From Legal Contracts to Legal Calculi: the code-driven normativity

Silvia Crafa
Dipartimento di Matematica "Tullio Levi-Civita"
Università di Padova
Italy
silvia.crafa@unipd.it

Using dedicated software to represent or enact legislation or regulation has the advantage of solving the inherent ambiguity of legal texts and enabling the automation of compliance with legal norms. On the other hand, the so-called code-driven normativity is less flexible than the legal provisions it claims to implement, and transforms the nature of legal protection, potentially reducing the capability of individual human beings to invoke legal remedies.

In this article we focus on software-based legal contracts; we illustrate the design of a legal calculus whose primitives allow a direct formalisation of contracts' normative elements (*i.e.*, permissions, prohibitions, obligations, asset transfer, judicial enforcement and openness to the external context). We show that interpreting legal contracts as interaction protocols between (untrusted) parties enables the generalisation of formal methods and tools for concurrent systems to the legal setting.

1 Code is law, really?

Ethereum's smart contracts popularised the *Code is Law* principle[1], that is the idea of relying on software code to provide unambiguous definition and automatic execution of transactions between (mutually untrusted) parties; and when in disputes, the code of the contract, which is always publicly available, shall prevail. This principle is rooted in the blockchain's dogma that trust is hardwired into intermediary transparent algorithms. On this account, several governments have recognised that smart contracts, and more generally programs operating over distributed ledgers, may indeed have legal value [8, 6, 3].

This approach encompasses the blockchain technologies, since most of the benefits of digitally encoding legally binding agreements come from the precise definition and the automatic execution of a piece of programmable software, not necessarily operating over a blockchain. Accordingly, there is an increasing trend, called *Code-Driven Law* [5], using dedicated software to represent or enact legislation or regulation. Technologies like Rules as Code [7], Catala [9] or Akoma Ntoso [2] propose to create a machine-consumable version of some types of rules issued by governments and public administrations, *e.g.*, the tax office, student grant provision or social security agency. This helps identify potential inconsistencies in regulation, reduce the complexity and the ambiguity of legal texts and support the automation of legal decisions by the code-driven enforcement of rules: instead of relying on *ex-post* enforcement by third parties (*i.e.*, courts and police), the rules hardwired into code are enforced *ex-ante*, making it very difficult for people to breach them in the first place [26].

However, transposing legal rules into technical rules is a delicate process, since the inherent ambiguity of the legal system is necessary to ensure a proper application of the law on a case-by-case basis. Regulation by code is instead always more specific and less flexible than the legal provisions it purports

[1] originally proposed by Lawrence Lessing [29]

V. Castiglioni and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics 2022 (EXPRESS/SOS 2022).
EPTCS 368, 2022, pp. 23–42, doi:10.4204/EPTCS.368.2

to implement, thereby giving software developers and engineers the power to embed their own interpretation of the law into the technical artefacts that they create [26]. More precisely, the process of translating parties' intentions, promises, actions, powers and prohibitions into computer code, although public and unambiguous for the machine, is problematic and does not solve the problem but moves it into another dimension ([20]). Secondly, the code-driven law is based on the the automation of compliance with preset rules: if certain conditions are met the code will self-execute whatever it was programmed to do, not leaving room for disagreement about the right way to interpret the norms. Even if the need for judicial arbitration cannot be eliminated (*e.g.* one always has the right to appeal to the court if the code adopted an incorrect tax rate), the code-driven normativity transforms the nature of legal protection potentially reducing the capability of individual human beings to invoke legal remedies [5].

As an example, the Ethereum's code-is-law dogma declined with the famous TheDAO attack [34]. Indeed, from the code-is-law perspective, a problem in the source code leading to unexpected behaviour of the smart contract, is a feature of the code and not an error. But the first hard fork of the Ethereum blockchain showed that this principle is not satisfactory in practice: when large volumes of money are at stake, no one is really willing to consider a security error in a program as part of the contract they have signed. Moreover, a less naive look nowadays leads us to state that blockchain does not hardwire trust into algorithms, but rather reassigns trust to a whole series of actors (miners, programmers, companies and foundations) who implement, manage and enable the functioning of this technological platform.

2 Form Legal Contracts to Legal Calculi

Despite the difficulties highlighted above, a sensible process of digitisation of legal texts has clear advantages. In this article we discuss a specific line of research, conducted in collaboration with Cosimo Laneve and Giovanni Sartor, focusing on a specific subset of legal documents, that is the legal contracts ([19, 21, 18] and other submitted articles). Legal contracts are defined as "those agreements that are intended to give rise to a binding legal relationship or to have some other legal effect" [35]. The principle of *freedom of form* in contracts, which is shared by modern legal systems, says that parties are free to express their agreement using the language and medium they prefer, including a programming language. Therefore, by this principle, software-based contracts may count as legal contracts. However, 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.). Moreover, the contract's institutional effects are guaranteed by the possibility of activating judicial enforcements. That is, each party may start a lawsuit if she believes that the other party has failed to comply with the contract. Therefore, the assimilation of software-based contracts to legally binding contracts, or rather the double nature of digital contracts as computational mechanisms and as legal contracts, raises both legal and technological issues.

First of all, in [19] we observe that different kinds of software-based solutions can be valuable in the different phases of the lifecycle of a legal contract, which goes through negotiation, contract storage/notarizing, performance, enforcement and monitoring, possible modification and dispute resolution. Accordingly, several projects are being developed for defining code-driven legal contracts, *e.g.* [30, 17, 37, 28, 24, 11, 31]. We focus here in the problem of defining suitable programming languages to write legal contracts, since finding the suitable abstraction level for legal languages is still an open issue. Indeed, such a language should be easy-to-use and to understand for legal practitioners, but at the same time, the language should be fairly expressive, have a running environment with a precise

semantics, and possibly supply sensible analyzers.

The solution we discuss in this article is the *Stipula* programming language, whose design is based on the following main remarks:

- it is an *intermediate domain-specific language*: a core calculus more concrete than a user-friendly contract specification language, and more abstract than a full-fledged programming language. This is in line with the research approach of desugaring the high level programming language into a core *Legal Calculus* [12, 25], pivoted on few selected, concise and intelligible primitives, together with a precise formalisation. This is the case of the Catala [31] language for modelling statutes and regulations clauses, the Orlando [11] language for modelling conveyances in property law, and the Silica language [16] language for generic smart contracts;
- the basic primitives of *Stipula* has been designed to easily map the *building blocks of legal contracts* into template programs and design patterns. Therefore, the direct formalisation of normative elements (*i.e.*, permissions, prohibitions, obligations, judicial enforcement and openness to the external context) as programming patterns, increases the transparency and the understanding of the link between executable instructions and institutional-normative effects;
- a legal contract is interpreted as an *interaction protocol*, that dynamically regulates permissions, prohibitions and obligations between parties, which behave concurrently as time flows. Accordingly, the definition of *Stipula* is influenced by the theory of concurrent systems, both in the definition of the operational semantics (with a precise control of nondeterminism) and in the definition of a bisimulation-based observational equivalence, that equates contracts that are syntactically different but are legally equivalent since they exhibit the same observable normative elements;
- the language definition is *implementation-agnostic*, and can be either implemented as a centralised platform or it can be run on top of a distributed system, such as a blockchain. Implementing *Stipula* in terms of smart contracts (*e.g.*, compiling in Solidity), would bring in the advantages of a public and decentralised blockchain platform. However, digital legal contracts are more general and encompass smart contracts: they can provide benefits in terms of automatic execution and enforcement of contractual conditions, traceability, and outcome certainty even without using a blockchain. In particular, running a legal contract over a secured centralised system allows for more efficiency, energy save, additional privacy. Moreover, a controlled level of intermediation can better monitor the contract enforcement, dealing with disputes between contract's parties and carrying out judicial enforcements. A prototype centralised implementation of *Stipula* as a Java application is available in [22].

We think that, even if only a concrete implementation can properly address specific issues, studying the theory of a domain-specific legal calculus is a first interesting step, that sheds some light on the digitalisation of legal texts.

3 Stipula and the code-driven normativity

A preliminary interdisciplinary research recognised that most real legal contracts are written by combining the following basic elements:

1. the *meeting of the minds*, that involves the contract's subscribers to accept the terms of the contract, and identifies the moment when legal effects are triggered;
2. a number of *permissions*, *prohibitions* and *obligation* clauses that may dynamically change, *e.g.*, the permission to use a good until a deadline;

legal contracts	*Stipula* **contracts**
meeting of the minds	agreement primitive
permissions, prohibitions	state-aware programming
obligations	event primitive
currency and tokens	asset-aware programming
openness to the environment	intermediary pattern
judicial enforcement and exceptional behaviours	authority pattern

Figure 1: Correspondence between legal elements and *Stipula* features

3. transfer of *currency* or other *assets*, *e.g.*, the property of a physical or digital good, to be used for payments, escrows and securities;
4. the *openness* to external conditions or data, *e.g.*, a triggering condition depending on the value of a stock at a given date;
5. the possibility of activating *judicial enforcements* triggered by a dispute resolution mechanism or by a third party monitoring conditions that can be hardly digitalised, as the diligent care or the good faith.

Accordingly, the basic primitives of *Stipula* has been designed to easily map these building blocks of legal contracts into template programs and design patterns, as summarised in Figure 1. More precisely, the `agreement` construct directly encodes the meeting of the minds. 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 and issue a corresponding penalty if the obligation has not been met. Moreover, to model the dynamic change of the set of normative elements according to the actions that have been done (or not), *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 [4]). This technique allows one to enforce the intended behaviour by prohibiting, for instance, the invocation of a function before another specific function is called.

In order to promote an *asset-aware programming* ([33, 23, 13]), assets are a specific value type, and asset manipulation is syntactically distinguished from standard operations, to stress the fact that assets cannot be destroyed nor forged but only transferred. Contract clauses depending on external data are implemented by means of a party that takes the role of intermediary and assumes the legal responsibility of timely retrieving data from the external source agreed in the terms of the contract (see the bet contract below). The contract's intermediary need not to be a third party authority, but one of the party can assume also the role of intermediary, provided that all the others agree. This is different from relying on Oracles web services, to whom legal responsibilities can hardly be attributed. Finally, dispute resolutions, judicial enforcement of legal clauses and exceptional behaviours due, *e.g.*, to force majeure, are implemented by including in the contract a party that takes the legal responsibility of interfacing with a court or an Online Dispute Resolutions platform[2].

[2]as The European ODR platform at `https://ec.europa.eu/consumers/odr`.

```
1  stipula Subscription {
2      assets wallet
3      fields cost, deposit
4
5      agreement (Editor,Buyer) {
6          Editor, Buyer: cost, deposit
7      } ⇒ @Inactive
8
9      @Inactive Buyer : subscribe [h]
10         (h == deposit) {
11             h ⊸ wallet
12             now + 1 month ≫ @To_Pay { wallet ⊸ Editor } ⇒ @End
13     } ⇒ @To_Pay
14
15     @To_Pay Buyer : annualFee [h]
16         (h == cost) {
17             h ⊸ Editor
18             now + 1 year  ≫ @Payed {} @To_Pay
19             now + 1 year + 1 month ≫ @To_Pay { wallet ⊸ Editor } ⇒ @End
20     } ⇒ @Payed
21
22     @Payed Buyer : terminate {
23            wallet ⊸ Buyer
24     }⇒ @End
25 }
```

Listing 1: The subscription contract

We illustrate the expressivity of *Stipula* by showing the contracts for a set of archetypal acts (taken form [19]). They are simple but they represent the distinctive elements that can be found in most contracts.

3.1 Subscription contract: obligation of periodic payment

We define a simple contract representing the annual subscription to a magazine or a service. Upon subscription the buyer must pay a deposit, then she must pay the annual fee. If she has not paid within one month, the deposit is transferred to the editor. At the end of the year an event changes the status of the contract so to enable the payment of the annual fee with a maximum delay of one month. If the buyer is up to date with the payments, she can terminate the subscription and get back the deposit.

The code in Listing 1 shows that a contract is similar to a class in an OOL, containing a set of fields, a constructor and a number of functions. Contract's fields are distinguished into standard fields (`cost` and `deposit` store numbers corresponding to the fees and the deposit) and assets. The contract's asset field `wallet` is initially empty and will hold the buyer's money in escrow. The agreement (lines 5-7) is a sort of constructor for the contract: it is intended as a multiparty synchronization between the parties, *i.e.* `Editor` and `Buyer`, who have to agree about the initial values of `cost` and `deposit`. After the agreement has been reached, the contract enters into the initial state `@Inactive`.

The possible states of the contract are `@Inactive, @To_Pay, @Payed`, and the contract's functions `subscribe`, `annualFee` and `terminate` are defined so that only the buyer (who subscribed the agree-

ment) can call them, and `subscribe` can be called only once at the beginning. The parameter `h` is an amount of assets, and a pre-condition checks that it corresponds to the expected amount. The operation `h —∘ wallet` transfers the assets `h` into the constract's `wallet`, while `h —∘ Editor` moves them to the editor.

Lines 12,18 and 19 issue the events corresponding to the annual payment obligation. Line 12 and 19, schedule an event that, after one month from now, resp. form the end of the paid year, check whether the (first) annual fee has not been paid (*i.e.* the state is still `To_Pay`), and in that case transfer the deposit to the editor and terminate the contract. Line 18 issues an event that in a year's time will allow the new payment by moving the contract's state from `@Payed` back to `@To_Pay`. Finally, the buyer is allowed to terminate the subscription only if all payments are regular; accordingly, the function `terminate` can be invoked only in state `@Payed` and the deposit is refunded to the buyer.

3.2 The Digital Licensee contract: usage and purchase, dispute resolution

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 [27]). 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, the licence contract holds two different assets: an indivisible non fungible `token` providing an handle to the digital service, and a `wallet` that is a fungible asset corresponding to the amount of currency kept in custody inside the contract.

A further important feature of the contract is Article 3 that defines specific constraints about the off-line behaviour of `Licensor` and `Licensee`, that is their behaviour in the physical world. This exemplifies the very general situations where contract's violations cannot be fully monitored by the (on-line) software, *i.e.* by the platform that runs the software (either a blockchain or a centralized application), such as the publication of a post in a social network, or the leakage of a secret password, or any non-automatically verifiable contextual circumstance. The intrinsic *open nature* of legal contracts is exactly this mix of external behaviour and automatic enforcement of contract clauses by means of software. The code in Listing 2 illustrates the *Stipula* programming pattern that relies on a trusted third party, the `Authority` included in the agreement, to supervise the disputes occurring from the off-line monitoring and to provide a trusted on-line dispute resolution mechanism.

The `agreement` of Listing 2 involves three parties: `Licensor` and `Licensee`, which agree to the parameters of the contract, according to Article 1. (line 6), and `Authority`, which does not need to agree

```
stipula Licence {
  assets token, wallet
  fields cost, t_start, t_limit

  agreement (Licensor,Licensee,Authority){
    Licensor, Licensee : cost, t_start, t_limit
   } ⇒ @Inactive

  @Inactive Licensor : offerLicence [t] {
    t ⊸ token
    now + t_start ≫ @Proposal { token ⊸ Licensor } ⇒ @End
  } ⇒ @Proposal

  @Proposal Licensee : activateLicence [h]
    (h == cost){
       h ⊸ wallet
       wallet*0,1 ⊸ wallet, Authority
       uses(token,Licensee) → Licensee
       now + t_limit ≫ @Trial {
                        wallet ⊸ Licensee
                        token ⊸ Licensor
                      } ⇒ @End
  } ⇒ @Trial

  @Trial Licensee : buy {
    wallet ⊸ Licensor
    token ⊸ Licensee
  } ⇒ @End

  @Trial Authority : compensateLicensor {
    wallet ⊸ Licensor
    token ⊸ Licensor
  } ⇒@End

  @Trial Authority : compensateLicensee {
    wallet ⊸ Licensee
    token ⊸ Licensor;
  } ⇒ @End
}
```

Listing 2: The contract for a digital licence

upon the contracts' parameters, but it is important that it is involved in the agreement synchronization. By calling the function `offerLicence`, the `Licensor` transfers to the contract the token corresponding to the full access to the digital service. This transfer is necessary to implement the fact that, after the activation of the the licence (within the agreed time limit `t_start`, see the event in line 11), the licensor has the legal prohibition of preventing the access to the digital service. The `Licensee` can then call `activateLicence` together with an amount of assets equal to the fixed `cost` of the license, that is then stored in the `wallet` (line 16). In line 17 a fraction of asset is moved towards the authority as a fee, while in line 18 a personal usage code associated to the token is communicated to the `Licensee`.

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 `buy` which grants him the full token, or (*ii*) the time limit for the free evaluation period is reached, thus the event scheduled in line 19 refunds the licensee (but for the fees) 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`. Notice that it is important that the code guarantees that in all possible cases the assets, both the token and the wallet, are not indefinitely locked in the contract.

3.3 Bike Rental contract: access to a good without transfer of ownership

We now consider a realistic contract for a city bike rental service[3], which exemplifies a general rental contract (this is taken from [18]). 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 payment triggers the legal bond, that is the borrower has the permission to use the bike 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 bike, 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 a 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.

This example puts forward the fact that, when a legal contract refers to a *physical* good, the digital contract needs a digital handle (an avatar) for that good. Moreover, the rent legal contract grants just the *usage* of a good without the transfer of ownership. 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. Moreover, 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, say a string, providing access to the object associated to the token (resp. a usage-code only valid (once) for the party `A`). Therefore, a physical object can be handled as a digital one using the same pattern used in the digital license contract above.

Figure 2 uses connected boxes to highlight the correspondence between the normative elements of a standard bike rental contract and the corresponding editing in *Stipula*. The parties agrees on the time limit for the rental and the cost of the service, which corresponds to the double of the fee in order to

[3]For instance see the contract in `http://www.thebicyclecellar.com/wp-content/uploads/2013/10/Bike-Rental-Contract-BW.pdf`

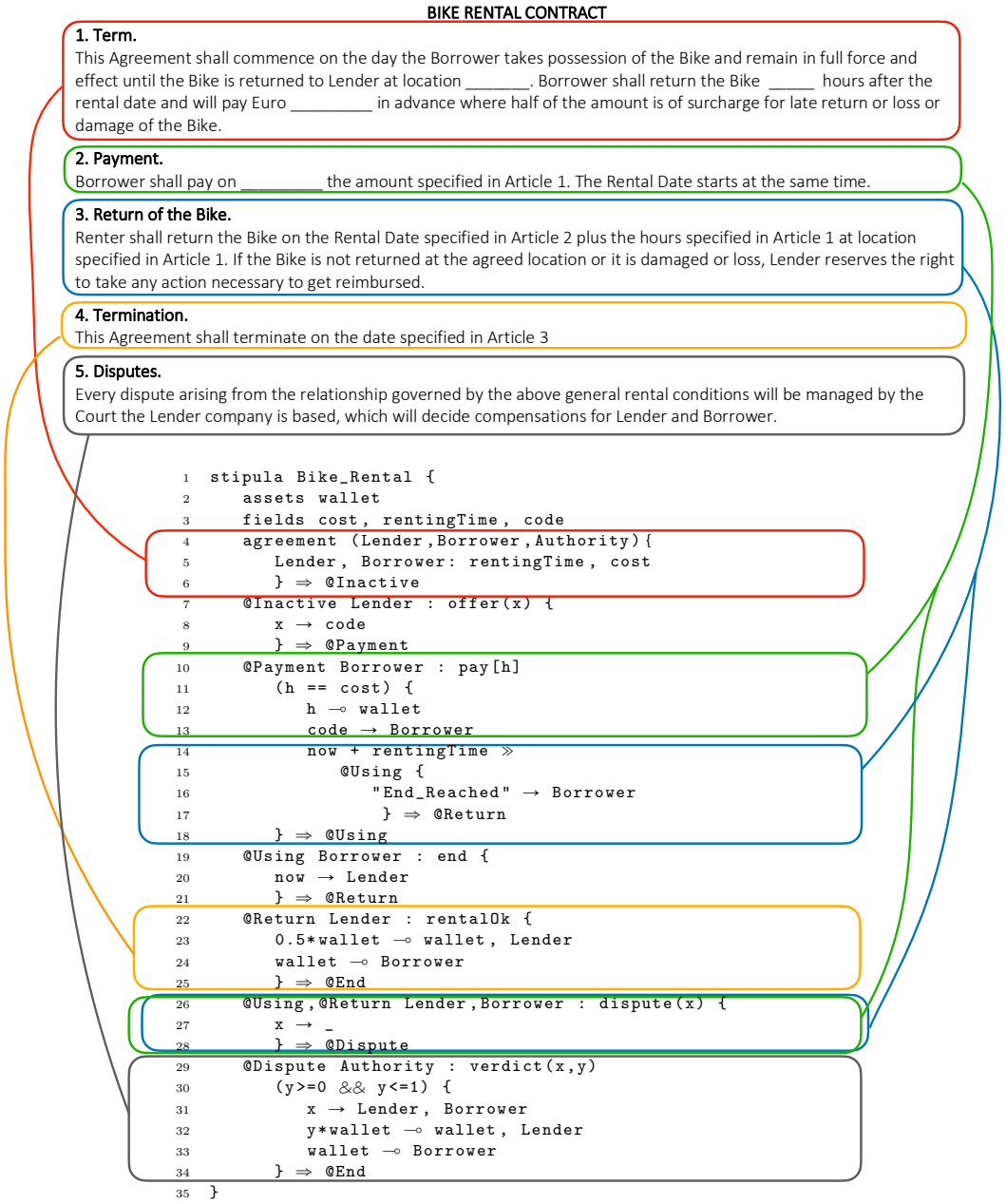

Figure 2: A standard Bike Rental contract and its modelling in *Stipula*

safeguard lender from damages, late returns or loss of the bike. For simplicity, in this code the `Lender` sends to the contract a simple usage code for the bike by calling the function `offer`. Then the `Borrower` pays the expected amount and receives the bike's usage code. Lines 14-18 issue an event corresponding to the obligation of returning the bike within the agreed time limit. Indeed, at time `now + rentingTime` the event is automatically triggered by the systems, and if the bike has not been already returned (*i.e.*, the state of the contract is still `@Using`), a message of returning the bike is sent to the borrower and the contract moves to the state `@Return`. The termination of the rental requires the `Borrower` to call the function `end`, after which the `Lender` has to confirm the absence of damages by invoking `rentalOK`. Only this sequence of actions allows the lender to be payed and the borrower to get back the money deposited as security. For the sake of simplicity this contract does not impose a penalty to the borrower for late return, but it is not difficult to modify the code with an additional state `@LateReturn` so to let the `Lender` keep the entire contract's wallet when `rentalOK` is called in the state `@LateReturn`.

The function `dispute` may be invoked either by the `Lender` or by the `Borrower`, either in state `@Using` or `@Return`, and carries the reasons for kicking the dispute off (`x` is intended to be a string). Once the reasons are communicated to every party (we use the abbreviation "`_`" instead of writing three times the sending operation) the contract transits into a state `@Dispute` where the `Authority` will analyze the issue and emit a verdict. This is performed by permitting in the state `@Dispute` only the invocation of the `verdict` function, that has two arguments: a string of motivations `x`, and a coefficient `y` that denotes the part of the wallet that will be delivered to `Lender` as reimbursement; the `Borrower` will get the remaining part. It is worth to spot this point: the statement `y*wallet ⊸ wallet, Lender` *takes* the `y` part of `wallet` (`y` is in [0..1]) and sends it to `Lender`; *at the same time* the `wallet` is reduced correspondingly. The remaining part is sent to `Borrower` with the statement `wallet ⊸ Borrower` (which is actually a shortening for `1*wallet ⊸ wallet, Borrower`) and the `wallet` is emptied.

3.4 Bet contract: dependency on external data

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, usually an accredited web page or a specific online service – stored in the field `data_source` – that will publish the final value of the aleatory event. This value will be communicated by the party that assumes the role of `DataProvider`, taking the legal responsibility of supplying the correct data from the agreed source. In particular, it is not necessary that the actual data is directly provided by a trusted institution or an accredited online service, such as an Oracle service, who could hardly take an active legal responsibility in a bet contract. But two betters, say Alice and Bob, can agree to rely on a third party Carl for supplying data, or they can simply agree on the fact that Alice takes both the role of `Better1` and `DataProvider`.

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 legal bond must be established before the occurrence of the aleatory event, the aleatory event might not happen, *e.g.* the football match is cancelled, or the data provider might fail to provide the required value, *e.g.* the online service is down.

The *Stipula* code in Listing 3 corresponds to the case where `Better1` and `Better2` place in `val1` and `val2` their bets, while the agreed amount of currency is stored in the contract's assets `wallet1` and

```
stipula Bet {
  assets wallet1, wallet2
  fields alea_fact, val1, val2, data_source, fee, amount, t_before, t_after

  agreement(Better1,Better2,DataProvider){
    DataProvider, Better1, Better2 : fee, data_source, t_after, alea_fact
    Better1, Better2 : amount, t_before
   } ⇒ @Init

  @Init Better1 : place_bet(x)[h]
    (h == amount){
        h ⊸ wallet1
        x → val1
        t_before ≫ @First { wallet1 ⊸ Better1 } ⇒ @Fail
    } ⇒ @First

  @First Better2: place_bet(x)[h]
    (h == amount){
        h ⊸ wallet2
        x → val2
        t_after ≫ @Run {
            wallet1 ⊸ Better1
            wallet2 ⊸ Better2 } ⇒ @Fail
  } ⇒ @Run

  @Run DataProvider : data(x,y,z)[]
    (x == data_source && y==alea_fact){
        if (z==val1 && z != val2){           // The winner is Better1
            fee ⊸ wallet2,DataProvider
            wallet2 ⊸ Better1
            wallet1 ⊸ Better1
        }
        else if (z==val2 && z != val1){      // The winner is Better2
            fee ⊸ wallet1,DataProvider
            wallet1 ⊸ Better2
            wallet2 ⊸ Better2
        }
        else {                                    //No winner
            fee*0.5 ⊸ wallet1,DataProvider
            fee*0.5 ⊸ wallet2,DataProvider
            wallet2 ⊸ Better1
            wallet1 ⊸ Better1

        }
   } ⇒ @End
}
```

Listing 3: The contract for a bet

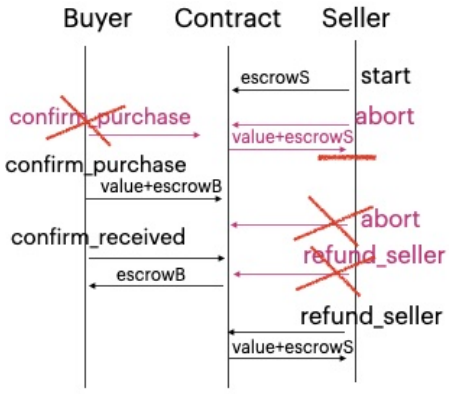

Figure 3: Safe Remote Purchase

`wallet2`[4]. Observe that both bets must be placed within an (agreed) time limit `t_before` (line 14), to ensure that the legal bond is established before the occurrence of the aleatory event. The second timeout, scheduled in line 21, 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 Authority pattern in the Digital Licence and Bike Rental examples, 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, that is entitled (and legally bound) 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`. As usual, any dispute that might render the contract voidable or invalid, *e.g.*, one better knew the result of the match in advance, or the DataProvider supplied an incorrect value, can be handled by including an `Authority` party, according to the pattern illustrated above.

3.5 Safe Remote Purchase contract: a distributed interaction protocol

In a remote purchase[5], the buyer would like to receive an item from the seller and the seller would like to get money (or an equivalent) in return. The problematic part is the shipment: there is no way to determine for sure that the item arrived at the buyer. The typical solution is to define the interaction protocol so that both parties have *an incentive to resolve the situation* or otherwise their money is locked forever.

The idea is that both parties have to put an amount into the contract as escrow. As soon as this happened, the money will stay locked inside the contract until the buyer confirms that he received the item. The intended protocol is the following sequence of actions (depicted in black in Figure 3): (1) the seller starts the transaction sending its escrow to the contract, (2) the buyer confirms the purchase by sending to the contract the money corresponding to the price of the good plus the escrow, (3) upon reception of the good, the buyer has to confirm the reception to the contract in order to get back the escrow, (4) finally the seller can receive from the contract the price of the good and the money he deposited in escrow.

Besides the intended sequence of actions, many situations can happen in a remote purchase:

[4]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.

[5]This example is taken from `https://docs.soliditylang.org/en/develop/solidity-by-example.html#safe-remote-purchase`, but most of e-commerce platforms has similar use cases.

- if the seller starts but the buyer does not confirm the purchase, seller can take back its escrow with a call to `abort`,
- if the seller does not start, the buyer does not send the escrow, so no money is locked,
- if the buyer confirms the purchase, the seller cannot take back its escrow (and the payment) until he sends the good (and it is received),
- if the buyer has confirmed the purchase but he does not confirm the reception, either because the good is not arrived or because the Buyer is cheating, nobody can take back escrow. Therefore we add to the contract a time limit, after which it is up to an Authority party to decide off-line who is to blame and then implement the decision by calling `refund`. In other terms, the mutual escrow is used as an incentive for the parties to collaborate, but *progress is not ensured* thus the contract requires *timeouts*.

```
stipula Purchase {
  asset wallet
  field value, escrow

  agreement(Buyer,Seller,Authority){
     Seller, Buyer, Authority, : value, escrow, time_limit
  } => @Init

   @Init Seller : start [h] (h == escrow) { h —∘ wallet } =>@Created

   @Created  Seller: abort { wallet —∘ Seller } =>@Inactive

   @Created Buyer : confirmPurchase [h] (h == value + escrow) {
       h —∘ wallet
       now + time_limit >> @Locked {
                           "nothing received (maybe!)" → Buyer
                           "nothing received (maybe!)" → Seller
                         }=> @Dispute
   } =>@Locked

   @Locked Buyer : confirmReceived { escrow —∘ wallet, Buyer } =>@Release

   @Release Seller : refundSeller  {
      wallet —∘ Seller // equal to (value+escrow) —∘ wallet, Seller
   } =>@Inactive

   @Dispute Authority : refund(x,y) (wallet==x+y) {
       x —∘ wallet, Buyer
       y —∘ wallet, Seller
   } =>@Inactive
}
```

Listing 4: The safe remote purchase contract

We remark that the contract in Listing 4 does not solve all legal issues. For instance in a purchase the consumer has the power to withdraw from an online sale, and there are usually warranties if the good was damaged or different from the sellers' description. To deal with all these situations the contract can be enriched with a more complex Authority pattern as in the previous examples.

3.6 Mutual Dissent and contract modification

There is a last distinctive element in legal contracts that deserves a comment: the management of exceptional behaviours, that is all those behaviours that cannot be anticipated due to the occurrence of unforeseeable and extraordinary events. For instance, legal contracts can always be dissolved if the parties agree.

We can model the *mutual dissent* by including a specific function in the contract, which can be activated with the agreement of both parties, that causes the contract to go into a stand-by state, which blocks the execution of all functions not yet performed. This prevents the contract from continuing when both parties no longer want it to. More precisely, the following code shows the Mutual Dissent pattern for a generic contract `C` where parties `P1` and `P2` may express mutual dissent:

```
stipula Rescindable_C {
  assets a1, ..., an
  fields ...
  agreement(P1,P2,....,Authority) ....

  // add a copy of this function for any state X of the contract C
  @X P1: dissent { now + 1 day >> @OneDissented { }=>@X } =>@OneDissented
  @OneDissented P2: dissent {} => @Rescinded

  @X P2: dissent { now + 1 day >> @TwoDissented { }=>@X } =>@TwoDissented
  @TwoDissented P1: dissent{} => @Rescinded

  @Rescinded  Authority : terminate {
    a1  --o Authority
    ...
    an  --o Authority
  } =>@End
```

Listing 5: The mutual dissent pattern

To prevent assets being locked indefinitely in the contract, the function `terminate` sends all the assets to the authority. More complex assets reallocation to the parties can also be implemented, provided that they mutually agree on the reallocation.

Finally, parties have the power of dynamically change the terms of the contract if they agree to it. Contract modification can be modelled by the termination of the running contract `C` (with the mutual dissent pattern), and the activation of a new contract `C'`, to which the assets remaining in `C` are transferred. The basic *Stipula* language does not allow to pass contracts' names as arguments, nor allows to invoke external contracts' activations or inter-contracts functions invocations (differently from, *e.g.* Solidity smart contracts). Therefore, the bridge between the termination of `C` and the activation of `C'` with remaining assets must be performed off-line by the Authority.

4 Legal contracts and the power of formal methods

As already discussed, the advantage of using a Legal Calculus to draft legal contracts is that a concise and well-defined language reduces the ambiguities (and therefore the grey areas) characteristics of traditional legal drafting. In particular, we remark that there are three levels of formalisation, corresponding to three different aspects of a language: the syntax, the semantics, and the analysis and verification tools.

Almost all projects for Code-Driven Law put forward a legal language based on a well-defined syntax. This is indeed the base to mechanise the writing of dedicated software that encodes legal content –not just legal contracts but any kind of legal data. These projects often come with templates for standard legal documents, that can be customised by setting template's parameters with appropriate values. There are legal language definitions based on context free grammars (as Lexon [30]), or domain specific markup languages and ontologies to wrap logic and other contextual informations around traditional legal prose (as OpenLaw [37], Accord [17], SLCML [24]), or legal specification languages based on visual programming interfaces (as in [32, 36]).

More complex is instead the formalisation of the semantics of a programming language, which is however essential to have the full understanding of the software, and the certainty of the dynamic contract's behaviour. Legal calculi, such as Catala [31], Orlando [11] and Stipula [19], have the suitable size to fully handle their formal semantics. We discuss below the case of *Stipula*, which acknowledges the concurrent nature of legal contracts as interaction protocols, and resorts to concurrency theory to define the semantics of contracts and to precisely control complex aspects like nondeterminism.

Finally, the most powerful benefit of formal methods is the deployment of automated tools to (statically) analyse the legal software in order to check safety properties, verify the absence of specific errors and possibly the reachability of convenient states. This level of formalisation is still at an initial stage in the literature, since it requires a robust definition (and implementation) of the language semantics, and because the identification of the desirable properties of legal software is still an open question.

4.1 Defining a formal semantics

The full definition of *Stipula*'s operational semantics (currently submitted to publication, but a preliminary version is available in [19]) is given in terms of a labelled transition system $\mathbb{C},\mathbb{t} \xrightarrow{\mu} \mathbb{C}',\mathbb{t}'$ that highlights the open nature of the contracts' behaviour, whose execution requires the interaction with the external context. The *runtime configuration* $\mathbb{C},\mathbb{t}$ is a pair where $\mathbb{C}$ is the runtime status of the running contract (storing its current state and the pending events), and $\mathbb{t}$ is the time value of the system's global clock. The actions that can be performed by a contract time $\mathbb{t}$ are the following

$$\mu \quad ::= \quad \tau \quad | \quad (\overline{A}, \overline{A_i} : \overline{v_i}^{i\in 1,..,n}) \quad | \quad A : \mathtt{f}(\overline{u})[\overline{v}] \quad | \quad v \to A \quad | \quad a \multimap A \,.$$

where the label $(\overline{A}, \overline{A_i} : \overline{v_i}^{i\in 1,..,n})$ observes the agreement that the parties are going to sign, that is who is taking the legal responsibility for which contract's role, and what are the terms of the contract, *i.e.*, the agreed initial values of the contract's fields. The label $A : \mathtt{f}(\overline{u})[\overline{v}]$ observes the possibility (at time $\mathbb{t}$) for the party A to call the function f. The labels $v \to A$ and $a \multimap A$ observe that (at time $\mathbb{t}$) the party A can receive a value and an asset, respectively. Contract's field updates, internal asset moves and event scheduling, as well as time progress, are not observed (label τ).
The behaviour of a *Stipula* legal contract can be described as the following procedure:

1. the first action is always an agreement, which moves the contract to an idle state;
2. in an idle state, if there is a ready event with a matching state, then its handler is completely executed, moving again to a (possibly different) idle state;
3. in an idle state, if there is no event to be triggered, either advance the system's clock 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, the semantics has three sources of nondeterminism: (*i*) the order of the execution of ready events' 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). We remark that a nondeterministic behaviour is not necessarily an error: even the execution of legal contracts written in natural language might lead to nondeterministic executions, in particular when the contract leaves room for a participant not to timely perform an action that was expected to do. Depending on how the contract is written, this may be admissible or may cause a legally uncertain situation that can only be solved by a court. Therefore, the precise formalisation of a contract's behaviour in terms of an operational semantics has the advantage of explicitly knowing what are the sources of nondeterminism, and allows to precisely control it.

4.2 Observing legal contracts through Normative Equivalence

One of the difficulties of writing contracts in natural language is the fact that the same legal bindings can be expressed with many similar texts. Then it is often difficult to properly check when two contracts that are syntactically different are instead legally equivalent, meaning that the parties using them cannot distinguish one from the other. By relying on the operational semantics, that formally defines the observable actions of a contract behaviour, we can define a *bisimulation-based observation equivalence*, where two contracts are deemed to be legally equivalent if they involve the same parties observing the same interactions during the contracts' lifetime.

More precisely, the so-called *Normative Equivalence* (see [19]) equates two contracts if

- they provide the same agreement, that is the same parties take the same legal responsibility and agree on the same terms of the contract (expressed by the action $(\overline{A}, \overline{A_i} : \overline{v_i}^{i\in 1..n})$);
- every party is subject to the same dynamic set of permissions, prohibitions and obligations;
- every party receives from the contract the same assets and values (actions $v \to A$ and $a \multimap A$);
- the bisimulation game abstracts away the ordering of the observations within the same time clock, and enforces a transfer property that shifts the time of observation to the next time unit.

To observe the permissions and prohibitions at time $\mathbb{t}$, we observe whether any party can invoke, resp. cannot invoke, any function (expressed by the action $A : \mathtt{f}(\overline{u})[\overline{v}]$). Obligations are captured implicitly by shifting the observation at a specific point in time, and observing –in the future– the effects of executing the event that encodes the legal commitment. In particular, the system's clock needs not to be directly observed: by checking the set of permissions and prohibitions at any time units, and since only a contract's state change can modify the set of valid permissions and prohibitions, it is sufficient to observe whether a function can be executed before of after another function or an event, disregarding its precise execution time unit.

As a consequence, the Normative Equivalence safely abstracts away the ordering of the observations within the same time unit: if a party receives two messages in different order it might be due to delays of communications, rather to sensible differences in the contracts. Nevertheless, the equivalence does not overlook essential precedence constraints, which are important in legal contracts, as the requirement that a function delivering a service can only be invoked after another specific function, say a payment. Additionally, the Normative Equivalence abstracts away from the names of the contract's assets, fields and internal states, and it is also independent from future clock values, allowing to garbage-collect events that cannot be triggered anymore because the time for their scheduling is already elapsed.

4.3 Verification of contracts' properties

By looking at legal contracts as interaction protocols and by relying on a well defined operational semantics, the rich theory of formal methods for concurrent systems can be a great source of inspiration to develop analysis and verification tools. However, first of all it is essential to conduct an interdisciplinary investigation to properly identify what are the errors and the properties that should be targeted by the techniques providing safety and liveness guarantees.

An important class of errors are those related to unsafe usage of assets, which must obey to a linear semantics (no forging, no duplication, no loss) and whose content must be meaningful. For instance, in *Stipula* the assets corresponding to currency, as the asset `wallet` in the examples above, must always contain a non negative amount of money. Accordingly, an asset transfer what would leave a contract's asset with a negative (unsafe) asset, *e.g.*, `100` $\multimap$ `wallet,A` when `wallet` holds less than 100 coins, is not executed and results in a stuck configuration. Similarly, if the contract's asset `token` already contains a non fungible token providing access to a good, say a digital service, then the operation `t` $\multimap$ `token` that would accumulate or overwrite the `token` with the asset `t` must not be executed. Moreover, assets must not be indefinitely locked into contracts: at any time it should be possible, at least for some party, to redeem the assets stored into the contract; this is often called *liquidity* property ([10]). These issues are at the core of the research about *resource-aware languages* as Obsidian [4, 15] Nomos [23, 14], Flint [33] and Move [13]; and even the questions "What is the type `Money` in a programming language? What are its suitable abstractions?" and "What is the difference between the more general type `Asset` and the type `Money`?" are still open issues.

Other kinds or errors are those related to non collaborative parties, that might prevent the progress of the contract or might move it to a problematic state. We have described *Stipula* design patterns, as the authority pattern or the mutual dissent pattern, that can be inserted in the drafting of the digital contract as a sort of escape hatch; however, a static analysis of the runtime behaviour of the contract would be very useful.

5 Conclusions

In this article we discussed the role of Legal Calculi in the process of digitisation of legal contracts. We illustrated the design choices of *Stipula*, whose primitives naturally support the encoding of contracts' normative elements (permissions, prohibitions, obligations, asset transfer, judicial enforcement and openness to the external context). We also remarked that legal contracts can be interpreted as interaction protocols between concurrent parties, leading to a fruitful connection with the rich toolset of formal methods available for concurrent systems.

Studying the theory of domain-specific legal calculi is a useful research line, that supplement the development of the Code-Driven Law trend. On the other hand, it is important to keep a lively connection between these calculi and other two fundamental abstraction levels: the effective implementation and the interdisciplinary assessment. The actual implementation of legal calculi brings in specific challenges, such as the *legally robust* management of the identities of the parties and their valid agreement to the legal bonds. Moreover, the implementation of obligations by scheduling an event that issues a corresponding penalty if the obligation has not been met, may not be always feasible, and asks for an accurate management of time, which is a well-known challenge in distributed platforms.

Finally, the dialogue with legal researchers and professionals provides valuable insights, non just on the usability of legal programming languages, but mainly on the actual meaning (in the epistemic sense) of their abstractions. This is important to unveil when partial or erroneous interpretations of the law

has been embedded in the technical artefacts, and to understand the actual extent of the legal protection provided by the software normativity. A main lesson that we learned is the intrinsic open nature of legal contracts, that is incompatible with the automatic execution of software-based rules claimed by the Code-Driven Law. Indeed, a contract produces the intended effects, declared by the parties, only if it is legally valid: the law may deny validity to certain clauses, as an excessive interests rate. 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 ([19]). Therefore, any technical solution based on a legal programming language must provide an escape mechanism (as the authority pattern in *Stipula*) that allows a flexible, and legally valid, link between what is true on-line and off-line.

References

[1] *Solidity Documentation: State Machine Common Pattern*. `https://docs.soliditylang.org/en/v0.8.0/common-patterns.html#state-machine`.

[2] (2018): *Akoma Ntoso XML for parliamentary, legislative and judiciary documents*. At `http://www.akomantoso.org/`.

[3] (2018): *Malta MDIA Act*. At `https://mdia.gov.mt/wp-content/uploads/2018/10/MDIA.pdf`.

[4] (2018): *Obsidian: A safer blockchain programming language*. Language Site at `http://obsidian-lang.com/`.

[5] (2019): *The CoHuBiCoL research project*. At `https://www.cohubicol.com/about`.

[6] (2019): *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`.

[7] (2020): *Cracking the Code: Rulemaking for humans and machines*. At `https://oecd-opsi.org/publications/cracking-the-code/`.

[8] (2021): *Wyoming Regulation Act*. At `https://www.wyoleg.gov/Legislation/2021/SF0038`.

[9] (2022): *Catala in action*. Language site at `https://catala-lang.org/`.

[10] Massimo Bartoletti & Roberto Zunino (2019): *Verifying Liquidity of Bitcoin Contracts*. In Flemming Nielson & David Sands, editors: *Principles of Security and Trust*, Springer International Publishing, pp. 222–247, doi:10.1007/978-3-030-17138-4_10.

[11] Shrutarshi Basu, Nate Foster & James Grimmelmann (2019): *Property Conveyances as a Programming Language*. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, Association for Computing Machinery, New York, NY, USA, p. 128–142, doi:10.1145/3359591.3359734.

[12] Shrutarshi Basu, Anshuman Mohan, James Grimmelmann & Nate Foster (2022): *Legal Calculi*. Technical Report, ProLaLa 2022 ProLaLa Programming Languages and the Law. At `https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi`.

[13] Sam Blackshear & et al. (2021): *Move: A Language With Programmable Resources*. `https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2020-04-09.pdf`.

[14] Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon & Yoni Zohar (2020): *Resources: A Safe Language Abstraction for Money*. *CoRR*. arXiv:2004.05106.

[15] Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers & Joshua Sunshine (2020): *Can advanced type systems be usable? An empirical study of ownership, assets, and typestate in Obsidian*. *Proc. ACM Program. Lang.* 4(OOPSLA), pp. 132:1–132:28, doi:10.1145/3428200.

[16] Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine & Jonathan Aldrich (2020): *Obsidian: Typestate and Assets for Safer Blockchain Programming*. *ACM Trans. Program. Lang. Syst.* 42(3), pp. 14:1–14:82, doi:10.1145/3417516.

[17] Open Source Contributors (2018): *The Accord Project*. `https://accordproject.org`.

[18] Silvia Crafa & Cosimo Laneve (2022): *Programming legal contracts - a beginner guide*. In: *The Logic of Software. A Tasting Menu of Formal Methods. Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 13360, Springer, doi:10.1007/978-3-031-08166-8.

[19] Silvia Crafa, Cosimo Laneve & Giovanni Sartor (2021): *Pacta sunt servanda: legal contracts in Stipula*. *CoRR*. arXiv:2110.11069.

[20] Silvia Crafa, Cosimo Laneve & Giovanni Sartor (2022): *Le forme del falso negli smart contract*. In: *Le forme del falso*, Bologna University Press, pp. 85–98, doi:10.30682/9791254770146.

[21] Silvia Crafa, Cosimo Laneve & Giovanni Sartor (2022): *Stipula: a domain specific language for legal contracts*. Technical Report, ProLaLa 2022 ProLaLa Programming Languages and the Law. At `https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi`.

[22] Silvia Crafa, Cosimo Laneve & Adele Veschetti (2022): *Stipula Prototype*. Available on github: `https://github.com/stipula-language`.

[23] A. Das, S. Balzer, J. Hoffmann, F. Pfenning & I. Santurkar (2021): *Resource-Aware Session Types for Digital Contracts*. In: *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 111–126, doi:10.1109/CSF51468.2021.00004.

[24] Vimal Dwivedi, Alex Norta, Alexander Wulf, Benjamin Leiding, Sandeep Saxena & Chibuzor Udokwu (2021): *A Formal Specification Smart-Contract Language for Legally Binding Decentralized Autonomous Organizations*. *IEEE Access* 9, pp. 76069–76082, doi:10.1109/ACCESS.2021.3081926.

[25] Vimal Dwivedi, Vishwajeet Pattanaik, Vipin Deval, Abhishek Dixit, Alex Norta & Dirk Draheim (2021): *Legally Enforceable Smart-Contract Languages: A Systematic Literature Review*. *ACM Comput. Surv.* 54(5), doi:10.1145/3453475.

[26] Primavera De Filippi & Samer Hassan (2016): *Blockchain technology as a regulatory technology: From code is law to law is code*. *First Monday* 21(12), doi:10.5210/fm.v21i12.7113.

[27] Guido Governatori, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor & Xiwei Xu (2018): *On legal contracts, imperative and declarative smart contracts, and blockchain systems*. *Artificial Intelligence and Law* 26, pp. 377–409, doi:10.1007/s10506-018-9223-3.

[28] Xiao He, Bohan Qin, Yan Zhu, Xing Chen & Yi Liu (2018): *SPESC: A Specification Language for Smart Contracts*. In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 01, pp. 132–137, doi:10.1109/COMPSAC.2018.00025.

[29] Lawrence Lessig (1999): *Code and Other Laws of Cyberspace*. Basic Books, Inc., USA.

[30] Lexon Foundation (2019): *Lexon Home Page*. `http://www.lexon.tech`.

[31] Denis Merigoux, Nicolas Chataing & Jonathan Protzenko (2021): *Catala: A Programming Language for the Law*. *Proc. ACM Program. Lang.* 5(ICFP), doi:10.1145/3473582.

[32] Christian Reitwiebner (2018): *Babbage—A Mechanical Smart Contract Language*. At `https://medium.com/@chriseth/babbage-a-mechanical-smart-contract-language-5c8329ec5a0e`.

[33] Franklin Schrans, Susan Eisenbach & Sophia Drossopoulou (2018): *Writing Safe Smart Contracts in Flint*. In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, ACM, New York, NY, USA, p. 218–219, doi:10.1145/3191697.3213790.

[34] David Siegel (2016): *Understanding the dao attack*. Available at `https://top-forex-brokers.com/2021/10/07/understanding-the-dao-attack/`.

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

Outline Edition. Sellier. Available at https://www.ccbe.eu/fileadmin/speciality_distribution/public/documents/EUROPEAN_PRIVATE_LAW/EN_EPL_20100107_Principles__definitions_and_model_rules_of_European_private_law_-_Draft_Common_Frame_of_Reference__DCFR_.pdf.

[36] Tim Weingaertner, Rahul Rao, Jasmin Ettlin, Patrick Suter & Philipp Dublanc (2018): *Smart Contracts Using Blockly: Representing a Purchase Agreement Using a Graphical Programming Language*. In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pp. 55–64, doi:10.1109/CVCBT.2018.00012.

[37] Aaron Wright, David Roon & ConsenSys AG (2019): *OpenLaw Web Site*. https://www.openlaw.io.