Stipula: a domain specific language for legal contracts

Silvia Crafa[1], Cosimo Laneve[2], and Giovanni Sartor[3]

[1] University of Padova
[2] University of Bologna – Inria FOCUS
[3] University of Bologna – European University Institute

Abstract. We illustrate *Stipula*, a novel domain specific language that can assist lawyers in programming legal contracts through specific software patterns. The language is based on a small set of programming abstractions that have a precise correspondence with the distinctive elements of legal contracts. We overview the language by using a simple rental contract, we discuss a number of formal methods developed for *Stipula*, and we hint at its implementation, that can take advantage of (but is not limited to) a blockchain platform.

1 Introduction

Law is one of the domains that are currently most influenced by the so-called digital revolution. The main difficulty in computationally dealing with the laws is represented by the complexity of the legal documents, whose understanding often requires human judgement in order to interpret the natural language, which is at the same time very expressive and quite ambiguous.

Our research focuses on a specific subset of legal documents, the *legal contracts*, which define "those agreements that are intended to give rise to a binding legal relationship or to have some other legal effect" [4]. According to the principle of *freedom of form*, which is shared by the contractual law of modern legal systems, the parties of a legal contract are free to express their agreement using the language and medium they prefer, including a programming language. When a programming language is selected, it is necessary that its abstraction level is high enough so that writing and inspecting a software contract do not require proficiency in computer science. In fact, a genuine agreement over the content of the contract arises only if the parties (ordinary citizens, possibly supported by legal experts, *e.g.* notaries or solicitors) are fully aware of the computational effects of their code. This is mandatory in legal contracts, in order to reduce or eliminate applications to courts for either misinterpretations or misunderstandings.

We then put forward a new domain-specific language, called *Stipula*, that uses few selected, concise and hopefully intelligible (to lawyers) primitives that have a precise correspondence with the distinctive elements of legal contracts. *Stipula* is based on the observation that contractual agreements basically are *protocols* that regulate *interactions* between parties in terms of permissions, obligations, prohibitions, escrows and securities. Accordingly, the formal definition of *Stipula* is influenced by the rich *theory of concurrency* in computer science, which provides a rich toolset of formal techniques that can be adapted to automatically verify the properties and the correctness of software contracts.

In this extended abstract we overview *Stipula* by discussing a paradigmatic example – the `Bike_Rental` contract – and we outline the corresponding toolset that has been devised for it and that is either already implemented or is going to be implemented.

2 An overview of *Stipula*

Stipula is pivoted on few abstractions that capture the distinctive elements of legal contracts, namely permissions, prohibitions, obligations, fungible and non fungible assets exchanges, and aleatory or real-world data retrieval. These elements are combined into common legal patterns, that either establish new obligations, rights, powers and liabilities between the parties, or transfer rights (such as rights to property) from one party to another, often subject to specific conditions

```
 1 stipula Bike_Rental {
 2     assets wallet
 3     fields cost, rentTime, code
 4
 5     agreement (Lender, Borrower, Authority)(rentTime,cost) {
 6          Lender, Borrower : rentTime, cost
 7     } ⇒ @Inactive
 8
 9     @Inactive Lender : offer (z){ z → code } ⇒ @Payment
10
11     @Payment Borrower : pay [y]
12          (y == cost) {
13               y ⊸ wallet
14               code → Borrower
15               now + rentTime ≫ @Using {
16                                  "End_Reached" → Borrower
17                                  wallet ⊸ Lender
18                                } ⇒ @End
19     } ⇒ @Using
20
21     @Using Borrower : end { wallet ⊸ Lender } ⇒ @End // the bike is returned
22
23     @Using Authority : compensation(v)
24          (v <= 1) {
25              v*wallet ⊸ wallet, Lender // drain the amount v from wallet, send it to Lender
26              wallet ⊸ Borrower         // drain the rest and sent it to Borrower
27     } ⇒ @End                          // contract's wallet is empty
28 }
```

Fig. 1. The bike rental contract

and by taking advantage of escrows and securities. The main features of *Stipula* are illustrated in Figure 1 by means of a simple contract for renting bikes.

A *Stipula* contract is an object with assets, fields and functions. Assets and fields are separated because the formers are *linear* entities and are handled by ad-hoc operations. A basic function in *Stipula* is the `agreement` that acts as a constructor: it defines the parties that get involved in the contract and the fields' values on which (a subset of) parties must reach a consensus. In Figure 1, the agreement at lines 5-7 expresses that there are three parties – `Lender`, `Borrower` and `Authority` – and `Lender` and `Borrower` agree on the `rentTime` and on the rental `cost`. The `Authority` is charged to monitor contextual/external constraints, such as obligations of diligent storage and care, and of litigations and dispute resolution (see lines 23-27 as a simple example). It does not contribute to setting `rentTime` and `cost`. Technically, the agreement is a joint synchronization [3].

Once the parties agree, the contract starts and it goes into a state `@Inactive` (line 7), expressing that no rent occurs until a payment is received. States are pervasive in *Stipula*, which commits to a state-aware programming style, an approach that is widely used to specify interaction protocols [1]. This is supported by the fact that normative elements of legal contracts are expressed by a strictly regimented behaviour: 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 that have been done (or not). Accordingly, *Stipula*'s function definitions specify state pre and post-conditions, and which party that is authorized to call that function. As an example, the function `offer` at line 9 can be invoked only by the `Lender` when the contract is in the state `@Inactive`. In other terms, state pre-conditions in `Bike_Rental` are programmed so that the agreement fragment is giving *permission* to the `Lender` to invoke `offer`. And, since no further function is defined at `@Inactive`, then the contract is *prohibiting* other parties to do any action at this stage. The same reasoning in terms of permissions and prohibitions, encoded by state conditions, holds for all the stages of the rental protocol, whose life cycle goes through the states `@Inactive, @Payment, @Using, @End`.

The function `offer` stores the temporary access code of a bike in the contract's field `code` (this disallows the lender to withdraw from the rental; the code is disclosed to the `Borrower` after the payment, see line 14 in the function `pay`). The corresponding operation – `z → code` at line 9

– is a standard update, to be distinguished from *assets updates* that are noted $\multimap$. For example, the payment of the rental performed by the function `pay` at line 11-19 moves the currency `y`, sent by `Borrower`, to the asset `wallet` – the operation `y` $\multimap$ `wallet` at line 13 –, which is the unique possible operation for assets parameters. More explicitly, *Stipula* adopts a different syntax to *send* and update *values* – the curved brackets in `offer(z)` and `z` $\rightarrow$ `code` – and to *transfer* and move *linear assets* – the square brackets in `pay[y]` and `y` $\multimap$ `wallet`. This design choice 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. Notice that, in the example, borrower's money is kept in the contract as an escrow until the end of the rental (either line 17 or 21). This guarantees the third-party enforcement of the compensations decided by the `Authority` in case of litigations (lines 23-27).

Finally, *Stipula* uses *events* to trigger *obligations* and schedule a future statement that automatically executes a corresponding penalty, if the obligation is not met. In Figure 1, the function `pay`, after sending the code to the borrower (line 14), issues an event that will be executed at the expiration of the renting time (line 15). At that time, if the contract will be in state `@Using`, *i.e.* the borrower is still using the bike, then lines 16-17 will be executed. In this simple example, the "penalty code" just sends a warning message to `Borrower` and transfers the rental payment to `Lender`. Observe that, even if this does not seem a serious penalty, once the lender has received the rental money, nothing prevents/discourages him form changing the usage code of the bike, so that the borrower cannot use it anymore.

3 A toolset of formal methods for *Stipula*

Since the definition of *Stipula* draws several concepts from concurrency theory, the corresponding toolset relies on techniques from this domain.

In [2], the syntax and the semantics of *Stipula* – as a transition system between states – are formally defined. Using a *bisimulation technique* we then develop an *observational equivalence* that bears an equational theory of software contracts. In particular, the equivalence identifies contracts differing for hidden elements, such as names of states, and singles out conditions for equating contracts that send assets in different order.

On top of observational equivalence, we define two techniques for verifying software contracts' properties in an automatic way. The first one is a type inference system. *Stipula* syntax is untyped to ease the understanding of the language and of the codes to lawyers (in particular), but we designed an algorithm for deriving types of assets, fields and functions. In particular, the algorithm recognizes whether an asset is used in a divisible or indivisible way. The second technique verifies *liquidity* of software contracts. This property is held by those contracts that do not freeze any asset forever, *i.e.* that are not redeemable by any party. Our technique works in two steps: it labels every function and event with the input-output behaviour on assets. Then (upper bound) balances of cycles and of computations to final states is computed and a warning message is emitted in correspondence of every positive difference in balances.

The definition of *Stipula* is implementation-agnostic: it can be executed either as a centralized application or it can be run on a distributed system. For simplicity, we are currently developing a centralized prototype of *Stipula* because the effort in implementing the corresponding primitives is low. Actually, blockchain systems have been advocated for digitally encoding legal contracts, bringing the advantages of a public and decentralized platform, such as a trusted execution that is trackable and irreversible. While this implementation is in our agenda, we observe that software contracts can benefit from more efficiency, energy save, and additional privacy of a centralized implementation. Moreover, a controlled level of intermediation (rather than a blockchain) can better monitor the contract enforcement, deal with disputes between contract's parties, and carry out third-party enforcements.

References

1. Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323?342, 1983.
2. Silvia Crafa, Cosimo Laneve, and Giovanni Sartor. Pacta sunt servanda: legal contracts in Stipula. Technical report, arXiv:2110.11069, 10 2021.
3. Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2000.
4. 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.