The Stipula Platform
a workbench for programming and analyzing legal contracts

Cosimo Laneve[0000−0002−0052−4061]

DISI, University of Bologna, Italy
cosimo.laneve@unibo.it

Abstract. *Stipula* is a domain-specific programming language specifically designed to model and enforce legal contracts. This paper presents the foundational design principles of *Stipula*, focusing on its ability to express and regulate permissions, obligations, prohibitions, and escrows. We introduce a comprehensive suite of tools developed to support contract development in *Stipula*, including a *visual code editor* for intuitive contract authoring, an interpreter for automatic execution, and a number of analyzers for verifying several legally relevant properties (unreachability of clauses, absence of frozen assets, etc.). These tools are integrated into a unified *workbench*, designed to empower legal practitioners and developers to write, test, debug, and manage smart legal contracts effectively and efficiently.

1 Introduction

Traditional legal texts are typically expressed in natural language, which is inherently ambiguous and susceptible to multiple interpretations. This linguistic flexibility, while essential for human communication, poses significant challenges for automated processing, analysis, and verification. The digitalization of legal texts offers substantial benefits: it can increase efficiency through faster information retrieval and even enable the automatic execution of well-specified procedures. Moreover, digital representations facilitate improved data organization and promote greater transparency in legal processes. However, the computational treatment of legal texts remains a difficult task. Their complexity, coupled with the expressiveness and ambiguity of natural language, often necessitates human judgment to achieve reliable interpretation.

To address these challenges, several research initiatives have proposed domain-specific programming languages for the specification of legal contracts *e.g.*, [35, 29, 30, 38]. These languages introduce a precise syntax that mitigates some of the ambiguities of natural language, and often provide graphical interfaces to help map normative elements directly to executable code. However, while such efforts advance the expressiveness and accessibility of digital contracts, they typically devote limited attention to the formal verification of correctness. This limitation is particularly critical in the legal domain, where ambiguities or unintended execution behaviours may result in severe legal and financial consequences. Although progress has been made in developing methodologies and tools for the formal verification of legal contracts, their adoption in practice remains rare. One major obstacle is the inherent complexity of formal reasoning tools, which discourages their use by legal practitioners and restricts their integration into standard workflows.

To overcome the current lack of automatic verification techniques, in 2021 we introduced *Stipula* [11, 12, 27], a domain-specific language explicitly designed for the specification of legal contracts. *Stipula* is built around a small set of concise and intelligible primitives that directly capture the distinctive elements of contracts. The language is grounded in a formal operational semantics, ensuring that its behaviour is rigorously defined and thus amenable to automated verification.

The supporting toolchain [13] has been developed to facilitate the entire contract lifecycle. It includes a *visual code editor* that enables intuitive authoring of contracts, an interpreter for automatic execution, a *type inference system* that minimizes the need for explicit annotations, and an *unreachability analyzer* that identifies clauses which can never be executed. Ongoing work extends this suite with a *liquidity analyzer*, which guarantees that no assets remain indefinitely frozen within a contract. While the theoretical framework of liquidity analysis has already been formalized, its implementation is currently under development. All of these tools are integrated into a unified *workbench*, designed to support both legal practitioners and developers in writing, testing, debugging, and managing smart legal contracts. The workbench provides a seamless environment that promotes efficiency, correctness, and accessibility, thereby bridging the gap between formal verification techniques and their practical use in the legal domain.

The development of the platform began in 2021 with the interpreter and type inference system described in [12]. It was later expanded through two undergraduate theses at the University of Bologna [36, 21], which introduced the visual editor and the unreachability analyzer studied in [26]. The environment that integrates and harmonizes these tools was also developed as part of an undergraduate thesis [17]. This paper consolidates these earlier contributions and provides a unified account of the resulting toolchain. Our aim is to present the platform in a way that is both technically rigorous and accessible to practitioners and legal experts, enabling them to experiment with and adopt the system without requiring extensive prior training in formal methods or programming.

The remainder of the paper is organized as follows. Section 2 introduces the *Stipula* language and illustrates its main constructs through an example that demonstrates how obligations, permissions, and events can be naturally expressed. Section 3 presents the execution environment, while Section 4 describes the visual editor that supports contract authoring. The integration of type inference within the interpreter is discussed in Section 5. Section 6 outlines the analysis technique used to identify unreachable clauses, whereas Section 7 is devoted to liquidity analysis, which determines whether contractual assets may remain indefinitely frozen. Section 8 situates our contribution in the context of related research on contract languages and formal verification. Finally, Section 9 concludes the paper and discusses avenues for future work.

2 Legal contracts in *Stipula*

We introduce *Stipula* through a case study modeling a legal contract for bike rentals, structured into three articles:

1. This *Agreement* shall commence when the Borrower takes possession of the Bike and remain in full force and effect until the Bike is returned to Lender. The Borrower shall return the Bike within `k` hours after the rental and will pay Euro `cost` in advance where half of the amount is of surcharge for late return.
2. *Payment.* Borrower shall pay the amount specified in Article 1 when this agreement commences.
3. *End.* If the Bike is returned within the agreed time specified in Article 1, then half of the `cost` will be returned to Borrower; the other half is given to the Lender. Otherwise the full `cost` is given to the Lender.

These articles are transposed into *Stipula* as follows:

```
1  stipula Bike_Rental {
2    fields cost, k
3    assets wallet, bike
4    agreement (Borrower, Lender){
5        Borrower, Lender : cost, k
6    } ⇛ @Inactive
7    @Inactive Lender : offer[b] {
8        b ⊸ bike // the bike access code is stored in the contract
9    } ⇛ @Payment
10   @Payment Borrower : pay[x]
11     (x == cost) {
12       x ⊸ wallet // the contract keeps the money
13       bike ⊸ Borrower // the Borrower keeps the bike access code
14       now + k ≫ @Using {
15           wallet ⊸ Lender // deadline expired: the whole wallet to Lender
16       } ⇛ @End
17   } ⇛ @Using
18   @Using Borrower : end { // bike returned before the deadline
19     0.5 × wallet ⊸ wallet, Lender // half wallet to Lender
20     wallet ⊸ Borrower // half wallet to Borrower
21   } ⇛ @End
22 }
```

Listing 1. The `Bike_Rental` contract in *Stipula*

The contract named `Bike_Rental` is defined using the `stipula` keyword and distinguishes between two core types of entities: `fields` at line 2, which store scalar values (*e.g.*, integers, booleans), and `assets` at line 3, which represent tangible or transferable items such as currencies, fungible tokens, and non-fungible tokens. The language provides ad-hoc operations for manipulating assets, thereby justifying a clear semantic distinction with fields.

Lines 4–6 define the `agreement` clause, a distinctive feature of a legal contract that models the moment when, after the possible negotiation of the contractual content, the parties, which are listed at line 4, express consent on the values of fields (in this case `cost` and `k`) – the *meeting of the minds.* The contract then starts in the state `Inactive` – line 6 –, thus producing its legal effects.

Stipula adopts a *state-machine programming style* to model *permissions*, *prohibitions* and *obligations* that usually change over time according to the actions that have been done (or not). For example, in the `Inactive` state, the contract is giving permission to the Lender to send an access code that will be used by the Borrower for unblocking the bike. Once the code has been sent, the contract goes into a state `Payment` where the Lender can no more withdraw from the rental and the unique permitted operation is the payment of the rental by the Borrower. In *Stipula*, permissions are specified in the *function signatures* that define the state, the party that can invoke the function and the arguments.

In the case of `offer`, the argument is an *asset* that represents the code to unblock the bike; for this reason it is in square brackets (on the contrary, standard values are in round brackets). The *move operation* `b` $\multimap$ `bike` at line 8 carries the token in `b` to `bike`, thus *emptying the content of* `b`. In this case the token is *non-fungible*, *i.e.* indivisible, which means that `bike` is an empty asset that will retain the code `b` after the move.

The function `pay` at lines 10-17 specifies the payment of the rental by the Borrower. In particular, the Borrower has to sent an asset `x` that corresponds to the agreed `cost` of the rental. The currency is stored in the `wallet` – line 12 – and, at the same time, the bike code is sent to the Borrower – the operation `bike` $\multimap$ `Borrower` at line 13 –, thus letting Borrower use the bike. The body of `pay` also contains another distinguishing feature of *Stipula*: the *event* clause. In this case, the event is scheduled at time `now + k` – lines 14-16 – and it asserts that, if the bike is not returned within the agreed time of `k` hours (starting from `now`, the time when the function `pay` is invoked), the Borrower has to pay the whole amount in the `wallet` as specified in Article 3. This mechanism is employed in *Stipula* to enforce the Borrower's *obligation* to return the bike within the specified timeframe. Notably, the `wallet` asset functions as an *escrow*, temporarily holding funds as a guarantee of compliance with the contractual terms.

The function at lines 18-21 defines the timely return of the bike by the Borrower (Article 3); the operation `0.5` $\times$ `wallet` $\multimap$ `wallet, Lender` halves the content of the `wallet` and sends that amount to the Lender; therefore the following operation `wallet` $\multimap$ `Borrower` sends to the Borrower the remaining half. The use of arithmetic operations on the `wallet` asset indicates that it is *fungible*, meaning its units are interchangeable and divisible, as is typical for digital currencies or tokens representing monetary value.

Notably, the foregoing code also reflects an implicit trust placed by both the Lender and the Borrower in the contract itself, which acts as an autonomous intermediary capable of securely storing sensible information and assets. In particular, assets can be temporarily retained by legal contracts and subsequently redistributed when specific conditions are met. To support this, the language treats assets as first-class values, equipped with dedicated operations for explicit and controlled management (the move $\multimap$ and the square brackets in functions' signatures). This contrasts with the traditional rental setting, in which the Borrower typically pays the Lender upfront via credit card, and the transaction is mediated – and guaranteed – by a financial institution acting as a trusted third party, often at a cost. In contrast, *Stipula* enables contracts to directly manage asset transfers, thereby eliminating the need for external intermediaries and enabling more autonomous, cost-efficient agreements. We refer to [12] for the complete definition of the syntax and the semantics of *Stipula*.

It is important to highlight a significant consequence of the digitalization of legal contracts. As illustrated by the code in Listing 2, it is possible for the Borrower to accept the contractual terms without actually completing the payment for the rental. In this scenario, the Lender's bike becomes inaccessible: the access code – stored in the `bike` asset – has been disclosed, but no party can make legitimate use of it thereafter.

This outcome is not a failure of the digitalization process itself, but rather a flaw in the original contract design, specifically within the Articles defined at the beginning of the section. In other words, the contract omits a crucial clause that should state:

2.a If the Borrower does not pay within `k`/6 hours, the contract is deemed invalid and the bike is returned to the Lender.

Translating this provision into *Stipula* entails refining the implementation of the `offer` function as follows:

```
@Inactive Lender : offer[b] {
    b ⊸ bike // the bike access code is stored in the contract
    now + k/6 ≫ @Payment { bike ⊸ Lender } ⇒ @End
} ⇒ @Payment
```

Listing 2. Refinement of the function `offer` in the `Bike_Rental` contract

3 Executing *Stipula* contracts

As a principled high-level language, *Stipula* is designed to be implementation-agnostic and does not assume any specific computing architecture. Nevertheless, given that legal contracts are critical instruments with binding obligations, their digital execution requires strong guarantees from the underlying computing platform. These guarantees are essential to ensure the correctness, security, and legal enforceability of the contract's behavior. We identify three critical guarantees:

trust: The platform must guarantee that all parties observe a consistent and tamper-proof execution of the contract. This includes providing a reliable notion of time and supporting verifiable asset transfers between parties and contracts, thus ensuring predictable and enforceable contract behaviour.

authentication: Every function invocation must be authenticated using secure identities (*e.g.*, digital signatures or cryptographic credentials), thus ensuring that only authorized parties can interact with the contract and invoke clauses specific to their roles.

auditability: To support accountability and legal enforceability, the platform must produce auditable logs, such as event traces and transaction histories. These records enable post-execution analysis, facilitate dispute resolution, and support compliance verification.

The prototype [13] is implemented in Java and can be executed on a standard computing platform. The notion of trust is grounded in the assumption that the program operates within a secure and trusted environment – such as a bank, court, or notary office – where the integrity of the execution is guaranteed. Authentication is achieved through security credentials issued and managed by the hosting institution, ensuring that only authorized parties can access and interact with the contract. Auditability is supported by systematically recording execution logs, which are archived daily to enable traceability and post-execution verification. (In [12] we also discuss the implementation of the main elements of *Stipula* on top of a distributed system such as a blockchain.)

Let us show the execution flow of `Bike_Rental`. The process begins with the syntactic analysis of the input code, followed by type inference (see Section 5). After these preliminary steps, the interpreter evaluates the contract's agreement clause, which involves generating and displaying a unique code for each party involved. These codes serve as identifiers and access credentials, enabling secure interaction with the contract. In practical deployments, they would correspond to IDs and passwords assigned to the contracting parties. In the case of the `Bike_Rental` contract, two possible access codes might be:

```
----------
Lender: ef6h4
Borrower: MHWBs
----------
```

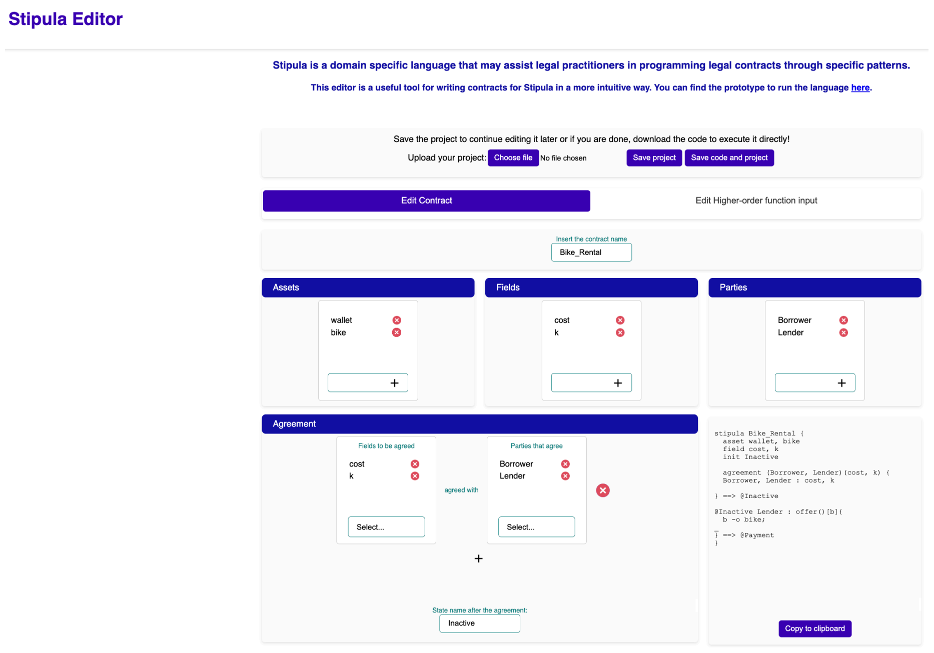
Next, the interpreter prompts the parties to input their assigned access codes, along with the initial values of the fields that require mutual agreement. Once these inputs are provided and their correspondence has been checked, the interpreter identifies and displays the set of functions that are eligible to be invoked at the current stage. For example:

```
# Please, choose which function should run:
Lender.offer()[Asset b]
```

In this scenario, only the Lender is authorized to invoke the offer function. The invocation includes a parameter representing the code used to unlock the bike, typically encoded as a numeric value. In the interpreter, this interaction is simulated by providing the Lender's access code, the name of the function to be invoked, and the corresponding unlock code, as illustrated below:

```
ef6h4.offer()[1234]
```

Afterward, the prototype interpreter executes the body of the selected function, allowing the contract to progress, by enabling the Borrower to initiate the payment for the rental. The `pay` function, in turn, schedules an event to be triggered `k` hours later. For example, if `k` is set to `0.1`, and the Borrower fails to return the



Stipula is a domain specific language that may assist legal practitioners in programming legal contracts through specific patterns.

This editor is a useful tool for writing contracts for Stipula in a more intuitive way. You can find the prototype to run the language here.

Save the project to continue editing it later or if you are done, download the code to execute it directly!

Upload your project: No file chosen

Insert the contract name: Bike_Rental

Assets: wallet, bike

Fields: cost, k

Parties: Borrower, Lender

Fields to be agreed: cost, k

Parties that agree: Borrower, Lender

State name after the agreement: Inactive

```
stipula Bike_Rental {
  asset wallet, bike
  field cost, k
  init Inactive

  agreement (Borrower, Lender)(cost, k) {
  Borrower, Lender : cost, k

} ==> @Inactive

@Inactive Lender : offer()[b]{
  b -o bike;

} ==> @Payment
}
```

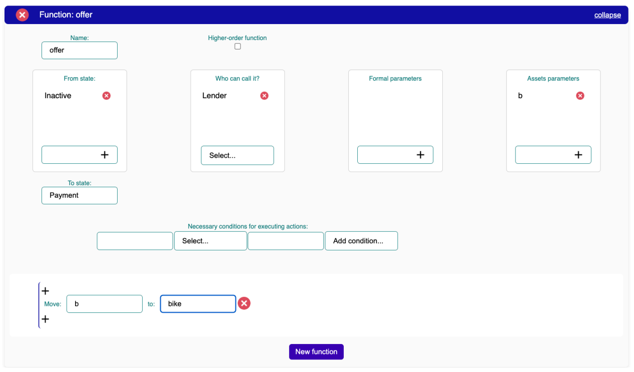
Fig. 1. The Visual Editor Interface: fields, assets, parties and the agreement

bike within 6 minutes, the event is activated automatically. As a consequence, the entire wallet balance is transferred to the Lender.

4 The visual code editor

Stipula is designed to support legal practitioners and individuals with little or no familiarity with programming or formal languages. To enhance accessibility, the language is complemented by a visual interface that facilitates the drafting of contracts through schematic patterns and intuitive graphical representations. This interface lowers the entry barrier for non-technical users, enabling them to express complex contractual logic without writing code directly.

The editor is implemented as a web-based application. Its initial interface is shown in Figure 1. The interface provides dedicated input fields through which the user can define the fundamental components of a contract, including its name, the involved assets and fields, the parties to the agreement, and the essential elements of the agreement clause. This structured input method simplifies the contract creation process and ensures that the resulting code adheres to the language's syntax and semantics. The bottom-right window contains the code of the contract (in the interface we have inserted the components of the `Bike_Rental` contract).



Function: offer

Name: offer

Higher-order function

From state: Inactive

Who can call it? Lender

Formal parameters

Assets parameters: b

To state: Payment

Necessary conditions for executing actions:

Move: b to: bike

Fig. 2. The Visual Editor Interface: functions

Following the same visual design, the editor also supports the definition of functions and events. In these cases, the interface provides graphical tools that guide the user in composing valid statements by offering selectable templates for each language construct – for example, update operations such as `x+1` $\rightarrow$ `x` or asset transfers like `x/2` $\multimap$ `x, A`. This visual support ensures syntactic correctness and lowers the barrier for non-technical users. Figure 2 shows a snapshot of the editor displaying the implementation of the `offer` function.

5 Type inference in *Stipula*

Stipula adopts a type-free surface syntax, deliberately omitting explicit type annotations. This design choice is motivated by the observation that standard legal contracts do not typically involve typed declarations, and such annotations may appear unintuitive or even opaque to non-technical users, particularly legal practitioners. Despite the absence of type annotations, *Stipula* includes a type inference system capable of automatically deriving the types of assets, fields, and function arguments. This system enables early detection of programming errors at compile time, thereby ensuring basic correctness and improving reliability. In this section, we outline the main design principles and technical foundations of the type inference system.

The *Stipula* interpreter has the following *primitive types*

$$T ::= \texttt{real} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{time} \mid \texttt{asset}$$

that mirror the set of values of the language: real numbers, booleans, strings, time values, and assets. In the current prototype a time value is a real number representing hours. For exampl, the *relative time* expressions `now + 2.0` means two hours later (with respect to the execution of the enclosing function) and `now + 0.5` means 30 minutes. *Stipula* also admits *absolute time* values as the date `"2022/1/1:00:15"T`. With respect to the `asset` type, the current implementation does not yet distinguish between divisible and indivisible assets.

Each asset is represented as a sequence of digits, which may encode either a unique identifier or a numerical amount. This simplification reflects the current state of the system, but future releases will refine this model by introducing an explicit distinction between divisible and indivisible assets. For instance, in the `Bike_Rental` contract, the `wallet` asset is considered divisible, as demonstrated by the operation at line 19.

The type inference system in *Stipula* largely follows standard techniques. It begins by associating distinct type variables to each identifier in the program and proceeds by traversing the code to collect a set of type constraints. Once parsing is complete, these constraints are resolved using a unification algorithm, and the type variables are substituted with their inferred types (see [31] for a comprehensive overview of the method). A notable innovation in our approach is that type inference *is re-executed after each function invocation*, allowing the system to refine the types of program elements based on the actual arguments passed to the function and their subsequent assignment to fields. This dynamic refinement enhances both flexibility and precision in the type system.

As usual, type inference is executed the first time after the parsing. In the case of `Bike_Rental`, the interpreter returns

```
TYPE CHECKING ===
Assets:
    wallet type: Asset
    bike type: Asset
Fields:
    cost type: Real
    k type: Time
Functions:
    Lender.offer()[Asset]
    Borrower.pay()[Asset]
    Borrower.end()[]
=================
```

As the reader may observe, following the initial type inference phase, no type variables remain in the program: the types of all elements – fields, assets, and functions – are fully resolved. While this outcome may appear straightforward for assets and functions in the `Bike_Rental` contract, it is less immediately evident in the case of fields. For example, the field `k` is inferred to have type `Time` due to its occurrence within a time expression. Similarly, the field `cost` is assigned type `Real` because of the equality "`x == cost`" at line 11. Since `x` is an asset and assets in this context represent real-valued quantities, the unification process infers `cost` to also have type `Real`.

If the equality "`x == cost`" is removed from the contract, the type inference system is no longer able to deduce a concrete primitive type for `cost`. As a result, the system outputs:

```
. . .
    cost type: Type0
```

This is because, in the absence of a usage context that constrains its type, `cost` remains unconstrained and retains its original type variable (`Type0`) that cannot be resolved to a specific primitive type.

The type of `cost` is determined *after the agreement*, once the Lender and the Borrower have decided its initial value. In fact the system, at that point, will return the refined type:

```
. . .
    cost type: Real
```

6 Spotting unreachable clauses

A substantial problem in the legal contract domain is spotting normative clauses, either functions or events, that will be never applied. In fact, it is possible, that an unreachable clause is considered too oppressive by one of the parties (take, for instance, the event of the Article 3 in Section 2), thus making the legal relationship fail. In [16], we established that the interplay between state evolution, temporal constraints, and non-determinism renders the *unreachability problem* – *i.e.*, determining whether a clause can never be executed – *undecidable*. As a consequence, no complete algorithm can exist for solving this problem in general. Therefore, our analysis must necessarily be conservative: we can aim at a sound analyzer that can safely identify unreachable clauses, though it may not detect all such cases. One such analyzer has been developed in [26], and it is now integrated into the *Stipula* workbench. In the following, we outline the core design principles that guided its development.

The unreachability analyzer determines the set of reachable clauses through a closure operation based on a fixpoint technique. A key challenge in this analysis is the detection of *time anomalies* – clauses whose execution times are incompatible with the overall behaviour of the contract. To address this issue, the analyzer operates over *logical times*, which are symbolic abstractions representing the system clock at runtime, approximated statically. A clause is included in the set of reachable clauses only if its associated logical time is consistent with the logical time of the computation leading to it. If the resulting constraints over logical times are unsatisfiable, the corresponding clauses are deemed unreachable and reported as such. A major complication arises from the presence of *cyclic behaviours*, which induce infinite computations and make direct consistency checks intractable. To circumvent this, we introduce the notion of *linear traces* – finite surrogates of computations in which each clause appears at most once. These traces are well-suited for static time reasoning due to their finiteness, but they lack the expressiveness to fully capture the semantics of cyclic behaviours. To maximize coverage, we adopt a hybrid strategy: the analyzer applies time-based reasoning to acyclic clauses using logical times, while reverting to a standard, untimed analysis for clauses involved in cycles. This combination balances precision and tractability, enabling the detection of many practical cases of unreachability despite the underlying undecidability of the problem.

For instance, applying the analyzer to the `Bike_Rental` contract yields a comprehensive diagnostic trace, which concludes with the following summarized result:

```
"unreachable_code": []
```

The analyzer's output includes a list of the clauses examined, along with the set of linear traces computed – referred to as R in the implementation. A clause is reported under the `unreachable_code` section if it does not appear in any linear trace, indicating that it cannot be executed in any feasible execution path. For example, consider the `Bike_Rental` contract: if a typographical error occurs at line 14, where `@Usng` is written instead of the intended `@Using`, the analyzer will produce the following error message:

```
"unreachable_code": [ "ev('Usng', 'Ev', 14, 'End')" ]
```

where `14` is the line number of the event of `Bike_Rental`.

The unreachability analyzer is also capable of identifying clauses that are conditionally reachable. This situation arises, for example, when the time expressions associated with events depend on contract fields (as illustrated by the `Bike_Rental` contract). In such cases, the continuation of a contract may be possible only under specific conditions on field values?for instance, the analyzer may report a reachability constraint of the form

```
"reachability constraint": [ x >= 5 ]
```

At present, our analyzer incorporates a lightweight constraint-solving mechanism. It performs partial evaluations that exploit the additive structure of time expressions, which suffices to detect straightforward conditions. However, its capabilities are limited: in particular, it does not compute transitive closures of constraints or reason about more complex dependencies. A natural next step in the development of the prototype is to integrate the analyzer with an established, off-the-shelf constraint solver, thereby enhancing its precision and enabling it to capture richer classes of conditional reachability scenarios.

7 Verifying liquidity

Liquidity is a fundamental security property for any program that manages assets, such as those written in *Stipula*. A *Stipula* contract is said to be *liquid* if no asset remains indefinitely locked within the program – *i.e.*, all resources are ultimately redeemable by at least one of the involved parties [3]. For instance, a contract violates liquidity if the body of a function fails to utilize resources transferred by the caller during invocation. Similarly, liquidity is compromised if the contract terminates with any asset holding a non-zero value – indicating that the resource has not been fully consumed or returned.

The concept of liquidity has been studied extensively, and multiple formalizations exist in the literature. In our previous work [10, 25], we adopt the notion of *multiparty strategyless liquidity*, as defined in the taxonomy of [3, 1]. This model assumes cooperative behavior: all parties to the contract invoke the functions made available to them, without adversarial strategies or omissions.

Our liquidity analyzer is based on a type system that symbolically tracks the effects of functions on assets using abstract names. The system ensures a correctness property, which guarantees that the inferred (liquidity) type of the final state of a computation soundly over-approximates the actual runtime state. This allows us to conservatively verify liquidity by checking whether, in all feasible computations, the symbolic asset values at termination are zero – thus ensuring that no residual resources remain trapped within the contract.

We identify two liquidity properties. The first one is `h`-*separate liquidity*: *if an asset* `h` *becomes not-empty in a state then there is a continuation where* `h` *is empty in its final state.* While `h`-separate liquidity is satisfactory in contracts where assets are separated (no asset field is moved to another asset field, for instance when pairwise different assets have different categories, *e.g.* euros, cars, houses), it is inadequate in unrestricted contracts that move assets between asset fields. In these cases, the following stronger property is more reasonable: *if an asset becomes not-empty in a state then there is a continuation where all the assets are empty in its final state.*

For the two properties, `h`-separate liquidity and liquidity, we have designed two analysis algorithms with different computational costs and precision accuracies. The computational cost of the less precise algorithm is quadratic with respect to the number of functions. The second algorithm is more precise because, for instance, it accepts contracts that empty assets by means of several function invocations. However, more precision requires more complexity because one has to analyze *computations.* The crucial issue of the analysis is therefore designing a terminating algorithm given that computations may be *infinitely many* because contracts may have cycles. For this reason we restrict to computations whose length is bound by a value (actually we found more reasonable computations where every function can be invoked a bounded number of times). The computational cost of `k`-separate liquidity and liquidity algorithms is higher than the previous case: it is exponential with respect to the number of functions. It is important to note that the algorithms assume all clauses in the contract are reachable. Consequently, as a preliminary step, they verify the absence of unreachable clauses, as discussed in Section 6 (in [10, 25] we circumvent this assumption by restricting to the sub-language without events).

We apply our liquidity analysis algorithm to the `Bike_Rental` contract. In this case, both `wallet`-separate and `bike`-separate liquidity analyses are effective, as the two assets are handled independently and are never intermingled. Focusing on `wallet`-separate liquidity, the algorithm proceeds as follows:

– It first checks whether there exists any clause that may render the `wallet` asset non-empty. This condition is satisfied by the pay function, which increases the balance of wallet.
– Next, the algorithm verifies whether, for every possible execution path following this clause, there exists at least one subsequent clause that empties `wallet`. Since the contract is acyclic, the set of continuations is finite. In

this case, there are two distinct continuations: (1) the invocation of the `end` function and (2) the execution of the scheduled event at line 14.
- In both cases, the analysis confirms that wallet is correctly emptied, as evidenced by the annotations in the liquidity type system.

With respect to `bike`-separate liquidity, the algorithm proceeds as follows:

- By analyzing the liquidity type system, it identifies that the `offer` function initializes (*i.e.*, fills) the `bike` asset.
- It then computes the set of possible continuations following this action. In this case, the continuations are: (1) `pay; end` and (2) `pay; event_14`.
- Both these execution paths results in the `bike` asset being emptied – *cf.* the move operation at line 13. Therefore, the contract is also `bike`-separate liquid.

While the theoretical framework of the liquidity analysis has been fully developed, the prototype implementation is still under development at the time of writing.

8 Related Work

Dwivedi et al. [19] provide a systematic review of smart contract languages, identifying critical features for drafting legally binding digital agreements. Some of these features are present in existing languages such as Solidity [20], Flint [33], Obsidian [9], and Move [8]. For instance, the representation of contracts as finite state machines, caller-based access restrictions (*e.g.*, Solidity modifiers, Flint's capability blocks, Obsidian's typestates), and the treatment of assets as primitive data types are widely adopted [2, 7, 34, 14]. However, these languages are not designed to model legal obligations explicitly and lack direct support for core legal constructs such as agreement, obligation, prohibition, and judicial enforceability.

In contrast, *Stipula* introduces constructs and patterns that establish a clear correspondence between program logic and legal semantics. Its design enables legal concepts like the meeting of the minds, permissions, and time-bound obligations to be expressed directly and unambiguously in code.

Several efforts have focused on transforming legal norms into formal specifications [22, 15, 24], offering semantic representations of rights and duties. However, these approaches often result in high-level specification languages that are difficult to compile into executable code. *Stipula* fills this gap by acting as an intermediate language – bridging the expressiveness of legal specifications with the executability of programming languages like Java or Solidity. In this respect, it shares motivations with Catala [30] and Orlando [4], which formalize specific legal domains through a core calculus with human-readable syntax. Following the idea of legal calculi [5], *Stipula* offers a minimal but expressive operational model, formalized using concurrency theory.

Unlike declarative legal markup languages such as OpenLaw [38], Lexon [29], and Accord [35], which rely on parameterized templates and external smart contracts, *Stipula* uses explicit programming patterns to capture legal constructs programmatically (see [12]). While Lexon attempts to bridge natural language and code via structured English-like syntax, it lacks formal semantics, leaving the behaviour of its contracts entirely to the generated Solidity code. In contrast, *Stipula* provides built-in primitives – such as agreement and asset transfer – that make the contract's behaviour both explicit and verifiable.

Stipula also differs significantly from financial contract domain specific languages like Marlowe[34] and Findel [7], which are built around algebraic combinators tailored to financial logic. Marlowe ensures liveness through mandatory timeouts and default paths, but this pull-based interaction model often results in indirect and complex control flows. *Stipula* enables a more direct and agent-oriented model, where roles, state transitions, and asset flows are clearly attributed, and timeouts or escape clauses are optional and programmable. While Marlowe executes on the Cardano blockchain using slot-based time, similar mechanisms can be adapted to implement *Stipula*'s timed events on blockchain platforms [12].

We argue that legal contracts are inherently more expressive than financial ones, and require a richer programming model. Unlike Marlowe and Findel, where an interpreter evaluates a fixed combinator structure, *Stipula* defines contracts as programs that are compiled into executable artifacts (e.g., Java applications or smart contracts). Its primitives – such as named states, functions, and events – enable fine-grained control over contract behaviour and lifecycle.

Finally, as regards *Stipula* visual editor, we have been inspired by visual programming tools (*e.g.*, Blockly [37], Babbage [32]) and legal markup languages (*e.g.*, SLCML [18]).

9 Conclusion

We have presented the *Stipula* platform, a comprehensive suite of tools designed to help legal practitioners and developers write, test, debug, and manage smart legal contracts effectively. The platform currently includes a *visual code editor* for intuitive contract authoring in *Stipula*, an interpreter for automatic execution, and a set of analyzers for verifying legally relevant properties such as clause unreachability and the absence of frozen assets. Together, these components form a unified toolchain that makes formal methods more accessible to non-experts while retaining technical rigor.

This paper has not included an empirical evaluation of the platform. As an immediate step in future work, we plan to provide benchmarks, scalability studies, and case analyses that extend beyond the illustrative examples available in the repository. In parallel, we aim to conduct user studies and collect evidence of adoption among legal practitioners, thereby assessing the usability and practical value of the editor and toolchain. Another priority is to evaluate the scope limitations of the current analyzers: for instance, the unreachability analysis assumes the absence of nested cycles, while the liquidity analysis requires bounded computations. We intend to demonstrate that these restrictions do not significantly limit applicability in practice.

Beyond these analyses, we are pursuing two broader research directions. The first concerns the evolution of the *Stipula* language itself. Recent work has explored mechanisms to support *amendments*, which capture runtime modifications of contractual obligations (*e.g.*, due to force majeure or hardship), and the *integration of norms*, which reflects the fact that contractual obligations are shaped not only by the clauses explicitly stated by the parties but also by implicit legal provisions that ensure validity and regulate legal effects. These extensions have been studied theoretically in [27, 28]; prototype support for amendments is already available [13], whereas tooling for norm integration remains an open challenge.

The second direction involves the development of advanced verification tools. We are currently investigating *formal verification mechanisms* to check both partial and total correctness [23]. Our approach translates *Stipula* contracts into JML-annotated Java code, which can then be analyzed using deductive verification tools such as KeY [6]. Counterexamples produced during verification can be mapped back to the contractual level, enabling systematic debugging. At present, our verifier supports contracts with cycles without any state in common (disjoint), and we have developed a prototype for a basic correctness analyzer. Ongoing work aims to lift this restriction, thereby extending verification to contracts with overlapping cycles and richer control flow, potentially through interactive theorem proving or hybrid strategies combining automation with user guidance.

In summary, while the current *Stipula* platform already offers a practical toolchain for authoring and analyzing legal contracts, substantial opportunities remain for empirical validation, language extensions, and advanced verification techniques. These lines of research will further strengthen the platform's applicability and reliability, and ultimately contribute to bridging the gap between formal methods and legal practice.

References

1. Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. Verifying liquidity of recursive bitcoin contracts. *Log. Methods Comput. Sci.*, 18(1), 2022.
2. Massimo Bartoletti and Roberto Zunino. Bitml: A calculus for bitcoin smart contracts. In *Proc. of Computer and Communications Security*, CCS '18, pages 83–100, New York, NY, USA, 2018. ACM.
3. Massimo Bartoletti and Roberto Zunino. Verifying liquidity of Bitcoin contracts. In *Proc. 8th International Conference on Principles of Security and Trust*, pages 222–247. Springer International Publishing, 2019.
4. Shrutarshi Basu, Nate Foster, and James Grimmelmann. Property conveyances as a programming language. In *Proc. 2019 ACM SIGPLAN Int. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 128–142, New York, NY, USA, 2019. Association for Computing Machinery.
5. Shrutarshi Basu, Anshuman Mohan, James Grimmelmann, and Nate Foster. Legal calculi. Technical report, ProLaLa 2022 ProLaLa Programming Languages and the

Law, 2022. At https://popl22.sigplan.org/details/prolala-2022-papers/6/Legal-Calculi.

6. Bernhard Beckert, Richard Bubel, Daniel Drodt, Reiner Hähnle, Florian Lanzinger, Wolfram Pfeifer, Mattias Ulbrich, and Alexander Weigl. The Java verification tool KeY: A tutorial. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Proc. 26th Intl. Symp. on Formal Methods, Milan, Italy*, volume 14934 of *LNCS*, pages 597–623, Cham, September 2024. Springer.
7. Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: Secure derivative contracts for ethereum. In *Proc. Financial Cryptography and Data Security - FC 2017*, volume 10323 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 2017.
8. Sam Blackshear and et al. Move: A language with programmable resources. https://developers.diem.com/main/docs/move-paper, 2021.
9. Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Trans. Program. Lang. Syst.*, 42(3), 2020.
10. Silvia Crafa and Cosimo Laneve. Liquidity analysis in resource-aware programming. In *Proc. 18th Int. Conference on Formal Aspects of Component Software FACS 2022*, volume 13712 of *Lecture Notes in Computer Science*, pages 205–221. Springer, 2022.
11. Silvia Crafa and Cosimo Laneve. Programming legal contracts: A beginners guide to *Stipula*. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen, editors, *The Logic of Software. A Tasting Menu of Formal Methods - Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, volume 13360 of *LNCS*, pages 129–146, Cham, 2022. Springer.
12. Silvia Crafa, Cosimo Laneve, Giovanni Sartor, and Adele Veschetti. Pacta sunt servanda: Legal contracts in *Stipula*. *Science of Computer Programming*, 225:102911, 2023.
13. Silvia Crafa, Cosimo Laneve, and Adele Veschetti. Stipula Prototype, July 2022. Available on github: `https://github.com/stipula-language`.
14. Karl Crary and Michael J. Sullivan. Peer-to-peer affine commitment using bitcoin. In *Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 479–488, New York, NY, USA, 2015. Association for Computing Machinery.
15. Joost T. de Kruijff and H. Hans Weigand. Introducing commitruleml for smart contracts. In *Proc. 13th Int. Workshop on Value Modeling and Business Ontologies (VMBO)*, volume 2383. CEUR-WS.org, 2019.
16. Giorgio Delzanno, Cosimo Laneve, Arnaud Sangnier, and Gianluigi Zavattaro. Decidability problems for micro-stipula. In *Proc. 27th Int. Conference COORDINATION 2025*, volume 15731 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2025.
17. Erik Dervishi. *Stipula* workbench: An integrated environment to design, analyze, and execute stipula code. Bachelor's thesis, University of Bologna, 2025.
18. Vimal Dwivedi, Alex Norta, Alexander Wulf, Benjamin Leiding, Sandeep Saxena, and Chibuzor Udokwu. A formal specification smart-contract language for legally binding decentralized autonomous organizations. *IEEE Access*, 9:76069–76082, 2021.
19. Vimal Dwivedi, Vishwajeet Pattanaik, Vipin Deval, Abhishek Dixit, Alex Norta, and Dirk Draheim. Legally enforceable smart-contract languages: A systematic literature review. *ACM Comput. Surv.*, 54(5), jun 2021.

20. Ethereum. Solidity Documentation. https://docs.soliditylang.org/en/v0.8.30/, 2025.
21. Samuele Evangelisti. Analisi di Contratti Legali in *Stipula*. Bachelor's thesis, University of Bologna, 2024.
22. Christopher K. Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *Proc. 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 210–215, 2016.
23. Reiner Hähnle, Cosimo Laneve, and Adele Veschetti. Formal Verification of Legal Contracts: A translation-based approach. In *to appear in Proc. 20th Int. Conference on Integrated Formal Methods (iFM 2025)*, Lecture Notes in Computer Science. Springer, 2025.
24. Xiao He, Bohan Qin, Yan Zhu, Xing Chen, and Yi Liu. Spesc: A specification language for smart contracts. In *Proc. 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 132–137, 2018.
25. Cosimo Laneve. Liquidity analysis in resource-aware programming. *J. Log. Algebraic Methods Program.*, 135:100889:1–18, 2023.
26. Cosimo Laneve. Reachability analysis in Micro-Stipula. In *Proc. 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024*, pages 17:1–17:12. ACM, 2024.
27. Cosimo Laneve, Alessandro Parenti, and Giovanni Sartor. Legal contracts amending with Stipula. In Sung-Shik Jongmans and Antónia Lopes, editors, *Proc. 25th Int. Conf.COORDINATION*, volume 13908 of *LNCS*, pages 253–270, Cham, 2023. Springer.
28. Cosimo Laneve, Alessandro Parenti, and Giovanni Sartor. Integration of Statutory Norms in Computable Contracts. submitted to a journal, September, 2025.
29. Lexon Foundation. Lexon Home Page. http://www.lexon.tech, 2019.
30. Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A programming language for the law. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, August 2021.
31. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
32. Christian Reitwiebner. Babbage - a mechanical smart contract language. At https://medium.com/@chriseth/babbage-a-mechanical-smart-contract-language-5c8329ec5a0e, 2018.
33. Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. Writing safe smart contracts in flint. In *Proc. 2nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, pages 218–Ă§219, New York, USA, 2018. ACM.
34. Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon Thompson. Marlowe: implementing and analysing financial contracts on blockchain. In *Proc. Workshop on Financial Cryptography and Data Security*, pages 496–511. Springer International Publishing, 2020.
35. Open Source Contributors. The Accord Project. https://accordproject.org, 2018.
36. Elia Venturi. Implementazione di un'interfaccia grafica per la scrittura di contratti legali nel linguaggio *Stipula*. Bachelor's thesis, University of Bologna, 2023.
37. Tim Weingaertner, Rahul Rao, Jasmin Ettlin, Patrick Suter, and Philipp Dublanc. Smart contracts using blockly: Representing a purchase agreement using a graphical programming language. In *Proc. 2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 55–64, 2018.
38. Aaron Wright, David Roon, and ConsenSys AG. OpenLaw Web Site. https://www.openlaw.io, 2019.