as the issue of a sanction or the impossibility to do further actions. On the other hand, the foregoing notion of observations abstracts away from the names of the contract's assets and internal states.

We will use the following notations:

- Let R be either SET or OK, then we write A R $[\mathtt{x} \mapsto v] \in \alpha$ if α contains A R $[\overline{\mathtt{z}} \mapsto \overline{w}]$ and $[\mathtt{x} \mapsto v]$ occurs in the sequence of assignments $[\overline{\mathtt{z}} \mapsto \overline{w}]$. We write $\alpha \sim \alpha'$ whenever A R $[\mathtt{x} \mapsto v] \in \alpha$ if and only if A R $[\mathtt{x} \mapsto v] \in \alpha'$. Intuitively, α and α' express the same agreement up to reordering of the field assignemnts.
- Let be $\overset{\mu}{\Longrightarrow} \overset{\text{def}}{=} \Longrightarrow \overset{\mu}{\longrightarrow} \Longrightarrow$, where $\Longrightarrow$ stands for any number of $\longrightarrow$ transitions, possibly zero.

The following definition of legal contract equivalence compares the observable behavior of contracts. It is defined over configurations, so to appropriately shift the time of the contract's observations. The equivalence is defined as a suitable bisimulation game that is consistent with the idea that in blockchain systems the interactions are batched in blocks of transactions.

Definition 5.1 (Legal Bisimulation). A symmetric relation $\mathcal{R}$ is a legal bisimulation between two configurations at time $\mathbb{t}$, written $\mathbb{C}_1, \mathbb{t}\ \mathcal{R}\ \mathbb{C}_2, \mathbb{t}$, whenever

(1) if $\mathbb{C}_1, \mathbb{t} \overset{\alpha}{\Longrightarrow} \mathbb{C}'_1, \mathbb{t}$ then $\mathbb{C}_2, \mathbb{t} \overset{\alpha'}{\Longrightarrow} \mathbb{C}'_2, \mathbb{t}$ for some α' such that $\alpha \sim \alpha'$ and $\mathbb{C}'_1, \mathbb{t}\ \mathcal{R}\ \mathbb{C}'_2, \mathbb{t}$;
(2) if $\mathbb{C}_1, \mathbb{t} \overset{\mu_1}{\Longrightarrow} \cdots \overset{\mu_n}{\Longrightarrow} \mathbb{C}'_1, \mathbb{t} \longrightarrow \mathbb{C}'_1, \mathbb{t}+1$ then there exist $\mu'_1 \cdots \mu'_n$ that is a permutation of $\mu_1 \cdots \mu_n$ such that $\mathbb{C}_2, \mathbb{t} \overset{\mu'_1}{\Longrightarrow} \cdots \overset{\mu'_n}{\Longrightarrow} \mathbb{C}'_2, \mathbb{t} \longrightarrow \mathbb{C}'_2, \mathbb{t}+1$ and $\mathbb{C}'_1, \mathbb{t}+1\, \mathcal{R}\, \mathbb{C}'_2, \mathbb{t}+1$.

Let $\simeq$ be the largest legal bisimulation, called *bisimilarity*. When the initial configurations of contracts C and C′ are bisimilar, we simply write $\mathtt{C} \simeq \mathtt{C}'$.

Being a symmetric relation, a legal bisimulation compares both contracts' permissions and prohibitions: if C permits an action (*i.e.* exhibits an observation), then C′ must permit the same action, and if C prohibits an action (*i.e.* does not exhibit a function call or an external communication), then also C′ must not exhibit the corresponding observation. Moreover, the bisimulation game enforces a *transfer property*, that is it shifts the time of observation to the future, so to capture and compare the changes of permissions/prohibitions and the (future) obligations. Observe that the equivalence abstracts away the ordering of the observations within the same time clock, since in a blockchain there is no strong notion of ordering between the transactions contained in the same block. Nevertheless, specific orderings of function invocations are important in *Stipula* contracts and the equivalence cannot overlook essential precedence constraints. For instance, the requirement that a function delivering a service can only be invoked after another specific function, say a payment. This is indeed the case for the legal bisimulation. To explain, consider the contract C with two functions `@Q A:f{_}⇒@Q` and `@Q A′:g{_}⇒@Q` and the contract C′ with two functions `@Q A:f{_}⇒@Q′` and `@Q′ A′:g{_}⇒@Q`. In C the functions can be called in any order, while in C′ the function g can be invoked only after f. Accordingly, $\mathtt{C} \not\simeq \mathtt{C}'$ since there is a runtime configuration of C that exhibits the observation $\overset{A':\mathrm{g}}{\Longrightarrow}$, while it is not the case for the contract C′, since at any time it can only exhibit $\overset{A:\mathrm{f}}{\Longrightarrow}\overset{A':\mathrm{g}}{\Longrightarrow}$.

The following theorem highlights the property that the internal state of the contract is abstracted away by the extensional semantics, which only observes the external contract's behavior. The proof is omitted because it is standard.

Theorem 5.2 (Internal refactoring). *Let* C *and* C′ *be two contracts that are equal up-to a bijective renaming of states. Then* $\mathtt{C} \simeq \mathtt{C}'$. *Similarly, for bijective renaming of assets and contract names.*

Bisimilarity is also independent from *future* clock values. This allows us to garbage-collect events that cannot be triggered anymore because the time for their scheduling is already elapsed.

Theorem 5.3 (Time shift).

(1) *If* $\mathbb{C}, \mathbb{t} \simeq \mathbb{C}', \mathbb{t}$ *and* $\mathbb{t} \leqslant \mathbb{t}'$, *then* $\mathbb{C}, \mathbb{t}' \simeq \mathbb{C}', \mathbb{t}'$.
(2) *If* $\mathbb{t} < \mathbb{t}'$ *then* $\mathtt{C}(\mathtt{Q}\ ,\ \ell\ ,\ \Sigma\ ,\ \Psi \mid \mathbb{t} \gg \mathtt{@Q}\ \{\ \mathtt{S}\ \} \Rightarrow \mathtt{@Q}')\ ,\ \mathbb{t}' \simeq \mathtt{C}(\mathtt{Q}\ ,\ \ell\ ,\ \Sigma\ ,\ \Psi)\ ,\ \mathbb{t}'$.

We finally put forward a set of algebraic laws that formalize the fact that the ordering of remote communications can be safely overlooked, as long as they belong to the same transaction. The laws are defined over statements, therefore, let $\mathsf{C}[\,]$ be a *context*, that is a contract that contains an hole where a statement may occur. We write $\mathsf{S} \simeq \mathsf{S'}$ if, for every context $\mathsf{C}[\,]$, $\mathsf{C}[\mathsf{S}] \simeq \mathsf{C}[\mathsf{S'}]$.

Theorem 5.4. *The following non-interference laws hold in Stipula (whenever they are applicable, we assume* $\mathsf{x} \notin \mathrm{fv}(\mathsf{E'})$ *and* $\mathsf{x'} \notin \mathrm{fv}(\mathsf{E})$ *and* $\mathsf{h} \notin \mathrm{fv}(\mathsf{E'})$ *and* $\mathsf{h'} \notin \mathrm{fv}(\mathsf{E'})$ *and* $\mathsf{h''} \notin \mathrm{fv}(\mathsf{E})$ *and* $\mathsf{h'''} \notin \mathrm{fv}(\mathsf{E})$*):*

$$\begin{array}{rcl}
\mathsf{E} \rightarrow \mathsf{A}\ \ \mathsf{E'} \rightarrow \mathsf{A'} & \simeq & \mathsf{E'} \rightarrow \mathsf{A'}\ \ \mathsf{E} \rightarrow \mathsf{A} \\
\mathsf{E} \rightarrow \mathsf{x}\ \ \mathsf{E'} \rightarrow \mathsf{A} & \simeq & \mathsf{E'} \rightarrow \mathsf{A}\ \ \mathsf{E} \rightarrow x \\
\mathsf{E} \rightarrow \mathsf{x}\ \ \mathsf{E'} \rightarrow x' & \simeq & \mathsf{E'} \rightarrow \mathsf{x'}\ \ \mathsf{E} \rightarrow \mathsf{x} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{A}\ \ \mathsf{E'} \rightarrow \mathsf{A'} & \simeq & \mathsf{E'} \rightarrow \mathsf{A'}\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{A} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{A}\ \ \mathsf{E'} \rightarrow \mathsf{x'} & \simeq & \mathsf{E'} \rightarrow \mathsf{x'}\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{A} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{h'}\ \ \mathsf{E'} \rightarrow \mathsf{A} & \simeq & \mathsf{E'} \rightarrow \mathsf{A}\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{h'} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{h'}\ \ \mathsf{E'} \rightarrow \mathsf{x'} & \simeq & \mathsf{E'} \rightarrow \mathsf{x'}\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{h'} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{A}\ \ \mathsf{E'} \multimap \mathsf{h''}, \mathsf{A'} & \simeq & \mathsf{E'} \multimap \mathsf{h''}, \mathsf{A'}\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{A} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{A}\ \ \mathsf{E'} \multimap \mathsf{h''}, \mathsf{h'''} & \simeq & \mathsf{E'} \multimap \mathsf{h''}, \mathsf{h'''}\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{A} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{h'}\ \ \mathsf{E'} \multimap \mathsf{h''}, \mathsf{h'''} & \simeq & \mathsf{E'} \multimap \mathsf{h''}, \mathsf{h'''}\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{h'}
\end{array}$$

6 IMPLEMENTATION

The semantics of Section 4 has not only a theoretical interest but it also drives us in the design of the implementation of *Stipula*. While a full discussion about this subject is out of the scope of the present work, we consider the three features that make the implementation of *Stipula* not trivial: agreements, assets, and events. (In this paper, we are overlooking issues regarding errors and backtracks.) Below, we analyze how they can be actually encoded in a lower level, Solidity-like language, that can be more directly implemented in the bytecode of some virtual machine running a mainstream smart contract language (*e.g.* Ethereum Virtual Machine [25], Move Virtual Machine [11]).

Agreements. The agreement code is technically a *multiparty synchronization* that expresses a consensus between the parties to start the contract with particular values of the (non-linear) fields. This construct can be implemented by resorting to a barrier-like protocol, where each party A_i may call, in whatever order, a specific function to propose the values he agrees on, and the barrier eventually checks the consistency of the proposed values before moving the contract to the initial state. The following snippet of Solidity-like code corresponds to the agreement code (see Section 3), where we assume that the fields of the contract are $\mathsf{x}_1,\cdots,\mathsf{x}_k$ with types $\mathsf{T}_1,\cdots,\mathsf{T}_k$, respectively.

```
address A_1 , ... , A_n ;
T_1 x_1; ... ; T_k x_k ;
enum State {Nothing, Q}
State state = State.Nothing ;
int counter = 1 ;

T_{j_1} aux_i_x_{j_1};...; T_{js_i} aux_i_x_{js_i};          // 1 ⩽ i ⩽ n
bool use_once_i = true ;                                      // 1 ⩽ i ⩽ n

function set_ok_i(T_{i_1} z_{i_1},...,T_{ir_i} z_{ir_i},T_{j_1} z_{j_1},...,T_{js_i} z_{js_i}){
                                                              // 1 ⩽ i ⩽ n
   if (sender == A_i && use_once_i){
        use_once_i = false ;
        x_{i_1} = z_{i_1} ; ... ; x_{ir_i} = z_{ir_i} ;
        aux_i_x_{j_1} = z_{j_1};...; aux_i_x_{js_i} = z_{js_i};
        if (counter == n){
           if (⋀_{1⩽i⩽n}(aux_i_x_{j_1}==x_{j_1} &&...&& aux_i_x_{js_i}==x_{js_i})
              ) state = State.Q ;
           else throw error;
        } else counter = counter + 1 ;
   } else throw error;
}
```

Each function $\mathtt{set\_ok}_i$ can be called only once by the party A_i, with two lists of parameters: the first r_i values are used to set the contract's fields $\mathsf{x}_{i_1},\cdots,\mathsf{x}_{i_{r_i}}$. The last s_i values are recorded into the auxiliary fields $\mathtt{aux}_i\mathtt{\_x}_{j_1},\cdots,\mathtt{aux}_i\mathtt{\_x}_{j_{s_i}}$, to express that A_i agrees *ex ante* with anyone setting the contract's fields $\mathsf{x}_{j_1},\cdots,\mathsf{x}_{j_{s_i}}$ to the values $\mathsf{z}_{j_1},\ \ldots,\ \mathsf{z}_{j_{s_i}}$. When all the parties have done the agreement, *i.e.* `counter` is equal to n, a check on the consistency of $\mathtt{aux}_i\mathtt{\_x}_h$ is performed and, in case it succeeds, the contract becomes active moving to the state Q. The snippet also shows that contract's states can be easily mapped to enumerations, as usual in the Solidity State Machine pattern.

There is a discrepancy between the above code and the semantics of the agreement in Table 2. While the agreement has a transactional nature (it may occur as a whole or not), the above Solidity protocol takes time, *i.e.* it is performed in several blocks and, in any block, a failure may occur. In this case, there is a backtrack to the *initial state of the block* and not to the initial state of the protocol, as it happens in *Stipula*. Therefore, the error management should take care of removing partial values stored in the fields of the contract. Nevertheless, another source of discrepancy seems more awkward: the gas consumption. In fact, the successful termination of the agreement as well as the failed one have a cost in Ethereum, while it is not the case. To bridge this gap, we should design agreements with fees payed by parties that are used for the consensus. We have not yet studied these details, which are postponed to future work.

Assets. Assets are linear resources that cannot be duplicated or leaked: when a resource value is assigned, the location previously holding the value is emptied. The community of smart contracts has already recognized the relevance of having linear resources as first-class entities in programming languages because they can significantly simplify programming and the effort required for verification. *Stipula* features a simple abstraction to manage assets, which is used to represent both currency and indivisible tokens. To implement these assets we can resort to the popular token standards on Ethereum (ERC-20 for virtual currency and ERC-721 for non fungible tokens[13]). Alternatively, we can rely on the Move language, whose designers have featured programmable linear resources by constraining them to adhere to ad-hoc rules specified by its declaring module [7, 11]. Using a pseudo-code inspired to Move, we might define (divisible) assets h and $\mathsf{h'}$ as resources of type H, so that the operations $\mathsf{E} \multimap \mathsf{h}, \mathsf{h'}$ and $\mathsf{E} \multimap \mathsf{h}, \mathsf{A}$ may be encoded by $\mathtt{h.move(E,h')}$ and $\mathtt{h.withdraw(E,A)}$, according to the definitions below:

```
resource H {
    T amount ;

    function move(T x, H h) {
         (x <= amount){ h = h + x ; amount = amount - x ; }
    }
    function withdraw(T x, address A) {
         (x <= amount){ A.send(x) ; amount = amount - x ; }
    }
    constructor(T x){ amount = x ; }
}
```

Events. The current blockchain technologies do not admit the record of statements that have to be performed afterwards in a future transaction (with one exception to our knowledge: Cardano, see Section 8). The standard technique adopted in Ethereum to circumvent this limitation is based on the Solidity's *event* construct and off-chain *oracle* services. To explain, the *Stipula* event `E ≫ @Q { S } ⇒ @Q′` emitted by a function `foo` can be mapped to the Solidity code:

```
event R(uint time, address lc) ;
function call_back_R() external {
    if (state == State.Q) { S ; state = State.Q'; }
}
function foo(T1 u1,..., Tn un){
    ...
    emit R(E,address(this)) ;
}
```

The Solidity function `foo` emits an event named `R` carrying the time `E` and the address of the issuing contract. Moreover, an external *DApp service* – an oracle – scans the blockchain looking for `R` events and calls the `call_back_R` function of the contract at the appropriate time `E`. Other more complex and safer protocols can be used. However a code external to the Ethereum blockchain is always necessary in order to trigger the scheduled event handler. A more satisfactory implementation of the *Stipula* events might adopt ideas taken from the implementation of timeouts in the Marlowe and Findel languages for financial contracts [21, 24] as discussed in Section 8.

7 EXPRESSIVITY OF *STIPULA*

Stipula has been devised for writing legal contracts in a formal and intelligible (to lawyers) way. In this section we analyze the expressivity of *Stipula* by writing the contracts for a set of archetypal acts, ranging from renting to (digital) licenses and to bets. We conclude this analysis with a table that summarizes the legal elements of the archetypal acts and the programming abstractions that we have used to express them in *Stipula*.

7.1 The free rent contract

The free rent is the simplest kind of legal contract. It involves two parties, the lender and the borrower, which initially agree about what good is rented, what use should be made of it, the time limit (or in which case it must be returned), the estimated of value and any defects in the good. Upon agreement, the delivery of the good triggers the legal bond, that is the borrower has the permission to use the good and the lender has the prohibition of preventing him from doing so. Note that there is no transfer of ownership, but only the right to use the good. The contract terminates either when the borrower returns the good, or when the time limit is reached. Litigations could arise when the borrower violates the obligations of diligent storage and care, the obligations of using the good only as intended, and not granting the use to a third party without the lender's consent. In these cases the lender may demand the immediate return of the object, in addition to compensation for the damage. On the other hand, the borrower is entitled to compensation if the good has defects that were known to the lender but that he did not initially disclose.

The free rent contract puts forward the following points:

- When a legal contract refers to a *physical* good, the smart contract needs a digital handle (an avatar) for that good. Many technological solutions, such as smart locks of IoT devices, are actually available. In *Stipula* we abstract from the specific nature of such a digital handle, and we simply represent it as an `asset`, which intuitively corresponds to a non fungible token associated to the physical good.
- The rent legal contract grants just the *usage* of a good without the transfer of ownership. Accordingly, while the communication of the token provides full control of the associated physical good, we assume an operation `uses(token)` (resp. `use_once(token)` or `uses(token,A)`) that generates a usage-code providing access to the object associated to the token (resp. a usage-code only valid (once) for the party `A`).
- In a legal rent contract it is important to acknowledge the delivery of the good, since this is the action that triggers the legal bonds. We rely on assets and their *semantics* to implement this feature.

The *Stipula* code for the free rent of a locker is written in Listing 2. The two parties agrees on the time limit for the locker usage (`time_limit`) and the time limit to start the usage (`time_start`). Contract's states allow one sequence of actions: first `Lender` sends the number `n` of the locker and the token `t` associated to `n` by calling `box_proposal` (line 10). This action moves the contract to the temporary state `Proposal` and schedule the event in line 13. This event is essential to prevent the unique token associated to the locker to be indefinitely locked-in in the smart contract when the borrower never calls the `boxUse` function to finalize the delivery of the good. If `Borrower` calls the function `boxUse` (line 16) within the timeout `time_start`, then the number of the rented locker and the access code are returned. At the same time, a second timeout is installed to check the time limit for the locker usage, and the final state change to `Using` (line 21) vanishes the timeout installed in line 13. This second event is needed to prevent a never ending use of the locker. If `time_limit` is reached and the contract's state is still `Using`, then (lines 18-19) a message is sent to `Borrower` and the token is sent back to `Lender`, which becomes again in full control of the locker and can thus invalidate the access code held by the borrower. Otherwise, the rent contract can terminate because the borrower explicitly returns the good before the time limit. This is represented by a call of the function `returnBox` (line 23).

```
1  legal_contract Free_Rent {
2    assets token
3    fields numBox, t_start, t_limit
4
5    agreement (Lender, Borrower) {
6         Lender =SET=> t_start, t_limit
7         Borrower =OK=> t_start, t_limit
8    } ⇒ @Inactive
9
10   @Inactive Lender :  boxProposal (n)[t] {
11        t ⊸ token
12        n → numBox
13        now + t_start ≫ @Proposal {
14              token ⊸ Lender } ⇒ @End
15   } ⇒ @Proposal
16
17   @Proposal Borrower :  boxUse {
18        (uses(token), numBox) → Borrower
19        now + t_limit  ≫ @Using {
20              "Time_Limit_Reached" → Borrower
```

```
            token ⊸ Lender
        } ⇒ @End
    } ⇒ @Using

    @Using Borrower : returnBox {
        token ⊸ Lender
    } ⇒ @End
}
```

Listing 2: The rent for free contract

The smart legal contract of Listing 2 does not consider the compensations that would be needed to deal with the disputes due to breaches of the contract, such as the borrower's diligent care during the locker's usage. These violations require off-chain monitoring and a dispute resolution mechanism. The next example illustrates how this off-chain monitoring and enforcement can be combined with the on-chain code.

7.2 The Digital Licensee contract

Let us consider a contract corresponding to a licence to access a digital service, like a software or an ebook: the digital service can be freely accessed for a while, and can be permanently bought with an explicit communication within the evaluation period (for a similar example, see [16]). The licensing contractual clauses can be described as follows:

Article 1. `Licensor` grants `Licensee` for a licence to evaluate the `Product` and fixes (*i*) the *evaluation period* and (*ii*) the *cost* of the `Product` if `Licensee` will bought it.

Article 2. `Licensee` will pay the `Product` in advance; he will be reimbursed if the `Product` will not be bought with an explicit communication within the evaluation period. The refund will be the 90% of the cost because the 10% is payed to the `Authority` (see Article 3).

Article 3. `Licensee` must not publish the results of the evaluation during the evaluation period and `Licensor` must reply within 10 hours to the queries of `Licensee` related to the `Product`; this is supervised by `Authority` that may interrupt the licence and reimburse either `Licensor` or `Licensee` according to whom breaches this agreement.

Article 4. This license will terminate automatically at the end of the evaluation period, if the licensee does not buy the product.

Compared to the previous example, this contract involves payment and refund: an amount of currency is escrowed, and two parts of it will be sent to different parties, the `Authority` and either the `Licensor` or the `Licensee`. *Stipula* provides the general `asset` abstraction, together with a general operation to move just a (positive) subset of the asset to a different owner. This is exactly what is needed to deal with currency, therefore the *Stipula* licence contract holds two different assets: an indivisible `token` providing an handle to the digital service, and a `balance` that is a divisible asset corresponding to the amount of currency kept in custody inside the smart contract.

A further important feature of the contract is Article 3 that defines specific constraints about the *off-chain* behaviour of `Licensor` and `Licensee`. This exemplifies the very general situations where contract's violations cannot be fully monitored by the on-chain software, such as the publication of a post in a social network, or the leakage of a secret password, or the violation of the obligation of diligent storage and care. In all these cases, it is required a trusted third party, say an `Authority`, to supervise the disputes occurring from the off-chain monitoring and to provide a trusted dispute resolution mechanism. The code in Listing 3 illustrates the encoding of the off-chain monitoring and enforcement mechanism with the on-chain smart contract code in *Stipula*.

```
1  legal_contract Licence {
2    assets token, balance
3    fields cost, t_start, t_limit
4
5    agreement (Licensor,Licensee,Authority) {
6      Licensor =SET=> cost, t_start, t_limit
7      Licensee =OK=> cost, t_start, t_limit
8      Authority =OK=> _
9    } ⇒ @Inactive
10
11   @Inactive Licensor : offerLicence [t] {
12     t ⊸ token
13     now + t_start ≫ @Proposal {
14               token ⊸ Licensor } ⇒ @End
15   } ⇒ @Proposal
16
17   @Proposal Licensee : activateLicence [b]
18     (b == cost){
19        b ⊸ balance
20        balance*0,1 ⊸ balance, Authority
21        uses(token,Licensee) → Licensee
22        now + t_limit ≫ @Trial {
23                       balance ⊸ Licensee
24                       token ⊸ Licensor
25                     } ⇒ @End
26   } ⇒ @Trial
27
28   @Trial Licensee : buy {
29     balance ⊸ Licensor
30     token ⊸ Licensee
31   } ⇒ @End
32
33   @Trial Authority : compensateLicensor {
34     balance ⊸ Licensor
35     token ⊸ Licensor
36   } ⇒@End
37
38   @Trial Authority : compensateLicensee {
39     balance ⊸ Licensee
40     token ⊸ Licensor;
41   } ⇒ @End
42 }
```

Listing 3: The contract for a digital licence

The agreement of Listing 3 involves three parties: `Licensor`, which fixes the parameters of the contract, according to Article 1., `Licensee`, which explicitly agrees, and `Authority`, which does not need to agree upon the contracts' parameters (*i.e.* the emptyset agreement noted _), but it is important that it is involved in the agreement synchronization, because it plays the role of the trusted third party that is entitled to call the functions `compensateLicensor` and `compensateLicensee`.

In `activateLicence`, the caller, *i.e.* the `Licensee`, is required to send an amount of assets equal to the fixed cost of the license. Notice the difference between line 19 and line 20: the first one is the move of a fraction of asset towards the authority, while the second is the simple communication to `Licensee` of a personal usage code associated to the token. Once entered in the `Trial` state, the contract can terminate in three ways: (*i*) the licensee expresses its willingness to buy the licence by calling the function `call` which grants him the full token, or (*ii*) the time limit for the free evaluation period is reached, thus the scheduled event refunds the licensee and gives the token back to the licensor, or (*iii*) during the evaluation period a violation to Article 3 is identified and the authority pre-empts the license by calling either the function `compensateLicensor` or `compensateLicensee`. Observe that it is important that the code guarantees that, in all the possible cases, the assets, both the token and the balance, are not indefinitely locked-in the contract.

7.3 A bet contract

The bet contract is a simple example of a legal contract that contains an element of randomness (*alea*), *i.e.* where the existence of the performances or their extent depends on an event which is entirely independent of the will of the parties. The main element of the contract is a future, aleatory event, such as the winner of a football match, the delay of a flight, the future value of a company's stock.

A digital encoding of a bet contract requires that the parties explicitly agree on the source of data that will determine the final value of the aleatory event (the *Data Provider*), that is a specific online service, an accredited institution, or any trusted third party. It is also important that the digital contract provides precise time limits for accepting payments and for providing the actual value of the aleatory event. Indeed there can be a number of issues: the aleatory event does not happen, *e.g.* the football match is cancelled, or the data provider fails to provide the required value, *e.g.* the online service is down.

```
1  legal_contract Alea {
2    assets bet1, bet2
3    fields alea_fact, val1, val2, data_source,
4           fee, amount, t_before, t_after
5
6    agreement(Better1,Better2,DataProvider){
7      DataProvider =SET=> fee, data_source
8      DataProvider =OK=> t_after, alea_fact
9      Better1 =SET=> alea_fact, amount, t_before, t_after
10     Better1 =OK=> fee, data_source
11     Better2 =OK=> alea_fact, amount, fee, t_before, t_after,
12                   data_source
13   } ⇒ @Init
14
15   @Init Better1 : place_bet(x)[y]
16     (y == amount){
17         y ⊸ bet1
18         x → val1
19         t_before ≫ @First { bet1 ⊸ Better1 } ⇒ @Fail
20     } ⇒ @First
21
22   @First Better2: place_bet(x)[y]
23     (y == amount){
24         y ⊸ bet2
25         x → val2
26         t_after ≫ @Run {
27             bet1 ⊸ Better1
28             bet2 ⊸ Better2 } ⇒ @Fail
29   } ⇒ @Run
30
31   @Run DataProvider : data(x,z)[]
32     (x==alea_fact){
33         (z==val1){               // The winner is Better1
34             fee ⊸ bet2,DataProvider
35             bet2 ⊸ Better1
36             bet1 ⊸ Better1
37         }
38         (z==val2){               // The winner is Better2
39             fee ⊸ bet1,DataProvider
40             bet1 ⊸ Better2
41             bet2 ⊸ Better2
42         }
43         (z != val1 && z != val2){       //No winner
44             bet1 ⊸ DataProvider
45             bet2 ⊸ DataProvider
46         }
47     } ⇒ @End
48 }
```

Listing 4: The contract for a bet

The *Stipula* code in Listing 4 corresponds to the case where `Better1`, respectively `Better2`, places `val1`, respectively `val2`, a bet corresponding to the agreed `amount` of currency, stored in the contract's assets `bet1` and `bet2` respectively[1]. Observe that both bets must be placed within an (agreed) time limit `t_before` (line 17), to ensure that the legal bond is established before the occurrence of the aleatory event. The second timeout, scheduled in line 24, is used to ensure the contract termination even if the `DataProvider` fails to provide the expected data, through the call of the function `data`.

Compared to the Digital Licence in Listing 3, the role of the `DataProvider` here is less pivotal than that of the `Authority`. While it is expected that `Authority` will play its part, `DataProvider` is much less than a peer of the contract. It is sufficient that it is an independent party that is entitled to call the contract's function to supply the expected external data. The crucial point of trust here is the `data_source`, not the `DataProvider`. In other terms, since the parties involved in the agreement need not to trust each other, it might happen that `DataProvider` supplies an incorrect value through the function `data`. In this case, the betters can appeal against the data provider since they agreed upon the data emitted by the `data_source`. As usual, any dispute that might render the contract voidable or invalid, *e.g.* one better knew the result of the match in advance, can be handled by adding to the code of an authority party, according to the pattern illustrated in the Digital Licence example.

7.4 Specification patterns in *Stipula*

The clauses of the foregoing legal acts have been specified in *Stipula* by means of patterns that, in our view, are common from the juridical point of view. This is summarized in Table 5.

8 RELATED WORKS

A number of projects have put forward legal markup languages, to wrap logic and other contextual information around traditional legal prose, and providing templates for common contracts. OpenLaw [26] also allows to reference Ethereum-based smart contracts into legal agreements, and automatically trigger them once the agreement is digitally signed by all parties. Signatures by all relevant parties are stored on IPFS (the Inter-Planetary File System) and the Ethereum blockchain. The Accord project [22] provides an open, standardized format for smart legal contracts, consisting of natural language and computable components. These contracts can

[1]For simplicity, this code requires `Better1` to place its bet before `Better2`, however it is easy to add similar function to let the two bets be placed in any order.

Example	**Legal clauses**	**Smart Encoding in *Stipula***
Free rent	Permission and Prohibition	States of the contract to allow or prevent the call of a function
	Obligation to return the good	Event: timeout that triggers a repercussion
	Access to a physical good without transfer of ownership	(non fungible) Token: transfer only a usage code associated to the Token, i.e. operation `uses(token,L)`$\rightarrow$ `L`
	Prohibition of preventing the usage	Token held in custody in the smart contract
Digital License	Permission, Prohibition, and Obligation	States and Events
	Currency and escrow	(divisible) Asset
	Usage and purchase	Token: either transfer only an associated, and personal, usage code, or transfer the token, i.e. `uses(token,L)`$\rightarrow$ `L` and `token`$\multimap$ `L`
	Off-chain constraints and Authority for dispute resolution	*Implicitly* trusted party included in agreement
Bet	Commitment and Obligation	Event
	Aleatory value	Event: timeout to decide the effective value
	Authority that decides the value	*Explicitly* trusted party included in agreement

Table 5: Legal clauses of the archetypal legal acts and their encodings in *Stipula*

then be interpreted by machines but they do not necessarily operate on blockchains. These projects come with sets of templates for standard legal contracts, that can be customized by setting template's parameters with appropriate values. In *Stipula*, rather than software templates, it is possible to define specific programming patterns that can be used to encode the building blocks of legal contracts. (see the Table 5). Lexon [14] uses context free grammars to define a programming language syntax that is at the same time human readable and automatically translated into, *e.g.* Solidity. Even if the high level Lexon code is very close to natural language, there is no real control over the code that is actually run: the semantics of the high level language is not defined, thus the actual behaviour of the contract is that of the automatically generated Solidity code, which might be much more subtle than that of the (much simpler and more abstract) Lexon source. Compared to the Solidity code of the Lexon examples in [15], the *Stipula* version of the same contracts is much clearer, thanks to primitives like agreement and asset movements. Thus, directly coding in *Stipula* appears to be safer than relying on the Lexon-Solidity pair. Nevertheless, it should be not difficult to design an automatic translation from Lexon to *Stipula*.

Our work aims at conducing a foundational study of legal contracts, in order to elicit a precisely defined set of building blocks that can be used to describe, analyse and execute (thus enforce) legal agreements. This is similar to what has been done in [17], which puts forward a set of combinators expressing financial and insurance contracts, together with a denotational semantics and algebraic properties that says what such contracts are worth. These ideas have been ported on the (Cardano) blockchain by the Marlowe language [5], which is a small domain specific language featuring constructs like participants, tokens, currency and timeouts to wait until a certain condition becomes true (similarly to *Stipula*).

In Marlowe, money cannot be locked forever in a contract because financial contracts have a finite lifetime, at the end of which any remaining money is returned to the participants. In *Stipula*, the finite lifetime of a contract may be easily programmed by arranging its value during the agreement phase, and issuing a corresponding event in the initial state. Marlowe semantics is written in Haskell and is executed on the Cardano blockchain almost directly because the bytecode is very similar to Haskell (at the time of writing, Cardano developers are defining a bytecode language called Plutus). In the Cardano implementation of Marlowe, the lifetime is computed in terms of slot numbers, after which a transaction is issued that closes the contract (refund all the money in its account) [21]. It turns that Cardano leverages scripts that express conditions on the time [5]. This means that *Stipula* events can be implemented in Cardano by using slot numbers, as well. However, porting an imperative language like *Stipula* to Cardano amounts to writing an interpreter/compiler that manages contract executions following a continuation-passing style. Overall, this is an issue that merits future investigations.

The class-based programming style of *Stipula* is similar to that of Solidity, but there are many differences between the two. Actually, *Stipula* is much similar to Obsidian [9], which is based on state-oriented programming and explicit management of linear assets, whose usability has been experimentally assessed [8]. Obsidian has a type system that ensures the correct manipulations of objects according to their current states and that linearly typed assets are not accidentally lost. On the contrary, *Stipula* is untyped: the introduction of a type discipline is orthogonal and is postponed to a later stage where we plan to investigate static analysis techniques specifically suitable to the legal setting. As a cons, Obsidian has no agreement nor event primitives, therefore the consensus about the contract's terms and the enforcing of legal obligations must be implemented in a much more indirect way.

9 CONCLUSIONS

This paper presents a domain-specific language for defining legal contracts that can be automatized on a blockchain system. *Stipula* features a distilled number of operations that enable the formalisation of the main elements of juridical acts, such as permissions, prohibitions, and obligations. *Stipula* is formal and we are proud of its semantics – the legal bisimulation – that allows one to equate contracts that differ for clauses (events) that can never be triggered or for the order of non-interfering communications. Furthermore, the *Stipula* semantics has also been used for sketching an implementation on top of the Ethereum blockchain.

We believe that a range of legal arrangements can be adequately translated into *Stipula*, using simple patterns for the key elements of legal contracts (see the archetypal examples in the Appendix). Nevertheless we acknowledge that legal contracts cannot be fully replaced by *Stipula* contracts, since the formers thoroughly use the flexibility and generality of natural languages, may appeal to complex and undetermined social-normative concepts (such as fairness or good faith), may need to be revised as circumstances change, may need intelligent enforcements in case on non-compliance, etc. It is matter of (our) future research to deeply investigate these interdisciplinary aspects and provide at lest partial solutions.

From the computer science point of view, a number of issues deserve to be investigated in full detail: the extension of the language with operations for failures, devising linear type systems that enforce the partial correctness of *Stipula* codes, the implementation of the language, the definition of a (formal) translation in *Stipula* of a high level language, such as Lexon or part of it, in order to relieve lawyers from understanding computer science jargons.

Overall, we are optimistic that future research on *Stipula* can satisfactorily address the above issues because its model is simple and rigorous, which are, in our opinion, fundamental criteria for reasoning about legal contracts and for understanding their basic principles. In our mind, *Stipula* is the backbone of a framework where addressing and studying other, more complex features that are drawn from juridical acts.

ACKNOWLEDGMENTS

Giovanni Sartor has been supported by the H2020 European Research Council (ERC) Project "CompuLaw" (G.A. 833647). Cosimo Laneve has been partly supported by the H2020-MSCA-RISE project ID 778233 "Behavioural Application Program Interfaces (BEHAPI)". We are grateful to the law student Alessandro Parenti for his help in constructing the legal examples in Section 7.

REFERENCES

[1] Solidity Documentation: State Machine Common Pattern. https://docs.soliditylang.org/en/v0.8.0/common-patterns.html#state-machine.
[2] Malta MDIA Act. At https://mdia.gov.mt/wp-content/uploads/2018/10/MDIA.pdf, 2018.
[3] Obsidian: A safer blockchain programming language. Language Site at http://obsidian-lang.com/, 2018.
[4] Smart contract legislation and enforceability in Italy. Gazzetta Ufficiale, Law of 11 febbraio 2019, n. 12, Art. 8 ter, at https://www.gazzettaufficiale.it/eli/id/2019/02/12/19G00017/sg, 2019.
[5] Cardano Documentation. https://docs.cardano.org/, 2020.
[6] Wyoming Regulation Act. At https://www.wyoleg.gov/Legislation/2021/SF0038, 2021.
[7] Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon, and Yoni Zohar. Resources: A safe language abstraction for money. *CoRR*, 2020.
[8] Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Can advanced type systems be usable? an empirical study of ownership, assets, and typestate in obsidian. *Proc. ACM Program. Lang.*, 4(OOPSLA):132:1–132:28, 2020.
[9] Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *ACM Trans. Program. Lang. Syst.*, 42(3):14:1–14:82, 2020.
[10] A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. Resource-aware session types for digital contracts. In *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 111–126, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.
[11] Sam Blackshear et al. Move: A language with programmable resources. https://developers.diem.com/main/docs/move-paper, 2021.
[12] Giusella Finocchiaro and Chantal Bomprezzi. A legal analysis of the use of blockchain technology for the formation of smart legal contracts. *MediaLaws*, July 2020.
[13] Ethereum Foundation. Token Standards. https://ethereum.org/en/developers/docs/standards/tokens/, 2015-21.
[14] Lexon Foundation. Lexon Home Page. http://www.lexon.tech, 2019.
[15] Lexon Foundation. Lexon Demo Editor. http://demo.lexon.tech/apps/editor/, 2020.
[16] Guido Governatori, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor, and Xiwei Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law*, 26:377–409, 2018.
[17] Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering, functional pearl. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 280–292. ACM, 2000.
[18] Giuliano Lemme. Blockchain, smart contracts, privacy, o del nuovo manifestarsi della volontà contrattuale. In Giuffrè Francis Lefebvre, editor, *Privacy digitale*, pages 293–323. Italy, 2019.
[19] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
[20] Margaret Jane Radin. The deformation of contract in the information society. *Oxford Journal of Legal Studies*, 37, 2017.
[21] Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon Thompson. Marlowe: implementing and analysing financial contracts on blockchain. 2020.
[22] Open Source Contributors. The Accord Project. https://accordproject.org, 2018.
[23] Research Group on EC Private Law (Acquis Group) Study Group on a European Civil Code. *Principles, Definitions and Model Rules of European Private Law: Draft Common Frame of Reference (DCFR), Outline Edition*. Sellier, 2009.
[24] Alex BiryukovDmitry KhovratovichSergei Tikhomirov. Findel: Secure derivative contracts for ethereum. 10323, 2017.
[25] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. At https://github.com/ethereum/yellowpaper/, 2021.
[26] Aaron Wright, David Roon, and ConsenSys AG. OpenLaw Web Site. https://www.openlaw.io, 2019.

A TECHNICAL APPENDIX

Theorem 5.4. The following non-interference laws hold in *Stipula* (whenever they are applicable, we assume $\mathsf{x} \notin fv(\mathsf{E}')$ and $\mathsf{x}' \notin fv(\mathsf{E})$ and $\mathsf{h} \notin fv(\mathsf{E}')$ and $\mathsf{h}' \notin fv(\mathsf{E}')$ and $\mathsf{h}'' \notin fv(\mathsf{E})$ and $\mathsf{h}''' \notin fv(\mathsf{E})$):

$$\begin{array}{rcl}
\mathsf{E} \longrightarrow \mathsf{A}\ \ \mathsf{E}' \longrightarrow \mathsf{A}' & \simeq & \mathsf{E}' \longrightarrow \mathsf{A}'\ \ \mathsf{E} \longrightarrow \mathsf{A} \\
\mathsf{E} \longrightarrow \mathsf{x}\ \ \mathsf{E}' \longrightarrow \mathsf{A} & \simeq & \mathsf{E}' \longrightarrow \mathsf{A}\ \ \mathsf{E} \longrightarrow x \\
\mathsf{E} \longrightarrow \mathsf{x}\ \ \mathsf{E}' \longrightarrow x' & \simeq & \mathsf{E}' \longrightarrow \mathsf{x}'\ \ \mathsf{E} \longrightarrow \mathsf{x} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{A}\ \ \mathsf{E}' \longrightarrow \mathsf{A}' & \simeq & \mathsf{E}' \longrightarrow \mathsf{A}'\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{A} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{A}\ \ \mathsf{E}' \longrightarrow \mathsf{x}' & \simeq & \mathsf{E}' \longrightarrow \mathsf{x}'\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{A} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{h}'\ \ \mathsf{E}' \longrightarrow \mathsf{A} & \simeq & \mathsf{E}' \longrightarrow \mathsf{A}\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{h}' \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{h}'\ \ \mathsf{E}' \longrightarrow \mathsf{x}' & \simeq & \mathsf{E}' \longrightarrow \mathsf{x}'\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{h}' \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{A}\ \ \mathsf{E}' \multimap \mathsf{h}'', \mathsf{A}' & \simeq & \mathsf{E}' \multimap \mathsf{h}'', \mathsf{A}'\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{A} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{A}\ \ \mathsf{E}' \multimap \mathsf{h}'', \mathsf{h}''' & \simeq & \mathsf{E}' \multimap \mathsf{h}'', \mathsf{h}'''\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{A} \\
\mathsf{E} \multimap \mathsf{h}, \mathsf{h}'\ \ \mathsf{E}' \multimap \mathsf{h}'', \mathsf{h}''' & \simeq & \mathsf{E}' \multimap \mathsf{h}'', \mathsf{h}'''\ \ \mathsf{E} \multimap \mathsf{h}, \mathsf{h}'
\end{array}$$

Proof. We prove the first equality. Let be $\mathsf{S}_1 = \mathsf{E} \longrightarrow \mathsf{A}\ \ \mathsf{E}' \longrightarrow \mathsf{A}'$ and $\mathsf{S}_2 = \mathsf{E}' \longrightarrow \mathsf{A}'\ \ \mathsf{E} \longrightarrow \mathsf{A}$, and let be $\mathsf{C}_1 = \mathsf{C}[\mathsf{S}_1]$ and $\mathsf{C}_2 = \mathsf{C}[\mathsf{S}_2]$, then we need to prove that $\mathsf{C}_1(\_, \varnothing, \_, \_)\,, \mathbb{t} \simeq \mathsf{C}_2(\_, \varnothing, \_, \_)\,, \mathbb{t}$. Let also $\mathbb{C}_i = \mathsf{C}_i(\Phi, \ell, \_, \Psi[\mathsf{S}_i])$, with $i = 1, 2$, be the runtime contract where the statement S_i occurs within a number of handlers of future events. We demonstrate that the symmetric closure of the following relation is a legal bisimulation:

$$\begin{aligned}&\{\, (\mathtt{C}_1(\_,\varnothing,\_,\_)\,,\mathbb{t}\ ,\ \mathtt{C}_2(\_,\varnothing,\_,\_)\,,\mathbb{t})\,\}\\ &\cup\{\,(\mathtt{C}_1(\mathtt{Q},\ell,\_,\Psi[\mathtt{S}_1])\,,\mathbb{t}'\ ,\ \mathtt{C}_2(\mathtt{Q},\ell,\_,\Psi[\mathtt{S}_2])\,,\mathbb{t}')\\ &\quad\mid\ \text{for every } \mathtt{Q},\ell,\Psi[\,],\mathbb{t}'\,\}\end{aligned}$$

Indeed, notice that the statement $\mathtt{E}\rightarrow\mathtt{A}\ \ \mathtt{E}'\rightarrow\mathtt{A}'$ can only contribute to the behavior of $\mathtt{C}$ with a couple of transitions during the evaluation of the body of a function or the evaluation of an event handler. Therefore the statement must be completely executed with the same time clock, possibly a number k of times due to the multiple function calls and event handlers that are executed during the same time clock.

Formally, if $\mathbb{C}_1,\mathbb{t}'\xRightarrow{\mu_1}\cdots\xRightarrow{\mu_n}\mathbb{C}_1',\mathbb{t}'\longrightarrow\mathbb{C}_1',\mathbb{t}'+1$, then the sequence $\mu_1\cdots\mu_n$ contains k occurrences of the pair $v\rightarrow\mathtt{A}, v'\rightarrow\mathtt{A}'$. Similarly, there exist $\mu_1'\cdots\mu_n'$ and a configuration $\mathbb{C}_2',\mathbb{t}'$ such that $\mathbb{C}_2,\mathbb{t}'\xRightarrow{\mu_1'}\cdots\xRightarrow{\mu_n'}\mathbb{C}_2',\mathbb{t}'\longrightarrow\mathbb{C}_2',\mathbb{t}'+1$, where the sequence $\mu_1'\cdots\mu_n'$ is identical to $\mu_1\cdots\mu_n$ but for the k occurrences of the pair $v\rightarrow\mathtt{A}, v'\rightarrow\mathtt{A}'$ that has been swapped into $v'\rightarrow\mathtt{A}', v\rightarrow\mathtt{A}$. The argument also holds in the converse direction.

□