Programming Contract Amending

Cosimo Laneve[1], Alessandro Parenti[2](✉), and Giovanni Sartor[2]

[1] Department of Computer Science and Engineering, University of Bologna, Bologna, Italy

[2] Department of Legal Studies, University of Bologna, Bologna, Italy

alessandro.parenti3@unibo.it

Abstract. Legal contracts can be generally amended either because real-world events require an adaptation of the contract to new circumstances or because new agreements between the parties take place. When legal contracts are defined by a programming language, amendments likely entail runtime modifications to the contract code. In this paper, we present a law-derived framework for amending contract codes that are written in *Stipula*, a programming language for legal contracts. The full language, called *higher-order Stipula*, is applied to modelling real-world examples of contract amendments, where modifications may add new clauses or may rewrite (part of) old ones. We also discuss the prototype implementation of the language and its graphical user interface.

1 Introduction

The use of computer code to represent, monitor or execute a legal agreement between parties has been studied and employed in various forms since the 1970s [20]. The coding of contracts can bring several benefits: lower costs of digital transactions, the monitoring of business procedures, or the avoidance of litigation because of an *ex-ante* automatic assessment of compliance [21].

Stipula [6] is a domain-specific language for drafting computable legal contracts. It was designed under the guiding principle of having an abstraction level as close as possible to contractual practice, to facilitate its use by legal professionals. For this reason, it is based on a small set of primitives that reflect the basic elements of legal contracts (*permissions, obligations*, etc.).

Current *Stipula* contracts are immutable, *i.e.*, they cannot be modified at runtime once the execution has been started. However, there may be several reasons to modify a contract, ranging from the fact that that parties have changed their mind, to the occurrence of an unexpected event that affects the contractual relationship. The latter case is usually dealt with in contracts by *hardship* clauses, and represents a crucial issue, especially in long-term commercial agreements. In Sect. 3, we analyse the most common scenarios requiring amendments, and discuss the legal basis for amending contracts in legal systems.

Because of the immutability of *Stipula*, in order to model contract amendments, one would have to anticipate the potential modifications causes at the time of contract formation, and provide for them accordingly, in the contract code. Besides being hardly feasible, such a practice would significantly raise the drafting costs, thus nullifying one of the main purposes for digitalizing legal contracts. Therefore, in the present work, we discuss the addition of a new feature to *Stipula* in order to support for future amendments during contract execution. Up to our knowledge, no other programming language for legal contracts has yet addressed this issue.

Our full language, called *higher-order Stipula* features functions that may carry computer code as an input parameter. The code is run when the function is invoked, thus possibly modifying the original contract protocol. This solution allows us to manage situations where the amendment affects the whole body of the contract, so that only the new provisions are operational, as well as situations where only some parts are changed, so that the rest of the previous code remains operational. In Sect. 4, we test the new feature on a real-world example directly taken from contract practice. In Sect. 5 we discuss two methods for restricting amendments. The first method requires the parties' agreement. With the second method, parties may pre-define constraints against which future amendments can be verified at run-time. We then explore in Sect. 6 the prototype implementation of *higher-order Stipula* and its graphical user interface. We end our contribution by discussing the state of the art in Sect. 7 and presenting our conclusions in Sect. 8.

2 Background: Modelling Contracts with *Stipula*

Stipula is a domain-specific language for modelling legal contracts that has been designed to be more concrete and execution-oriented than a specification language and, at the same time, more abstract than a full-fledged programming language [6]. *Stipula* consists of a small set of primitives that reflect some key features of legal contracts:

– A contract enters into force at the moment of the 'meeting of the minds' of the parties. This is represented by the *agreement* operator through which parties are called to agree on the terms of the contract. For example, the following script

```
    agreement (Supplier, Buyer, PriceProvider) {
            Supplier, Buyer : formula
    } init ⇒ @Start
```

  defines a contracts whose parties are `Supplier`, `Buyer` and `PriceProvider`, where `Supplier` and `Buyer` *must initially agree* on the value of the field `formula`;
– legal contracts may create, extinguish or regulate the parties' normative positions such as permissions, prohibitions, obligations or powers. *Stipula* uses *states* to model and automatically enforce prohibitions and permissions. In each state only certain functions may be invoked while others are precluded. States are indicated by an "`@`" in front. In the foregoing code, the state enabling the invocation of the `update_price` function is `@Start`. *Functions*

are used to express actions of parties and may correspond to contract clauses. For example, in the state `@Start` the `PriceProvider` updates the price and sends the new value to the `Supplier` and the `Buyer`.

```
@Start PriceProvider: update_price(p)[] {
        p → price
        p → Supplier, Buyer
} ⇛ @Waiting_order
```

– Legal contracts are usually required to manage currencies or digital goods. In *Stipula* these entities are called *assets* and operations involving them (transfers, escrows, etc.) are characterised by ad hoc syntactical specifications, thus separating them from other data types. For example, an asset transfer is expressed by

```
w ⊸ wallet
```

that *empties* the asset `w` and *moves* its value into `wallet` (that is, the value of `wallet` *is augmented* by the value of `w`). In contrast to assets, fields hold standard values (e.g., price, deadline) and are not diminished by the transferred value. For example, `p` → `price` updates the value of `price` to `p`, but `p` is not emptied;
– *events* are used to check the fulfilment of obligations at a certain time and eventually to issue a penalty. The operation

```
now + time_due >> @Inactive{
    "the contract ends" → Buyer
}⇛ @End
```

triggers a transition at time `now+time_due`. The transition, which may only take place in the state `@Inactive`, informs the `Buyer` that the contract is terminated;
– in most cases, contracts' execution may depend on external events, such as updates of timetables, sporting event results, mortgage rates, etc., whose data need to be fed into the contract. This need is satisfied by involving a trusted intermediary party, who participates in the contract and is only allowed to call specific functions. The same solution may be used to solve disputes and establish non-automatically verifiable (e.g. open-ended) circumstances, such as *force majeure*, serious damage, etc.

The formal semantics of *Stipula*, as well as the definition of techniques for verifying legal contracts and the design of the prototype, are fully reported in [6]. Here we only provide a simplified description of the syntax, so as to allow the comprehension of the examples presented.

As an example of *Stipula* code, consider the following real-world scenario directly derived from a contractual dispute which arose before the Court of Arbitration of the International Chamber of Commerce[1]. The dispute concerned

[1] ICC case n. 10351/2001.

```
 1    stipula GasSupply {
 2        parties: Supplier, Buyer, PriceProvider
 3        fields: price, formula, order
 4        assets: wallet
 5
 6        agreement (Supplier, Buyer, PriceProvider) {
 7            Supplier, Buyer : formula
 8        } init ⇒ @Start
 9
10        @Start,@Waiting_order PriceProvider: update_price(p)[] {
11            p → price
12            p → Supplier, Buyer
13        } ⇒ @Waiting_order
14
15        @Waiting_order Buyer: place_order()[w] {
16            w/(price×formula) → order
17            order → Supplier
18            w ⊸ wallet
19        } ⇒ @Order
20
21        @Order Supplier: send_gas()[g] (g == order){
22            g ⊸ Buyer
23            wallet ⊸ Supplier
24        } ⇒ @Waiting_order
25
26        @Waiting_order Supplier,Buyer: formula_revision(f)[] {
27            f → formula
28        } ⇒ @Waiting_order
29    }
```

Fig. 1. The Gas Supply contract in *Stipula*

a long-term contract for the purchase of liquid natural gas. In order to set the cost of oil, parties define a price formula (that, in our case, for simplicity, is a multiplicative factor) and agree to refer to a specific pricing agency's publications to retrieve the necessary parameters (the market price, in our case). They also provide for a price revision clause regulating the procedures and conditions for updating the formula. Table 1 presents a simplified version of the agreement in its main points. The contract could be represented in *Stipula* as shown in Fig. 1.

Lines 1–4 define parties, fields, and assets of the contract called `GasSupply`. The contract includes the third party `PriceProvider` representing the reporting agency to which `Supplier` and `Buyer` refer to retrieve the necessary market data

Table 1. The Gas Supply contract in natural language

1. Agreement. *Supplier* and *Buyer* stipulate a long term agreement for the supply of liquified natural gas. Parties indicate *Price Provider* as the reference price agency and fix a *price formula*.

2. Price. Gas price is calculated by applying the *price formula* to the price provided by *Price Provider*.

3. Purchase. *Buyer* can place an order for gas by paying in advance the corresponding price, as resulting from the *price formula*. *Supplier* shall deliver the gas ordered and payed without undue delay.

4. Formula Revision. Parties can, upon agreement, decide to revise the *price formula*, in order to match market needs.

to calculate the (discounted) price. The `formula` stores the multiplicative factor to be applied to the market price; the field `order` stores the gas order made by the buyer each time.

Through the `agreement` clause (lines 6–8) parties set the field `formula` and express their binding acceptance of contract terms (*meeting of the minds*). The contract is initialised in the state `@Start` (line 8) where the only action admitted is the setting of `price` by the reporting agency `PriceProvider` – lines 10–13. [Line 10 specifies that `update_price` may be also invoked in the state `@Waiting_order`.] Then the contract transits in the state `@Waiting_order`. In this state, the Buyer purchases gas by means of the function `place_order`. This function takes `w` representing the currency sent for the purchase (the square brackets identify the parameters that are assets). The corresponding amount of gas, *i.e.* $\texttt{w/(price}\times\texttt{formula)}$, is stored into `order` (line 16) and communicated to the `Supplier` (line 17); then `w` is escrowed by the contract and stored in `wallet` (line 18). At this point, the contract transits to the `@Order` state, enabling the `Supplier` to call the `send_gas` function. Through this, the gas (`g`) is sent to the `Buyer` (line 22) and the money stored in `wallet` sent to the `Supplier` (line 25). We remark that `send_gas` can be invoked provided the gas `g` is exactly what has been ordered by the `Buyer` (that has been stored in `order`). Finally, Lines 26–28 define the price determination formula. It can be called by both parties and allow them to update `formula` with a new one.

Further transpositions of legal contracts in *Stipula* can be found in [5]. The basic definition of *Stipula* does not admit the management of *exceptional behaviours*, *i.e.* all those behaviours that cannot be anticipated due to the occurrence of unforeseeable and extraordinary events, which, in legal contracts, are usually dealt with amendments. The extension of the language with a feature for modelling amendments is discussed in the following sections.

3 Amending Contracts

The principle of *freedom of contract* allows parties to modify contracts at their will, provided that there is an agreement and that the new content is not against the law. Occasionally, one party may yield to the other the power to change some parts of the agreement unilaterally [2]. This is a common practice for consumer contracts and standard terms of service, where the right to modify is usually tied to certain requirements, such as notifying the other party of the change and giving them the possibility to withdraw. In specific cases, the right to unilateral modification (*jus variandi*) may be directly conferred by the legislator (e.g., in Italian law, the employer's right to change employees'ì tasks[2]).

The modification to contract may also originate externally to the parties, such as when a court declares a contract partially void due to formal or substantial flaws or when unexpected events outside the control of parties affect the contractual relationship. These last cases are particularly relevant in long-term contracts and require legal solutions in order to deal with occurrences that couldn't be anticipated by the parties.

[2] Art. 2103 Codice Civile.

Contracts are entered into with the expectation that both parties will fulfil their obligations as agreed upon. The roman brocard *pacta sunt servanda* (agreement have to be respected), constitutes a foundational principle of contract theory: the contract is a mutual promise in which each party can hold the other one to the promised performance However, parties accepted to be bound by those promises under the particular set of circumstances standing at the time of stipulation: if these circumstances change, this commitment may need to be revised. For example, the beginning of a war could drastically raise the price of commodities needed for production or the outbreak of a pandemic could halt factories' activity. Such changes of circumstances may make performance of contractual obligations impossible, excessively onerous or even deprive the performance of its original utility for the counterparty. To address these situations, a legal basis to justify non-performance or to legitimately request an amendment of the contract is provided by the principle of *clausola rebus sic stantibus*: a contract is binding only as far as the relevant circumstances remain the same as they were at the time of conclusion of the agreement [22].

The matter is known to most legal systems but it is addressed in different ways. In common law systems, courts have elaborated the doctrine of *frustration*[3]. Frustration represents an excuse when an unforeseen change in circumstances deprives the contract of all utility for one party, even though the material capacity to perform the obligation is not affected. In the United States, one can also find the notion of *impracticability* which, recognised by the Uniform Commercial Code (§ 2-615), offers a defense in case performance became impractical due to a contingency that the parties, at the time of stipulation, assumed would not take place. Unlike under *frustration*, impracticability also applies where performance has become extremely difficult or onerous for one party [18].

In civil law countries, the issue is often addressed by national legislation. For example, in France and Italy, the respective civil codes include a provision dealing with supervening events that render the performance excessively onerous for one party (Art. 1195 Code Civìl, art. 1467 Codice Civile). These norms give to the burdened party the possibility to request an amendment of the contract in order to recover the original contractual balance.

In contract practice, especially in international context, the eventuality of unexpected changes in circumstances that might affect the agreement is usually dealt with by specific clauses defining the conditions and procedures to be followed in such cases. By writing such clauses, parties can avoid the uncertainty of being at the mercy of the relevant national legislation and adjudication [10]. The main examples in this sense are *force majeure* and hardship clauses. While force majeure occurs when performance becomes impossible and usually leads to suspension or termination, hardship cases take place where the equilibrium of a

[3] The *frustration* doctrine was originally developed by English courts as a consequence to the famous *Coronation* cases in 1902–1904. The cancellation of King Edward VII's coronation frustrated the purpose of the defendants who leased apartments to witness the procession from a privileged spot. See *Krell v Henry* (1903).

contract is altered making compliance significantly more onerous for one party[4]. In these cases, the burdened party is usually entitled to request an amendment of the contract, or its termination.

The contract defined in the language of Sect. 2 do not provide ways to deal directly with modifications during its execution. Therefore, in order to model either *force majeure* or hardship, one should anticipate all the appropriate amendments for each possible circumstance at the time of first drafting. While this is easy for termination clauses (it is enough to include a transition to a final state), it is clearly impossible for other kinds of amendments [16]. Even an attempt to do that would raise drafting costs and introduce huge complexities in the contract, thus nullifying one of the main objectives of *Stipula*, which is to have a simple and intelligible code. For these reasons, in the next section, we discuss an extension of the language with a feature that allows parties to remove or amend the effects of a contract in a direct and intelligible way.

4 *Stipula* with Amendments

Switching to the programming perspective, contract amendments entail a modification of the contract protocol. However, we notice that different kinds of amendment produce different effects on the code. Some cases may only require the addition of a new function or the removal of an old one, while others may affect the whole existing protocol. Moreover, old and new codes will often have to be operational at the same time, potentially giving rise to invocation conflicts. In order to implement amendments, it is necessary to handle these situations effectively.

Technically, amendments are runtime adjustments to the contract's behavior. In programming languages, these runtime adjustment are usually expressed by *higher-order* functions that may also take code as an input parameter. This code is run when such a function is invoked, thus possibly modifying the function's behaviour. A *higher-order* function in *Stipula* is

```
    @Q Party: amendment (|X,Y,Z|) {remove X add Y run Z}
```

This function carries three parameters in brackets $(\!|\cdot|\!)$, whose roles are indicated by the directives `remove` X `add` Y `run` Z. X is a sequence of function names that will be removed from the contract (the terms of the sequence may be either `f` or `A:f` or `Q A:f`); Y represents the new code added and may include declarations of new parties, fields, assets as well as new functions that will amend the contract; Z is the body of the higher-order functions. Therefore Z is defined in the form $\{\,..\}$ ⇛ @`Q`, and may also include the new elements defined in Y. It is worth mentioning that, while X and Y are potentially empty sequences (i.e., optional parameters), Z is mandatory (it is, in fact, necessary to at least define the state that the contract will transit to after the function invocation).

The formal semantics of *higher-order Stipula* has been defined in [15]; the purpose of this paper is rather to discuss the underlying legal basis to the new

[4] Art. 6.2.2 UNIDROIT Principles.

higher-order feature and to provide practical design patterns. We will do this by presenting real-world cases of contract amendments.

We build on the example presented in Sect. 2, drawing from the dispute arose in case n. 10351 of the ICC Court of Arbitration (hereinafter, "the dispute"). In addition to the contract code in Fig. 1, parties provide an amendment clause which is represented by the *higher-order* function

```
@Waiting_order Seller,Buyer: amendment(|X,Y,Z|){remove X add Y run Z}
```

that can be called by `Seller` and `Buyer` to introduce a modification. For the purposes of this section, we assume that parties find an agreement on the amendment outside the contract. We discuss in Sect. 5 how to constrain amendments in *higher-order Stipula*.

4.1 Additive Amendment

From the history of the dispute[5] we can see that parties included in their contract a so-called take-or-pay clause. A *take-or-pay* clause is a provision in a contract stating that a buyer has the obligation of either taking delivery of goods from a Supplier or paying a specified penalty amount to the Supplier for not taking them. This kind of provision benefits both parties because reduces the risk of the investment on the supplier side, and allows the buyer to negotiate a lower price[6] For the purposes of our example, we can assume that an external event (*e.g.*, a war) affected the gas market as to increase the contractual risk on all market participants. Therefore, parties agree to introduce a take-or-pay clause to reduce their exposure for the following two years of their business relationship, providing a penalty in case the purchase threshold is not reached. To do so, `Supplier` invokes the `amendment` function as shown in Fig. 2 from the top.

At this stage, all the function's parameters (`remove` X `add` Y `run` Z) are instantiated. The first parameter ε specifies that there are no functions to be removed from the old code. The second parameter, $\mathbb{D}$, defines a list of new functions. They generally reflect the original code presented in Fig. 1 but with some additions necessary to implement the *take-or-pay* clause.

Two fields – `threshold` and `t_thre` – as well as the asset `penalty` are introduced. The fields respectively represent the amount of gas that the buyer committed to buy and the time span within which the parties have committed to do it (two years), while the asset will be used to store the escrowed money for the penalty.

Being the *take-or-pay* clause essentially an obligation (on the buyer), *Stipula* uses *events* to model it (see Sect. 2). In particular, the function `take_or_pay` is used to escrow the penalty fee (line 5) and to schedule an event (lines 6–8) that can be read as follows: if, after two years, in whatever state `@X` of the contract, the threshold amount of gas to be purchased will not be reached yet, then the penalty fee will automatically be sent to `Supplier`. Otherwise, the penalty fee is returned to the `Buyer`.

[5] Available at https://tinyurl.com/Case10351.
[6] investopedia.com, available at https://tinyurl.com/5xxjp997.

```
1     Supplier: amendment⦇ε, 𝔻, {2y → t_thre  100 → threshold} ⇛ @Restart⦈
```

where 𝔻 is:

```
1    fields: threshold, t_thre
2    asset: penalty
3
4    @Restart Buyer: take_or_pay()[t] (t==1000){
5        t ⊸ penalty
6        now + t_thre >> @X {
7            (threshold>0) penalty ⊸ Supplier  ;  penalty ⊸ Buyer
8        } ⇛ @X
9    } ⇛ @New_Start
10
11   @New_Start, @New_Waiting PriceProvider: update_price(p)[] {
12           p → price
13           p → Supplier
14           price → Buyer
15       } ⇛ @New_Waiting
16
17   @New_Waiting Buyer: place_order()[w] {
18           w / (price×formula) → order
19           order → Supplier
20           w ⊸ wallet
21   } ⇛ @New_Order
22
23   @New_Order Supplier: send_gas()[g] (g == order){
24           g ⊸ Buyer
25           wallet ⊸ Supplier
26           threshold-order → threshold
27   } ⇛ @New_Waiting
28
29   @New_Waiting Supplier,Buyer: amendment⦇X, Y, Z⦈  {remove X add Y run Z}
```

Fig. 2. Take-or-pay amendment

After the execution of this function is over, the contract transits to the new state `@New_Waiting`. Both the `update_price` and the `place_order` function mirror the ones present in the old version of the contract, while `send_gas` features an important difference. After the Supplier has received the payment for the gas sold, the `threshold` field is updated by subtracting to it the value of gas just purchased by the buyer (line 25). This allows the automatic assessment of obligation compliance made by the event in lines 6–8.

Finally, the third parameter of the `amendment` defines its body. The newly introduced fields are instantiated with the values agreed by the parties and the contract moves to the new state `@Restart`. In this state, the buyer can invoke the `take_or_pay` function to actually implement the amendment.

Notice that 𝔻 introduces a whole new set of contract states and none of the new functions provide transitions to previous states from Fig. 1. This means that the old code is completely deactivated, even though it is not definitively removed. In fact, it will be enough to invoke again the `amendment` function and make a transition to an old state to render it operational again. For example, after the two years negotiated by parties, they will be able to get to the old version of the contract. We call this kind of amendment *additive*: it introduces a whole new piece of protocol replacing the old one, and the two never overlap.

```
    Supplier,Buyer: amendment(| ε, 𝔻, {"Provider_revision"→~} ⇛ @Waiting_Order |)
```

Where 𝔻 is:

```
1  @Waiting_Order Supplier,Buyer: price_revision(x)[] (x=="dispute_provider")
    {
2       x → Supplier, Buyer
3  } ⇛ @Meeting
4
5  @Meeting Supplier,Buyer: define_Provider(| X, Y, Z |)   {remove X add Y run Z}
```

Fig. 3. Price revision clause amendment

4.2 Overriding Amendment

Along the course of the contractual relationship, parties from the dispute frequently modified the contract, particularly in relation to the determination of price. With one of these amendments, in 1981, parties introduced a modification to the price revision clause, providing for the possibility to replace the price reporting agency where this had stopped publishing reliable data necessary to the price determination. Should one party dispute the publications, they would meet to negotiate a new trusted source.

In order to introduce such a possibility, the `amendment` function is invoked by either party as shown in the first line of Fig. 3. Also in this case, there is no function to remove (the first argument is ε). The short body of the amendment function, the third parameter, is indicated directly in the first line. It simply communicates the title of the modification to all parties and makes a transition to the `Waiting_Order` state, already existing in the original contract.

In contrast to the previous example, here the new code $\mathbb{D}$ does not substitute completely the old set of states, therefore old functions are kept operational. $\mathbb{D}$ introduces a new version of the `price_revision` function from Fig. 1 and implements the possibility of calling a parties meeting in case the provider becomes unreliable. Having the two functions the same name, the same callers and same state, the new and the old `price_revision` function clearly overlap with each other and enter into conflict once they are invoked. *Higher-order Stipula* handles conflicts though priorities and by leveraging constraints. As a general rule, conflicts are resolved by giving priority to functions of the newest code. However, this can be handled more smoothly by providing specific constraints that have to be satisfied. In this case, the newest `price_revision` requires the input `x` to be equal to the string `"dispute_provider"` (line 1). This means that, whether parties want to use that function to change provider, they will have to fulfill that condition, otherwise they will just call the old version from Fig. 1 and simply modify the price formula. Once the contract has transited to the `@Meeting` state (line 3), parties are enabled to call the `define_provider` function. This is modeled as a *higher-order* one in that it is necessary to modify the code to add a new party to the contract.

We call this kind of amendment *overriding*: old and new codes are both operational at the same time and are linked with each other. Potential conflicts are solved through priorities and constraint management.

```
Buyer: define_Provider(|update_price, 𝔻′, {"New_Price_Provider" →~} ⇛ @Waiting_order|)
```

where $\mathbb{D}'$ is:

```
    parties: NewPriceProvider = NewPriceProvider

    @Waiting_order NewPriceProvider: update_price(p)[] {
        p → price
        p → Supplier,Buyer
    } ⇛ @Waiting_order
```

Fig. 4. Substitution of Provider

After some time, according to parties' opinion, the data published by the reporting agency stop being truly representative of the market situation and therefore they decide to change provider by triggering the newly added clause. To implement this kind of amendment, the intervention needed is two-folded: on the one hand, a new party has to be introduced into the contract, while, on the other, the old price provider has to be prevented from interacting with the contract. Once in the `@Meeting` state, this can be obtained by invoking `define_Provider`.

As shown in Fig. 4, the first parameter of `define_Provider` indicates `update_price`, meaning that the function is removed from the old code of Fig. 1. At the same time, $\mathbb{D}'$ introduces the new price provider as a party to the contract and defines a new `update_price` function that can be only accessed by the new provider. Lastly, the body of the function in line 1 communicates the name of the new provider to all parties and moves the contract to `@Waiting_Order`.

At this stage, the provisions regulating the parties' relationship results from the combination of the original agreement, plus the two following amendments. This is represented by the code from Fig. 1, the code $\mathbb{D}$ from Fig. 3 and $\mathbb{D}'$ from Fig. 4, which are all operational at the same time. Just like the previous one, this amendment does not introduce a new set of states: old and new code uptimes are preserved simultaneously. However, no overlaps occur. The old and new `update_price` differ as to the party that is able to invoke it. For this reason, the removal of the old function is necessary to prevent the old provider form interacting with the contract. We call this type of amendment *overriding with removal.*

5 Amendment Supplements: Agreements and Constraints

The purpose of the present work was to discuss how to technically define amendments in *higher-order Stipula*. One basic condition for modifying a legal contract was left out of the representation: the *mutual consent of the parties.* In fact, excluding the cases where the legislator confers to one party the power to modify the contract unilaterally, in all other cases, the manifestation of mutual agreement is required[7]. In order to deal with this principle within the contract, it is necessary to allow parties to express the consent to amendments at runtime.

[7] Art. 2.1.1 UNIDROIT Principles.

```
constraints [ (parties: fixed;)?  (fields: z̄ constant;)?
                    (assets: k̄ not-decrease;)?  (reachable states: @Q̄)? ]
```

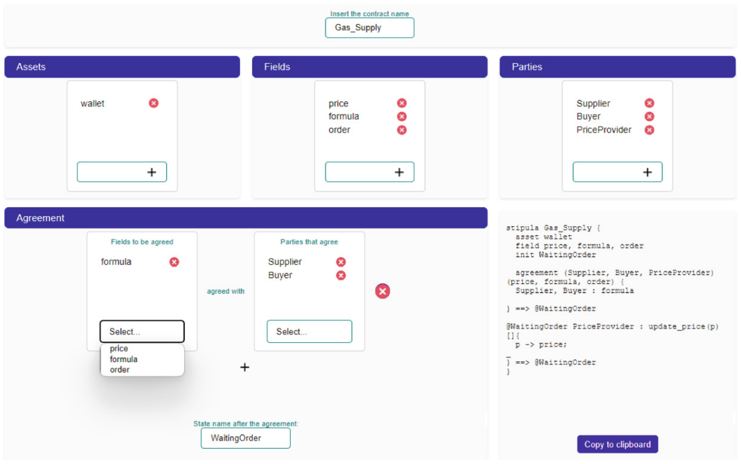
Fig. 5. Static constraints on Amendments

The *Stipula* language already features an agreement clause which is triggered at the deployment of the contract and that corresponds to the "meeting of the minds": every one must accept the terms of the contract in order for the legal bound to arise. By following the same pattern, the *higher-order Stipula* prototype [7] already provides an agreement clause that occurs in correspondence of every amendment that requires the consent of every party.

In addition, it is possible for parties to set specific boundaries to contract amending. In fact, there may be several factors providing limitations to parties freedom in amending an agreement. Beside the general limit represented by legal systems' mandatory rules (*cf.* the principle in Art. 1418 of the Italian Civil Code, the Art. 1:103 of PECL – the European Principle of Contract Law – and the Art. 1.4 of the international Unidroit Principles) legislators can also provide for more domain specific boundaries such as limits to prices for basic commodities, employees' salary or loan interest rates. Moreover, parties themselves may want to limit their behaviour.

In order to specify and implement such possibilities, in [15] we have studied a technique for defining amendments that are mandatorily accepted by parties. That is, the technique allow parties to agree on the type of amendments they might accept in the future when a contract is stipulated. To this aim, we extend *higher-order Stipula* with the syntactic clause in Fig. 5. This clause allows us to define an amendment verifier that automatically checks whether amendments comply or not with the restrictions in the clause.

Every constraint in the clause may be missing (when all the constraints are empty then "`constraints [ ]`" is omitted and we are back to the basic syntax). The constraint "`parties: fixed`" specifies that amendments cannot modify the set of parties. If this constraint was present in the *gas supply* contract, then the amendment $\mathbb{D}'$ of Fig. 4 would have been rejected. The constraint "`fields:` $\overline{\mathtt{z}}$ `constant`" disables updates of fields in $\overline{\mathtt{z}}$. The constraint "`assets:` $\overline{k}$ `not-decrease`" protects private assets to be drained by unauthorised parties. Finally, the constraint "`reachable states:` $\overline{\mathtt{Q}}$" guarantees that, whatever contract update is performed, the states in $\overline{\mathtt{Q}}$ can be reached from the ending state of the amendment. This is because, for example, the corresponding functionalities cannot be disallowed forever. The foregoing clause has not yet been integrated in [7]: the prototyping work is ongoing.



```
stipula Gas_Supply {
  asset wallet
  field price, formula, order
  init WaitingOrder

  agreement (Supplier, Buyer, PriceProvider)
(price, formula, order) {
  Supplier, Buyer : formula

} ==> @WaitingOrder

@WaitingOrder PriceProvider : update_price(p)
[]{
  p -> price;

} ==> @WaitingOrder
}
```

Fig. 6. The *Stipula* editor - Heading and Agreement

6 The *Stipula* Prototype and its Graphical Interface

Stipula has been prototyped, therefore one can experiment enforcement of contractual conditions, traceability, and outcome certainty. The prototype is a Java application that is available on the github website of the project, together with a number of sample contracts [7]. The development has taken three months and ∼3000 lines of Java code. In order to use it, it is necessary to clone the repository and install the prototype following the instructions provided on the website. Once installed, it is possible to write a brand new contract and run it to test its execution.

The prototype is also provided with the *higher-order Stipula* extension, supporting runtime amendments. At the implementation level, this functionality is achieved by updating the contract instance with changes contained in the *higher order*. Once the changes are overwritten, the new version is deployed. A technical explanation of the prototype design is contained in [6].

Although *Stipula* features an intuitive syntax, the project is provided with a user-friendly graphical interface to guide the drafting. This is clearly useful for first-time users as well as for users that are already familiar with the language in that avoids potential syntactic mistakes. Figure 6 shows how the interface is presented. For the heading of the contract, the user has to fill in the contract name, the parties, assets and fields in their corresponding windows. For the agreement clause, it is necessary to choose among the already declared fields and parties that are going to comply with the fields' values. As the editor is filled in, the corresponding *Stipula* code is automatically created and updated real-time in the box on the bottom-right. Similarly, writing functions requires declaring both the starting and ending states, the parties entitled to call it and the input parameters needed. Afterwards, the user can choose from a list of possible operations that can be executed by the function call (field update, asset transfer, event, etc.). Once the code is ended, it can be copied in the prototype and executed.

7 Related Works

The digital representation of legal contracts has long been explored, for the main purpose of monitoring and automating contract-related procedures [11,17]. In this context, substantial work has been done in the wake of 'Ricardian Contracts', originally introduced by Ian Grigg in 1996 [13]. This approach consists in linking written contract documents with the related computer executable code via *parameters*. Through the use of mark-up languages, the natural language document is annotated to indicate which parts of the contract are the values to be inputted to the code. Further works extended this approach by building a template model for contracts [4] and providing specifications to increase contract's intelligibility [19]. However, the capability of capturing the semantics of an agreement by annotating natural language documents is limited to the input that is provided by the tagged data. Moreover, operational code may still remain opaque to legal professionals, thus preventing the validation of whether it is faithful to the actual agreement [3].

A different approach to express contracts is represented by Domain-Specific languages (DSL). A well-designed, relatively understandable DSL for legal contracts has the advantage of keeping code and agreement (or a straightforward representation of it), within a single artefact. With a single artefact to deal with, it is simpler to check whether the meaning of the agreement and its code implementation match [3]. Such a contract can still be coupled with natural language explanations of the meaning of the code, but the code, rather than these explanations would provide the binding formulation of the contract. Different formalism and approaches have been studied in the literature. For instance Flood and Goodenough have described a loan agreement (in the financial domain) as a particular kind of finite state machine [9]. These machines are mathematical entities used to describe systems with finite set of states and transitions, where transitions allow movements from a state to another in response to given inputs (*events*). While this approach is interesting when the contract is simple enough, it becomes cryptic when the contract is more complex. In particular, it becomes hard to connect the machine to the standard formulation of the contract in natural languages.

Another interesting technique is based on Controlled Natural languages (CNL) [1,8]. A CNL resembles natural language in wording, but is based on formally defined syntax that is automatically converted to a programming language. As a consequence, the code is easily readable. However, due to the constrains imposed by the CNL, it may result harder to write the contract (with respect to natural language) because the formalism may miss computational constructs. It has also been argued that a CNL might represent a "false friend" for the user [14], i.e., it might induce the user to assume that a CNL-expression has the same meaning as natural language expression, which might not always be the case.

Declarative specifications could provide advantages in formally representing legal contracts and reasoning upon them. For example, they can be more compact than other paradigms, therefore easier to draft and to verify, as well as easier to understand by parties [12]. *Stipula* commits to an imperative paradigm that allows one to represent in a more direct way the stages of contracts' life-cycle by means of a state-aware programming style. Additionally, the pretty straightforward syntax should hopefully make *Stipula* as intuitive as the declarative specifications for legal professionals. The formal semantics defined in [6,15] should allow one to define automatic analyzers that verify legal contract correctness.

Higher-order Stipula is a DSL that is based on state-oriented programming with explicit management of assets and with higher-order to express runtime modifications of the code. In our formalism, states are not finitely many because the contracts have memories that store settings and assets. Rather, states are used to express permissions and prohibition of invoking functionalities by contract parties.

8 Conclusions

The present work showcased an extension of *Stipula* for amending legal contracts at run-time. The extension relies on higher-order functions and allows one to program situations were the old code is completely replaced by new one as well as situations where old and new code are both operative and coexist. Overall, we believe that the higher-order mechanism is a simple and intelligible feature that may assist legal practitioners in programming contract amendments.

Up-to our knowledge, *higher-order Stipula* is the first legal contract language natively integrating amendments in its syntax. From a legal perspective, we believe this technique to reflect contractual practice, where contracts include clauses that allow for future amendments. In our context this is done though the higher-order predicate that enables the revision of the contract, i.e., the removal of old clauses and the insertion of new ones. Simply terminating an instance of the contract and deploying a new version of it would depart legal practice, possibly resulting less intuitive for legal professionals. From the programming perspective, the *higher-order* has the advantage that amendments may be analysed using the same techniques on which the first-order language is based. The compliance assessment of the types of the amendments with respect to the types of the original code is done by using the same original type inference system. By exploiting this property, for example, one can design techniques for constraining amendments at the moment of the drafting, as shown in Sect. 5. Adding a runtime extension to some existing tool that copes with amendments is not the same as it would be unconstrained.

References

1. Lexon language (2022). http://lexon.org/. Accessed 13 Apr 2023
2. Caldarelli, G.: Unilateral modification of long term contracts: American change of terms clauses and Italian Ius Variandi from a 'relational' point of view. Eur. Rev. Contract Law **17**(1), 37–53 (2021)
3. Clack, C.D.: Languages for smart and computable contracts. CoRR abs/2104.03764 (2021). https://arxiv.org/abs/2104.03764
4. Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions. CoRR abs/1608.00771 (2016). http://arxiv.org/abs/1608.00771
5. Crafa, S., Laneve, C.: Programming legal contracts – a beginners guide to *Stipula*. In: Ahrendt, W., Beckert, B., Bubel, R., Johnsen, E.B. (eds.) The Logic of Software. A Tasting Menu of Formal Methods. LNCS, vol. 13360, pp. 129–146. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-08166-8_7
6. Crafa, S., Laneve, C., Sartor, G., Veschetti, A.: Pacta sunt servanda: legal contracts in *Stipula*. Sci. Comput. Program. **225**, 102911 (2023)
7. Crafa, S., Laneve, C., Veschetti, A.: The Stipula prototype. https://github.com/stipula-language/stipula. Accessed 31 Mar 2023
8. Datoo, A., Kowalski, R.: Logical English meets legal English for swaps and derivatives. Artif. Intell. Law **30**(2), 163–197 (2022)
9. Flood, M.D., Goodenough, O.R.: Contract as automaton: representing a simple financial agreement in computational form. Artif. Intell. Law **30**(3), 391–416 (2022)
10. Fontaine, M., De Ly, F.: Drafting international contracts. BRILL (2006)
11. Governatori, G.: Representing business contracts in RuleML. Int. J. Coop. Inf. Syst. **14**(02n03), 181–216 (2005)
12. Governatori, G., Idelberger, F., Milosevic, Z., Riveret, R., Sartor, G., Xu, X.: On legal contracts, imperative and declarative smart contracts, and blockchain systems. Artif. Intell. Law **26**, 377–409 (2018)
13. Grigg, I.: The Ricardian Contract (1996). https://iang.org/papers/ricardian_contract.html. Accessed 13 Apr 2023
14. Idelberger, F.: The uncanny valley of computable contracts: analysis of computable contract formalisms with a focus towards controlled natural languages. Ph.D. thesis, European University Institute (2022)
15. Laneve, C., Parenti, A., Sartor, G.: Legal contracts amending with *Stipula*. In: Jongmans, S.S., Lopes, A. (eds.) COORDINATION 2023. LNCS, vol. 13908, pp. 253–270. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-35361-1_14
16. Mik, E.: Smart contracts: terminology, technical limitations and real world complexity. Law Innov. Technol. **9**(2), 269–300 (2017)
17. Milosevic, Z., Gibson, S., Linington, P., Cole, J., Kulkarni, S.: On design and implementation of a contract monitoring facility. In: Proceedings of the First IEEE International Workshop on Electronic Contracting, pp. 62–70 (2004)
18. Palmer, V.V.: Excused performances: force majeure, impracticability, and frustration of contracts. Am. J. Comput. Law **70**(Supplement_1), i70–i88 (2022)
19. Palmirani, M., Cervone, L., Vitali, F.: Intelligible contracts. In: 53rd Hawaii International Conference on System Sciences, pp. 1780–1789 (2020)
20. Pfeiffer, H.K.: The Diffusion of Electronic Data Interchange. Springer, Cham (2012). https://doi.org/10.1007/978-3-642-51559-0
21. Surden, H.: Computable contracts. UCDL Rev. **46**, 629 (2012)
22. Zimmermann, R.: The Law of Obligations: Roman Foundations of the Civilian Tradition. Juta and Company Ltd. (1990)