Computer Law & Security Review 60 (2026) 106223

Contents lists available at ScienceDirect

Computer Law & Security Review: The International Journal of Technology Law and Practice

journal homepage: www.elsevier.com/locate/clsr

Integration of statutory norms in computable contracts☆

Cosimo Laneve[1], Alessandro Parenti[1,*], Giovanni Sartor

University of Bologna, Italy

ARTICLE INFO

Keywords:
Computable contracts
Domain-specific languages
Contract integration
Implied contract terms
Consumer contract law
Programming languages for the law

ABSTRACT

Legal contracts are governed not only by their explicit terms but also by statutory norms, a principle recognized across legal systems. As contracts become computable and executable as code, ensuring compliance with these norms becomes critical. This paper introduces a method for integrating legislative provisions into computable contracts using the *Stipula* language, via a novel `import` construct. We distinguish between `mandatory` and `default` imports to model imperative and optional legal norms, respectively, and define a mechanism to enforce the priorities between these norms and contract's provisions. This approach supports the automated creation of legally compliant contracts and lays the foundation for a broader framework aimed at enhancing the effectiveness of consumer rights through programmable legal tools.

1. Introduction

Upon entering into a contractual agreement, the parties undertake obligations not only as explicitly articulated within the contract but also in accordance with a set of terms that, while not expressly written in the contract, constitute an integral component of the contractual relationship. Every contract operates indeed within a broader legal framework that underpins its validity and delineates its legal effects.

This foundational principle of contract law is shared and recognized across many legal systems, albeit implemented in different forms.[2] Civil law jurisdictions generally refer to this principle as *contract integration*, whereas in common law systems it is encapsulated under the concept of *implied terms*, denoting obligations not expressly stated but considered essential by virtue of statute, judicial precedent, customary practice, or equitable principles.

Computable contracts are software programs executed within computational environments, designed to represent and autonomously enforce specific elements of legal agreements [1]. While the automated execution of such contracts enhances predictability and contributes to the reduction of transaction costs, it simultaneously raises concerns regarding automated operations violating legal norms. Moreover, the absence of standardized or regulated computational infrastructures challenges institutional oversight, rendering it difficult for authorities to intervene in or suspend the execution of such contracts, especially when they are deployed within distributed ledger environments [2,3]. It is therefore imperative to develop methodologies that ensure compliance with statutory provisions from the outset of contract formation [4,5]. Such an approach should encompass, on the one hand, the automated incorporation of mandatory legal norms that may have been omitted by the contracting parties, and on the other, the systematic exclusion of contractual clauses that contravene binding statutory requirements.

This paper proposes a solution based on the the *Stipula* programming language [6–9]. *Stipula* is a domain-specific language whose primitives are meant to explicitly capture the foundational elements of contract law. Its current implementation is available at github.com/stipula-language. It is accompanied by a comprehensive suite of tools supporting the entire contract development process, including a *visual code editor* for intuitive authoring, an interpreter for automatic execution, and a collection of analyzers that verify legally relevant properties such as type correctness, clause reachability, and the absence of frozen assets [10]. These tools are integrated into a unified *workbench*, enabling legal practitioners and software developers to write, test, debug, and manage computable contracts in an effective and efficient manner.

☆ This article is part of a Special issue entitled: 'Legal Systems' published in Computer Law & Security Review: The International Journal of Technology Law and Practice.

* Corresponding author.
E-mail addresses: cosimo.laneve@unibo.it (C. Laneve), alessandro.parenti3@unibo.it (A. Parenti), giovanni.sartor@unibo.it (G. Sartor).

[1] These authors contributed equally to this research.

[2] This principle is stated in many cross-national charters aimed at fixing a common understanding of contract law. For example, it is included within the Principles of European Contract Law under Article 1:103 or in the United Nations Convention on Contracts for the International Sale of Goods in Article 9.

https://doi.org/10.1016/j.clsr.2025.106223

Available online 1 December 2025

To enhance the integration of statutory provisions within computable contracts, we introduce a novel feature within *Stipula*, termed `import`. This construct enables the incorporation of statutory provisions into specific contractual instances. Our construct implements two paradigms of contract integration, shared by both civil and common law systems: *mandatory integration* and *default integration*. The former pertains to the application of statutory provisions from which the contracting parties are not permitted to deviate, whereas the latter encompasses default rules that apply in the absence of contrary stipulations within the contract. We capture them respectively through `mandatory import` and `default import`, each characterized by distinct operational semantics: under `mandatory import`, the imported statutory provisions take precedence over any conflicting clauses within the contract; conversely, under `default import`, contractual clauses override conflicting default rules. Since *Stipula* is equipped with analytical tools capable of detecting legal inconsistencies during the contract drafting phase [11], it is possible to identify and resolve in advance conflicts between imported provisions and contract-specific clauses, thereby supporting the creation of legally compliant contracts.

To showcase the functioning of the new feature we discuss the context of unilateral contract termination under the Italian law and the consumers' right to withdraw under EU law.[3] Using consumers' rights as a use case is motivated by two main considerations. First, consumer contracts are among the most heavily regulated by statutory law,[4] making it essential for the field of computable contracts to develop tools that can accommodate these legal requirements. Second, we contend that computable contracts, by embedding legislative logic and procedures, can enhance the currently limited effectiveness of certain consumer protection provisions.

Building on *Stipula*'s `import` mechanism, this paper additionally sketches the basis of a framework for the use computable contracts to support and enhance the effectiveness of consumer contractual rights. We report some of the current barriers to effective enforcement and preliminarly distinguish different ways of regulating consumer contracts to identify the areas in which computable contracts can offer meaningful advantages and where they would be less effective.

The paper is structured as follows. Section 2 provides a general overview of contract integration mechanisms across different legal systems and introduces the use case of unilateral contract termination. Section 3 describes the current state of the art of languages for computable contracts, as well as of other formal approaches developed in Artificial Intelligence and Law (AI and Law) for the integration of applicable statutory provisions in legal reasoning systems. Within this framework, we then position the *Stipula* project. Section 4 introduces the main primitives and formal syntax of the *Stipula* language to help readers better understand the technical aspects discussed in the paper. Section 5 demonstrates the practical use of the `import` feature in *Stipula* through a running example. Section 6 formalizes the semantics of the `import` construct. Section 7 explores the applicability of computable contracts and of the `import` feature to support consumer contract law to identify promising directions for future work. Finally, Section 8 summarizes the main findings and takeaways of the paper.

2. The role of law in contracts

The role of statutory law in shaping the content of contracts is recognized across different legal systems. For example, in the Italian legal system, Article 1374 of the Civil Code embodies this principle by stating that parties to a contract are bound not only by its explicit terms but also by statutory or customary rules that arise from the conclusion of the contract [12]. A similar provision is found in Article 1194 of the French Code Civil. In common law jurisdictions, the same principle is recognized through the notion of *implied terms*. In English law, courts have long accepted the possibility of implying terms within contracts[5] either *in fact*, to reflect the presumed intentions of the parties, or *in law*, where certain rules are regarded as necessary "incidents" of specific categories of contracts[6] [13]. A similar recognition, although with differences in the specific scope of implication in fact and in law, can be found in the U.S. legal system,[7] where the concept of implied terms is also reflected in the Restatement (Second) of Contracts.[8]

In all these forms, this legal institute designates provisions not expressly set out by the parties yet incorporated into the contract by operation of law, judicial decision, or custom [14]. For clarity, we will refer to this concept as *contract integration* throughout the paper. Moreover, while integration may derive from various sources, this paper focuses specifically on that originating from statutory law.

Across different legal traditions we can identify two general categories of statutory contract integration: *mandatory integration* and *default integration* [13,15,16].

Default integration occurs for gap-filling purposes and to preserve contracts' validity in the context of incomplete contracts [15]. In this case, certain statutory rules apply when parties have not expressly addressed a particular issue in their agreement nor have expressly excluded the application of default statutory provisions. An example is Section 2-305 of the US Uniform Commercial Code, which governs open price terms, allowing a contract to remain valid even if the price has not been expressly agreed upon, provided certain conditions are met.

Mandatory integration, instead, operates when the legislator requires contracts to comply with certain rules which cannot be derogated by parties' will [17]. In the Italian Civil Code, for example, articles 1339 and 1419 refer to this possibility stating that contract provisions contrary to law are automatically replaced by mandatory rules.

To better explain the mechanics of contract integration, consider the legal framework for *unilateral termination* under the Italian legal system. This aspect is regulated by the Italian Civil Code in Art. 1373. It provides, among other things, that unilateral termination shall only affect future performances, therefore leaving untouched those already performed.[9] However, the article's final paragraph states that this applies without prejudice to any contrary agreement.[10] This essentially indicates the default nature of such provision and that any specific agreement between parties contrasting it should prevail.

While the Civil Code applies generally to all contracts, those concluded in a business-to-consumer relationship and unilaterally drafted by a stronger party also trigger the application of the wide corpus of laws on consumer protection. In this context, legislators have intervened to limit the

[3] Directive 2011/83/EU and implemented in Italy as part of the *Codice del Consumo*.
[4] See the EU corpus of norms on consumer protection: https://commission.europa.eu/law/law-topic/consumer-protection-law_en.
[5] See Liverpool City Council v Irwin [1977] AC 239; Scally v Southern Health and Social Services Board [1992] 1 AC 294.
[6] For example, the Sale of Goods Act 1979 provides standard rules applying to commercial contracts.
[7] See Henningsen v. Bloomfield Motors, Inc., 161 A.2d 69 (N.J. 1960).
[8] Sections 204–205.
[9] Art. 1373 par. 2 *Codice Civile*.
[10] Art. 1373 par. 4 *Codice Civile*.

freedom of contract and establish boundaries to protect consumers, by defining prohibited or necessary contents [18,19]. The boundaries defined serve as mandatory pro-consumer arrangements [20] that cannot be derogated by parties unless in a way that is more favorable to the consumer.

In the context of unilateral contract termination, Directive 2011/83/EU on Consumer Rights establishes the Right of Withdrawal, which grants consumers the opportunity to reconsider their decision to conclude a contract within a specified period of time[11] and without incurring any costs. As any EU directive, it has to be implemented through national legislation to have legal force. In Italy, this provision is transposed into national law through Article 52 of the Consumer Code, resulting in its mandatory application to all consumer contracts and its precedence over any conflicting clauses.

In the remaining of the paper we will show how the mechanism of contract integration (and, specifically, the use case just described) can be captured in the *Stipula* domain-specific language and demonstrate how such functionality, where properly applied, can help improve the effectiveness of consumers' rights. However, it is essential to first examine to what extent current computational approaches to law incorporate, or fail to incorporate, such a mechanism. The next section therefore reviews the state of the art in languages for computable contracts and related formal methods, providing the background against which we position our proposal.

3. Background and comparative framework

Languages for computable contracts. The digital representation of legal contracts has been the subject of extensive research, motivated by the dual goals of monitoring and automation [1,21,22]. A substantial line of work stems from the notion of *Ricardian Contracts* [23–26], which link natural language documents to executable code through explicitly declared parameters — that is, contract-specific values that can be instantiated and referenced in both the written text and the associated code. A markup layer mediates this correspondence, ensuring that parameters are consistently bound across the two representations. While influential, this approach has been criticized for its limited ability to capture the deeper semantics of contractual obligations and for difficulties in validation [27].

An alternative direction has been the design of dedicated programming languages for contracts, enabling a fully computable representation. Some of these languages aim for accessibility and intelligibility by relying on controlled natural languages [28–30]. Others emphasize rigorous formal semantics, often grounded in defeasible or deontic logic, to reflect legal reasoning with precision [31–33]. Logic-based approaches have been successfully applied to tasks such as detecting inconsistencies in contract clauses [32–34] and providing formal specifications for integration with business process management tools to monitor contract execution [31]. However, these approaches are generally not designed to realize computable contracts in the sense of [1,35], that is, contracts whose provisions can be directly executed to automate aspects of contractual performance.

A recent contribution in this space is [36], which addresses compliance in data supply chains. Here, obligations arising from data protection law – typically articulated in contractual instruments such as Data Protection Addenda and compliance assessments – are formally represented using the `Symboleo` specification language. The approach enables the generation of smart contract code to model obligations and powers, verify consistency, and monitor execution within a blockchain infrastructure. Nevertheless, compliance in this framework is conceptualized primarily as process-to-contract, rather than as contract-to-law. By contrast, to the best of our knowledge, no existing contract language provides native mechanisms for legal integration in the strict sense, that is, the ability to import statutory provisions into a contract with formally defined priorities between legal rules and contractual clauses, together with an explicit distinction between mandatory and default norms.

Logic-based approaches for conflicting legal sources. Logic-based analyses of conflicts between norms originating from different sources or carrying different levels of stringency have been extensively explored in the field of AI and Law. Various formal approaches have been proposed, including non-monotonic and defeasible reasoning [37], belief revision [38], and argumentation frameworks [39]. A central observation in this body of work is that defeasibility – the idea that a conclusion derivable from a given set of premises may no longer hold when additional premises are introduced – constitutes a fundamental feature of legal reasoning. In legal contexts, a rule may be overridden by an exception, displaced by a superior rule, or superseded by a subsequent one. To address this, a variety of logical systems have been developed that allow conflicting rules to coexist within the same knowledge base. These systems typically rely on priorities and exceptions to resolve conflicts and identify justified conclusions. A notable example is defeasible logic, a rule-based formalism that provides a logic-programming framework for defeasible reasoning and has been successfully applied in many legal domains [40].

Conflicts and priorities among norms have also been addressed through formal argumentation [41–43]. In this perspective, conflicts between rules or legal provisions give rise to conflicts between the arguments in which they appear. Resolution then depends on the argumentation semantics adopted. For instance, the ASPIC framework incorporates both strict and defeasible rules, and allows conflicts among defeasible rules to be settled according to explicit priorities.

For our purposes, approaches that combine multiple legal sources within a single reasoning process are particularly relevant. One prominent example is the work of Sartor and Dung [44], who apply modular argumentation to private international law. In this framework, rules on jurisdiction and choice of law determine which legal system governs a dispute. Each legal system is represented as an independent module, subdivided into components that specify rules on jurisdiction, competence, and applicable law. Conflict-of-law rules may activate submodules corresponding to foreign jurisdictions or international conventions, which are then integrated into the argumentation framework. This modular architecture enables a systematic representation of fragmented and overlapping legal sources without resorting to ad hoc mechanisms for international disputes.

In a related vein, Prakken [45] employs structured argumentation to combine different modes of legal reasoning, which can be dynamically applied to distinct sets of rules. Legal problem-solving often requires integrating factual reasoning with normative reasoning, where both deductive and defeasible methods must be employed in a coordinated fashion.

[11] The minimum period is fourteen days, starting from a moment that varies depending on the type of contract, such as its conclusion, delivery, or other relevant events. See Article 9(2) of Dir. 2011/83/EU.

Table 1
Syntax of *Stipula* with import features (in red).

```
stipula C [import_clause] {
    assets h̄
    fields x̄
    agreement(Ā){
        Ā_1 : x̄_1
        ...          // ⋃_{i∈1..n} Ā_i ⊆ Ā ,  ⋃_{i∈1..n} x̄_i ⊆ x̄ ,  ⋂_{i∈1..n} x̄_i = ∅
        Ā_n : x̄_n
    } ⇒ @Q
    F
}

norm N <_P>(_z)[_r]/_Q/ [constraints φ] [baseline ♭] { F }

import_clause ::=  [mandatory] import N < P̄ >( z̄ )[ r̄ ]/ Q̄′ /
```

Functions	F	$::=$	$\_ \mid \texttt{@Q A : f}(\overline{y})[\overline{k}](E)\{ S\ W \} \Rightarrow \texttt{@Q}'\ F$
Prefixes	P	$::=$	$E \to \texttt{x} \mid E \to \texttt{A} \mid E \multimap \texttt{h}, \texttt{h}' \mid E \multimap \texttt{h}, \texttt{A}$
Statements	S	$::=$	$\_ \mid P\ S \mid \texttt{if}(E)\{ S \}\texttt{else}\{ S \}\ S$
Events	W	$::=$	$\_ \mid E \gg \texttt{@Q}\{ S \} \Rightarrow \texttt{@Q}'\ W$
Expressions	E	$::=$	$v \mid X \mid \texttt{now} \mid E\ \texttt{op}\ E \mid \texttt{uop}\ E$
Values	v	$::=$	$n \mid \texttt{false} \mid \texttt{true} \mid s \mid \mathbb{t}$
Constraints	φ	$::=$	$\texttt{x rop } w \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$
Baselines	$\flat$	$::=$	$\texttt{x} = w \mid \texttt{x} = w, \flat$
Data	w	$::=$	$v \mid \texttt{now} \mid \texttt{now} + k$ (k *is a natural number*)

Positioning. *Stipula* is designed as an imperative framework that simultaneously captures the semantics of contract law and offers a practical, implementable approach [10]. It adopts an object-oriented, imperative programming style, enriched with dedicated primitives that encode legally meaningful concepts such as obligations, permissions, and asset transfers. The present contribution extends this framework by showing how the legal doctrine of contract integration can be embedded directly into the language design, thereby transforming it into an executable mechanism. To the best of our knowledge, no other domain-specific language for computable contracts currently provides such functionality.

Although the execution mechanism itself has not yet been fully detailed, this paper specifies the underlying algorithm with particular emphasis on the resolution of conflicts among imported norms. By contrast, the integration of external rules and the systematic management of normative conflicts are not easily expressible in existing contract languages, which are predominantly rooted in general-purpose imperative paradigms. For instance, the integration of norms is not clear for smart contracts implementations of legal contracts in Solidity[12] on Ethereum or in Java and Go for Hyperledger Fabric[13] [46,47].

4. *Stipula*

Stipula is a domain-specific language designed for modeling legal contracts [6–10]. It is based on primitives that abstractly capture the fundamental elements of contract law. A legal contract is written in *Stipula with import clauses* according to the syntax of Table 1. The parts in red are not in the basic language and define the import mechanisms discussed in Sections 5 and 6. In the syntax, the parts in brackets $[\cdots]$ are optional. *Stipula* uses different types of names to indicate different entities: $\texttt{C}$, $\texttt{C}'$, $\cdots$ denote *contract names*; $\texttt{A}$, $\texttt{A}'$, $\cdots$ denote *parties*; $\texttt{f}$, $\texttt{g}$, $\cdots$ denote *functions*. The language distinguishes between standard parameters, such as the days of validity of a contract or the cost of a good, and entities like currencies, or digital goods, that are linear resources. The former are stored in *fields* that are denoted by names like $\texttt{x}$, $\texttt{y}$, $\cdots$; the latter ones are stored in *assets* denoted by $\texttt{h}$, $\texttt{k}$, $\cdots$. Finally, we will use $\texttt{Q}$, $\texttt{Q}'$, $\cdots$, to range over contract *states*. To simplify the syntax, we often use the vector notation $\overline{\texttt{x}}, \overline{\texttt{h}}, \overline{\texttt{A}}$ to denote possibly empty sequences of elements.

A contract in *Stipula* has a name, which is $\texttt{C}$ in Table 1; its body contains the *assets* and *fields*, the *agreement code*, and the sequence F of functions. Technically, the agreement is the constructor of the contract which captures the "meeting of the minds". This denotes the main requirement of contract formation, namely the mutual assent (or *consensus ad idem*) between the parties, and marks the moment when the contract terms become binding.[14] It specifies the set $\overline{\texttt{A}}$ of involved parties and the terms of the contract — the initial values of a subset of the contract's *fields*. The terms of the contract do not include assets, that are initially empty and that must be explicitly transferred during the execution. The agreement also specifies the initial state of the contract. For example,

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

defines a contract whose parties are `Supplier`, `Buyer` and `PriceProvider`, where `Supplier` and `Buyer` *must initially agree* on the value of the field `price`. The initial state of the contract will be `Start`.

[12] soliditylang.org.
[13] https://hyperledger-fabric.readthedocs.io/en/release-2.5/.
[14] Depending on the legal system, in addition to *consensus ad idem*, the exact terminology and number of additional requirements may vary (e.g., *consideration* in common law, *causa* in the Italian legal system).

Table 2
Digital service contract.

VPN SERVICE AGREEMENT
1. Object: *The Trader grants the Consumer access to its VPN software and related services ("Service") for secure and private internet usage.*
2. Price: *The Service is provided at a fixed price of 10 € per year. This price includes all applicable fees but may be subject to additional taxes depending on your location.*
3. Duration: *This agreement shall remain in effect for a period of twelve (12) months from the date of purchase.*

States and functions. Legal contracts may create, extinguish or regulate the parties' normative positions such as permissions, prohibitions or obligations. *Stipula* uses *states* to model and automatically enforce prohibitions and permissions. In each state only certain *functions* in F may be invoked while others are precluded. Every function defines the caller party `A` and the state `Q` when the invocation is admitted. Function's parameters are split in two lists: the *formal parameters* $\overline{y}$ in brackets and the *asset parameters* $\overline{k}$ in square brackets. The *precondition* E constrains the execution of the body of `f`. Finally the *body* `{` S W `}` ⇛ `@Q'` specifies the *statement part* S, the *event part* W, and the state `Q'` of the contract when the function execution terminates. For example, according to the code

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

the state enabling the invocation of the `update_price` function is `Start`. Through this, the `PriceProvider` records the updated price in the field `price` and sends the new value to the `Supplier` and the `Buyer`. Then the contract transits to the state `Waiting_order`.

Assets and fields. Computable 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 characterized by ad hoc syntactical primitives, thus separating asset types from other data types. Accordingly, the *statements* S in Table 1 include the empty statement `_` and different prefixes followed by a continuation. Prefixes P use the two symbols ⊸ and → to differentiate operations on assets and of fields, respectively. The prefix $E \to$ `x` updates the field or the parameter `x` with the value of E; $E \to$ `A` sends the value of E to the party `A`; $E \multimap$ `h,h'` subtracts the value of E to the asset `h` and adds it to `h'` – *resources stored in assets can be moved but cannot be destroyed* –, $E \multimap$ `h,A` subtracts the value of E to the asset `h` and transfers it to `A`. The operational semantics will prevent assets with negative values. Statements also include a *conditional* `if(`E`){` S `}else{` S' `}` with the standard semantics. We will always abbreviate `h ⊸ h,h'` and `h ⊸ h,A` (which are very usual, indeed) into `h ⊸ h'` and `h ⊸ A`, respectively.

Events. Obligations in *Stipula* are abstractly captured by *Events* W. These are *ad-hoc* operations that monitor whether an obligation is fulfilled at a given time, and, if not, automatically trigger its enforcement or the imposition of a penalty. More precisely, the term $E \gg$ `@Q{`S`}` ⇛ `@Q'` schedules an execution that is triggered k time slots ahead, where k is the value of E. When triggered, the continuation S will be automatically executed if the contract's state is `Q`. At the end of the execution of S, the contract transits to `Q'`. The scheduling of the events is nondeterministic when several events are ready to be executed at the same time. For example, the operation (that occurs inside the body of a function)

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

triggers a transition at time `now + time_due` that may take place if, at that time, the contract is in the state `Inactive`. It informs the `Buyer` that the contract is terminated and transitions the contract to the state `End`.

Expressions E include constant values ranged over by $v, u, \ldots$, names of assets, fields and parameters generically ranged over by X, the keyword `now` representing the current time (when the code is executed), and both standard binary (`op`) and unary (`uop`) operations (arithmetic operations, logical operations, relational operations, etc.). Expressions are also *values*, which include real numbers n, booleans `true` and `false`, strings s that are sequences of characters that are pre- and post-fixed by "", and time values $\mathbb{t}$ that represent the global system clock and are expressed in the format `"2025/1/1:00:15"`, with minute-level precision.

The formal semantics of *Stipula* is reported in the Appendix. We conclude this section by discussing the encoding in *Stipula* of a simple contract for the supply of a `VPN` digital service. The contract is defined in Table 2; it has two parties – the `Trader` and the `Consumer` – and initially sets the basic elements of the relationship, namely the object, the price, and the duration. It is intended that, once the `Consumer` has paid the 10 €, he gets the passkey to access to the `VPN`. We can encode this contract in *Stipula* as follows.

```
stipula VPN {
    fields: tstart, tend, price, product
    asset: wallet, good

    agreement (Trader, Consumer)(){
        Trader, Consumer : tend, price, product
            // price 10 Euro,  tend = 12M, product = VPN
    } ⇛ @Init

        @Init Trader: supply(type)[v] (type == product){
            v ⊸ good
        } ⇛ @Pay

        @Pay Consumer: pay()[w] (w == price) {
            now → tstart
            w ⊸ wallet
            tstart + tend ≫ @Run { wallet ⊸ Trader } ⇛ @End
        } ⇛ @Run
}
```

Listing 1: *Stipula* VPN service Agreement with termination clause

With the function `supply` (line 10), the `Trader` delivers the product by transferring the digital asset. The function `pay()` (line 14) escrows the purchase price and stores it in the contract's `wallet`. The funds remain in escrow until the contract reaches its end, at which point they are released to the `Trader`. This procedure is implemented by the event in line 17. The contract does not specify any cause of termination other than the natural twelve-month term.

5. The import mechanisms in *Stipula*

The extension of ***Stipula***'s syntax for importing statutory provisions is written in red in Table 1. According to the `import_clause`, there are two distinct modes of integration:

- *default import*: statutory provisions that fill contractual gaps without overriding explicit terms agreed upon by the parties;
- *mandatory import*: statutory provisions that override conflicting clauses in a contract, ensuring compliance with non-derogable legal norms.

The clause also specifies the identifier `N` of the norm to be imported, and the mapping that replaces the placeholder names used in the norm with corresponding names defined in the contract.

In turn, a norm defines a number of functions in the same style of ***Stipula*** functions and, possibly, *constraints* and *baseline values* on the fields of the norm. Constraints are conjunctions or disjunctions of simple requirements, such as "the value of a field `x` must be always greater than a lower bound value w". The operations `rop` are the standard relational operators, namely $=$, $\neq$, $>$, $<$, $\geq$ and $\leq$. The baselines define default values for fields if they are not specified in the contract that integrates the norm. Therefore, when present, it is a nonempty sequence of terms $\mathtt{x} = w$. We assume that pairwise different elements in $\mathfrak{b}$ address different fields. Finally, data w are either constant values or terms like `now` or $\mathtt{now} + k$ indicating the current time (when the data is evaluated) or a future time.

The import clause plays a crucial role in the agreement between parties. Suppose the imported norm is associated with a set of constraints φ and a set of baseline values $\mathfrak{b}$. Formally, the baseline values $\mathfrak{b}$ is a *function* from fields' names to value; hence $\mathfrak{b}(\mathtt{x})$ returns the value of `x` in $\mathfrak{b}$ (in Section 6 these functions are called *memories*). If the parties do not specify a value for a field `x` whose baseline value is defined in $\mathfrak{b}$, then the field is initialized to $\mathfrak{b}(\mathtt{x})$. Furthermore, if the import is *mandatory*, and the value of `x` agreed upon by the parties does not satisfy the constraints in φ, the agreement is considered invalid if $\mathtt{x} \notin dom(\mathfrak{b})$, and, as before, the field is initialized to $\mathfrak{b}(\mathtt{x})$ (and we always assume that $\mathfrak{b}(\mathtt{x})$ satisfies φ). No constraint checking is performed in the case of a *default import*.

Regarding functions, let $F_{\mathtt{N}}$ denote the set of functions defined by the norm, and F the set defined by the contract. The import clause modifies the function selection mechanism accordingly. If the import is mandatory, the function is selected from $F_{\mathtt{N}}$. If no function from $F_{\mathtt{N}}$ can be executed – either because none is specified in the current state or their preconditions are not satisfied – then the selection falls back to F. Conversely, if the import is default, the selection process prioritizes F, and only considers $F_{\mathtt{N}}$ if no suitable function is found in F.

We showcase the practical functioning of the *import* through a use-case scenario involving unilateral termination of contract and the consumer's right of withdrawal; the formal semantics of the import is provided in Section 6.

Default import. Take the contract in Listing 1 and consider now Article 1373(2) of the Italian Civil Code, related to Unilateral Termination:

Norm 1. *[..] In contracts for continuous or periodic performance, this option [unilateral termination] may also be exercised subsequently [after performance has begun], but the termination has no effect on performances already performed. [..]*

This provision functions as a default rule within the Italian legal system, meaning it shall apply to all contracts within the scope of the provision, unless expressly excluded by the parties. As such, it applies also to the contract in Table 2. Here, since the price is calculated based on time, the fact that the termination shall not affect previous performances means that if the purchaser terminates the contract early, they would still be required to pay for the portion of time the service has been used.

Statutory norms can be represented in ***Stipula*** using the import mechanism, as code modules designed for later integration into specific contracts. These modules, referred to as `norm`, are therefore parametric. For example, Article 1373(2) is formalized as shown below:

```
1 norm m_1373cc <_trader,_consumer>(_tstart,_tend)[_wallet]/_Qa,_Qb/
2     constraints     _tend>0
3     baseline        _tend=1 {
4
5     @_Qa _consumer: withdraw()[] (now <= _tstart+_tend){
6         ((now-_tstart)/_tend)×_wallet ⊸ _wallet, _trader
7         wallet ⊸ _consumer
8     } ⇛ @_Qb
9 }
```

Listing 2: Norm `m_1373cc`: Article 1373 of the Italian Civil Code

The declaration in line 1 indicates that the norm features two parties, two fields (`_tstart`, storing the contract's starting time, and `_tend`, storing the contract duration), one asset `wallet`, and two states `_Qa` and `_Qb` (to ease the reading, the parameters of norms are always prefixed by "_"). As mentioned above, norms can define boundaries for fields, that is, constraints limiting the range of admissible values a field may assume, and baselines, which assign default values to fields when they are uninitialized or when their assigned values violate the constraints. The effect of these boundaries depends on the type of import (formal semantics is provided in Section 6). For norm `m_1373cc`, `_tend` must be positive; if the agreement phase of the contract does not initialize the field, then, because of the baseline clause, `_tend` is instantiated to 1 (this is a generic positive value to highlight the usefulness of the `baseline` construct). The norm also defines the `withdraw()` function that allows the `_consumer` to terminate the contract (entering state `_Qb`). When this function is called, the contract automatically calculates the amount due to the `_trader` based on the elapsed time and refunds the remaining balance to the `_consumer` (lines 6, 7). For example, if half of the time is elapsed, *e.g.* `now - _tstart = _tend/2`, then line 6 becomes

```
        0.5×_wallet ⊸ _wallet, _trader
```

meaning that the `_trader` gets half of the money and `_consumer` the other half (line 7). This mechanism reflects the provision of Article 1373 c.c., according to which termination shall not affect previous performances.

The integration of norm `m_1373cc` within the contract in Listing 1 is performed by modifying the contract declaration as follows:

```
stipula VPN import m_1373cc <Consumer, Trader>(tstart,tend)[wallet] / Run,End /
```

As a consequence, the module implementing Article 1373 c.c. is automatically incorporated in `VPN`, and `Consumer` will have access to the unilateral withdrawal mechanisms.

Next, assume to extend the `VPN` service agreement in Table 2 in order to include the following unilateral early termination clause:

4(a). Termination and Cancellation Fee: *The Consumer may unilaterally withdraw from this agreement at the latest within seven (7) days from the conclusion of the agreement. In such a case, the Consumer will be refunded of the total price deducted a termination fee of 10%. No withdraw is possible after seven (7) days.*

With this new clause, the *Stipula* representation of the whole agreement would be:

```
1  stipula VPN2 import m_1373cc <Consumer, Trader>(tstart,tend)[wallet]/ Run,End / {
2      fields: tstart, tend, price, twith, product
3      assets: wallet, good
4
5      agreement (Consumer, Trader)(){
6          Consumer, Trader: tend, price, product
7            // tend= 12M, price= 10Euro, product= VPN
8      } ⇒ @Init
9
10         @Init Trader: supply(type)[v] (type == product){
11             v ⊸ good
12         } ⇒ @Pay
13
14         @Pay Consumer: pay()[w] (w == price) {
15             now → tstart
16             w ⊸ wallet
17             tstart + tend ≫ @Run { wallet ⊸ Trader} ⇒ @End
18         } ⇒ @Run
19
20         @Run Consumer: withdraw()[] (now <= tstart+7D) {
21             0.1 × wallet ⊸ wallet, Trader
22             wallet ⊸ Consumer
23         } ⇒ @End
24
25         @Run Consumer: withdraw()[] (now > tstart+7D) {
26         } ⇒ @Run
27 }
```

Listing 3: *Stipula* `VPN` service Agreement with termination clause

To implement clause 4(a), this contract defines a dedicated `withdraw()` function that may be invoked before the deadline. When executed, this function deducts the fee from the contract's wallet (where the escrowed price was stored) prior to issuing a refund. The second `withdraw()` function (lines 27–28) handles invocations that occur after the deadline. In such cases, the function explicitly specifies that the invocation has no effect, thereby ensuring that late withdrawal attempts are safely ignored. The contract also imports the default rule of Article 1373 of the Italian Civil Code, which, as a general provision, applies to contracts in the absence of specific terms. The *Stipula default import* mechanism prioritizes contract-specific provisions over imported defaults. Consequently, the general clause never becomes applicable because of the two `withdraw()` functions at lines 22–25 and 27–28.

Mandatory import. If not individually negotiated, the contract shown in Table 2 qualifies as a consumer contract and is therefore subject to consumer protection regulations — most notably, the right of withdrawal established by EU Directive 2011/83. Consider now Article 52 of the Italian Consumer Code that implements the consumer's right of withdrawal introduced by EU Directive 2011/83, as discussed in Section 2:

Norm 2. *Subject to the exceptions provided for in Article 59, the consumer shall have a period of fourteen days to withdraw from a distance or off-premises contract without giving any reason and without incurring any costs.*

As previously noted, this norm represents a mandatory pro-consumer arrangement [20] that cannot be derogated from, except in ways that are more favorable to the Consumer.[15] The foregoing clause 4(a) clearly conflicts with the right of withdrawal: the deadline granted for withdrawal is too short and it imposes a cancellation fee. In such a situation, despite the specific contract term, Article 52 would prevail over Clause 4(a), and the consumer would retain the right to withdraw without incurring in any charge. To represent this situation, we define the *mandatory import* feature. First of all, we encode the mandatory norm as follows:

```
1  norm m_52cons <_consumer,_trader>(_tstart)[_wallet, _good]/_Qa,_Qb/
2
3      @_Qa _consumer: withdraw()[] (now <= _tstart+14D){
4          _wallet ⊸ _consumer
5      } ⇒ @_Qb
```

Listing 4: Norm `m_52cons`: Article 52 of Italian Consumer Code

[15] Article 25 of Directive 2011/83 and, at the national level, Article 59 of the Italian Consumer Code.

Table 3
Early termination clause 4b.

4(b).	**Termination and Cancellation Fee.** *The Consumer may cancel the subscription with a fee of €2 within thirty (30) days from the original purchase date. After this period, no cancellation is possible anymore.*

In contrast to the norm in Listing 2, the foregoing function `withdraw` includes a fixed constraint that allows its invocation only within 14 days from the conclusion of the agreement (`now <= _tstart + 14D`). If the constraint holds, the entire escrowed amount is returned to the consumer (`_wallet ⊸ _consumer`), reflecting the free-of-charge nature of the right of withdrawal. This norm can be automatically integrated in `VPN2` by replacing lines 1 and 2 as follows:

```
stipula VPN2a mandatory import m_52cons <Consumer, Trader>(tstart,tend)[wallet]/ Run,End /
```

In this case, the imported norm's functions have priority over the contract's functions. Therefore, when `withdraw` is invoked, the runtime will execute the version defined in Listing 4.

Assume now that the `VPN` service agreement includes a different clause regulating unilateral termination – Clause 4(b) – that is defined in Table 3.

In contrast to Clause 4(a), this provision is only partially inconsistent with the consumer's right of withdrawal. While the right must be exercisable free of charge within the initial 14-day period, Article 52 does not prohibit the imposition of cancellation fees once that period has elapsed. This means that after the first fourteen days, the two euros fee imposed by 4(b) is legitimate. This form of partial integration is captured through the *mandatory import* mechanism, as illustrated by the following ***Stipula*** version of the contract incorporating Clause 4(b):

```
stipula VPN3 mandatory import m_52cons
   <Trader, Consumer>(tstart)[wallet]/ Run,End /  {

      fields: tstart, tend, price, product
      assets: wallet, good

      agreement (Trader, Consumer)(){
            Trader, Consumer: tend, price, product
              // tend= 12M, price= 10Euro, product= VPN
      } ⇛ @Init

       ...

      @Run Consumer: withdraw()[] (now <= tstart+30D){
             2 ⊸ wallet, Trader
             wallet ⊸ Consumer
      } ⇛ @End

      @Run Consumer: withdraw()[] (now > tstart+30D){
      } ⇛ @Run
}
```

Listing 5: ***Stipula*** VPN service Agreement with termination clause 4(b)

This version closely mirrors the contract in Listing 3; for this reason, the `supply` and `pay` functions are omitted for brevity. The main difference is that the imported module is of type *mandatory*, corresponding to the statutory right of withdrawal. When the `withdraw` function is invoked, the contract prioritizes the implementation in Listing 4, provided that the constraint `now` $\leq$`tstart` $+$ `14D` is satisfied. After this period, the invocation triggers the `withdraw` function defined in Listing 5, assuming its own constraints hold, and issues a partial refund to the consumer. As previously noted, the *import* mechanism does not remove or override protocols; instead, it defines an execution priority, enabling seamless resolution of partial conflicts between contractual terms and legal provisions.

6. The semantics of the import

The semantics of ***Stipula*** with import clauses is defined as a transition relation between *configurations* (the reader may find the semantics of the basic language in the Appendix). These configurations are tuples $\mathbb{F} \vdash_{\mathtt{t}} \mathtt{Q}, \ell, \Sigma, \Psi$ where

- $\mathbb{F}$ is the declaration part of a contract. It may be either the functions F of a simple contract without any imported norm or $F \blacktriangleright F'$ where, according to the type of import, one of the two are the functions of a norm and the other those of a contract. The operation $F \blacktriangleright F'$ prioritizes the functions in F with respect to those in F';
- $\mathtt{t}$ is the time value of the system's global clock;
- $\mathtt{Q}$ is a state;
- ℓ, called *memory*, is a mapping from names (parties, fields, assets and function's parameters) to values. The values of parties are noted with italic fonts $A, A', \ldots$ (they are initialized when the contract is instantiated);
- Σ is a (possibly empty) residual of function bodies, *i.e.* Σ is either $\_$ – in this case we say Σ *is idle* – or a term $S \Rightarrow \mathtt{Q}$. We assume that $\_\ S \Rightarrow \mathtt{Q}$ is equal to $S \Rightarrow \mathtt{Q}$;
- Ψ is a (possibly empty) multiset of *pending events* that have been already scheduled for future execution but not yet triggered. In particular, Ψ is either $\_$, when there are no pending events, or it is

$$\mathtt{k}_1 \gg \mathtt{Q}_1\{\ S_1\ \} \Rightarrow \mathtt{Q}'_1\ |\cdots|\ \mathtt{k}_h \gg \mathtt{Q}_h\{\ S_h\ \} \Rightarrow \mathtt{Q}'_h$$

where "|" is commutative and associative with identity $\_$. In every term $\mathtt{k}_i \gg \mathtt{Q}_i\{\ S_i\ \} \Rightarrow \mathtt{Q}'_i$, the constant $\mathtt{k}_i$ indicates the time slots that must elapse in order to activate the event.

Table 4
The semantics of the import clause in *Stipula*.

[Agree]

$$\frac{\begin{array}{l}\texttt{stipula C \{ assets } \overline{\texttt{h}} \texttt{ fields } \overline{\texttt{x}} \texttt{ agreement(}\overline{\texttt{A}}\texttt{)\{ d \}} \Rightarrow \texttt{@Q } F \texttt{ \}} \\ \texttt{d} = \overline{\texttt{A}_1} : \overline{\texttt{x}_1} \cdots \overline{\texttt{A}_n} : \overline{\texttt{x}_n} \qquad \ell = [\overline{\texttt{A}} \mapsto \overline{A}, \overline{\texttt{x}_i} \mapsto \overline{v_i}^{\,i\in 1..n}, \overline{\texttt{h}} \mapsto \overline{0}]\end{array}}{\_ \vdash_{\mathbb{t}} \_, \varnothing, \_, \_ \xrightarrow{(\overline{A}, \overline{A_i} : \overline{v_i}^{\,i\in 1..n})} F \vdash_{\mathbb{t}} \texttt{Q}, \ell, \_, \_}$$

[Agree-Mandatory]

$$\frac{\begin{array}{l}\texttt{norm N <}\overline{\texttt{P}}\texttt{>(}\overline{\texttt{z}}\texttt{)[}\overline{\texttt{r}}\texttt{]/}\overline{\texttt{Q}}\texttt{/ constraints } \varphi \texttt{ baseline } \flat \texttt{ \{ } F_{\texttt{N}} \texttt{ \}} \\ \texttt{stipula C mandatory import N < } \overline{\texttt{P}} \texttt{ >(} \overline{\texttt{z}} \texttt{)[} \overline{\texttt{r}} \texttt{]/ } \overline{\texttt{Q}'} \texttt{ / \{} \\ \qquad \texttt{assets } \overline{\texttt{h}} \texttt{ fields } \overline{\texttt{x}} \texttt{ agreement(}\overline{\texttt{A}}\texttt{)\{ d \}} \Rightarrow \texttt{@Q } F \texttt{ \}} \\ \texttt{d} = \overline{\texttt{A}_1} : \overline{\texttt{x}_1} \cdots \overline{\texttt{A}_n} : \overline{\texttt{x}_n} \qquad \ell = [\overline{\texttt{A}} \mapsto \overline{A}, \overline{\texttt{x}_i} \mapsto \overline{v_i}^{\,i\in 1..n}, \overline{\texttt{h}} \mapsto \overline{0}] \\ \ell' = \varphi\{^{\overline{\texttt{z}},\texttt{t}}/_{\overline{\texttt{z}},\texttt{now}}\} \succ_{\texttt{M}} \ell, \flat\{^{\overline{\texttt{z}},\texttt{t}}/_{\overline{\texttt{z}},\texttt{now}}\} \qquad F'_{\texttt{N}} = F_{\texttt{N}}\{^{\overline{\texttt{P}}}/_{\overline{\texttt{P}}}\}\{^{\overline{\texttt{z}}}/_{\overline{\texttt{z}}}\}\{^{\overline{\texttt{r}}}/_{\overline{\texttt{r}}}\}\{^{\overline{\texttt{Q}'}}/_{\overline{\texttt{Q}}}\}\end{array}}{\_ \vdash_{\mathbb{t}} \_, \varnothing, \_, \_ \xrightarrow{(\overline{A}, \overline{A_i} : \overline{v_i}^{\,i\in 1..n})} F'_{\texttt{N}} \blacktriangleright F \vdash_{\mathbb{t}} \texttt{Q}, \ell', \_, \_}$$

[Agree-Default]

$$\frac{\begin{array}{l}\texttt{norm N <}\overline{\texttt{P}}\texttt{>(}\overline{\texttt{z}}\texttt{)[}\overline{\texttt{r}}\texttt{]/}\overline{\texttt{Q}}\texttt{/ constraints } \varphi \texttt{ baseline } \flat \texttt{ \{ } F_{\texttt{N}} \texttt{ \}} \\ \texttt{stipula C import N < } \overline{\texttt{P}} \texttt{ >(} \overline{\texttt{z}} \texttt{)[} \overline{\texttt{r}} \texttt{]/ } \overline{\texttt{Q}'} \texttt{ / \{} \\ \qquad \texttt{assets } \overline{\texttt{h}} \texttt{ fields } \overline{\texttt{x}} \texttt{ agreement(}\overline{\texttt{A}}\texttt{)\{ d \}} \Rightarrow \texttt{@Q } F \texttt{ \}} \\ \texttt{d} = \overline{\texttt{A}_1} : \overline{\texttt{x}_1} \cdots \overline{\texttt{A}_n} : \overline{\texttt{x}_n} \qquad \ell = [\overline{\texttt{A}} \mapsto \overline{A}, \overline{\texttt{x}_i} \mapsto \overline{v_i}^{\,i\in 1..n}, \overline{\texttt{h}} \mapsto \overline{0}] \\ \ell' = \varphi\{^{\overline{\texttt{z}},\texttt{t}}/_{\overline{\texttt{z}},\texttt{now}}\} \succ_{\texttt{D}} \ell, \flat\{^{\overline{\texttt{z}},\texttt{t}}/_{\overline{\texttt{z}},\texttt{now}}\} \qquad F'_{\texttt{N}} = F_{\texttt{N}}\{^{\overline{\texttt{P}}}/_{\overline{\texttt{P}}}\}\{^{\overline{\texttt{z}}}/_{\overline{\texttt{z}}}\}\{^{\overline{\texttt{r}}}/_{\overline{\texttt{r}}}\}\{^{\overline{\texttt{Q}'}}/_{\overline{\texttt{Q}}}\}\end{array}}{\_ \vdash_{\mathbb{t}} \_, \varnothing, \_, \_ \xrightarrow{(\overline{A}, \overline{A_i} : \overline{v_i}^{\,i\in 1..n})} F \blacktriangleright F'_{\texttt{N}} \vdash_{\mathbb{t}} \texttt{Q}, \ell', \_, \_}$$

The transition relation of *Stipula* is $\mathbb{F} \vdash_{\mathbb{t}} \texttt{Q}, \ell, \Sigma, \Psi \xrightarrow{\mu} \mathbb{F} \vdash_{\mathbb{t}'} \texttt{Q}', \ell', \Sigma', \Psi'$, where

$$\mu ::= \quad \_ \quad | \quad (\overline{A}, \overline{A_1} : \overline{v_1}, \ldots, \overline{A_n} : \overline{v_n}) \quad | \quad A : \texttt{f}(\overline{u})[\overline{v}] \quad | \quad v \to A \quad | \quad v \multimap A \,.$$

That is the label μ is either empty, or denotes an initial agreement, or a function call, or a value send, or an asset transfer. The initial state of a contract is

$$\_ \vdash_{\mathbb{t}} \_, \varnothing, \_, \_$$

and the first transition

$$\_ \vdash_{\mathbb{t}} \_, \varnothing, \_, \_ \xrightarrow{(\overline{A}, \overline{A_1} : \overline{v_1}, \ldots, \overline{A_n} : \overline{v_n})} \mathbb{F} \vdash_{\mathbb{t}} \texttt{Q}', \ell', \Sigma', \Psi'$$

defines the effects of the agreement. The formal definition of this transition is given in Table 4 where we use the following *auxiliary functions*:

- we say that φ is *satisfiable* if there exist a substitution $\{^{\overline{v}}/_{\overline{\texttt{x}}}\}$, where $\overline{x}$ are the variables in φ, such that $\varphi\{^{\overline{v}}/_{\overline{\texttt{x}}}\}$ is true;
- $\flat \setminus \texttt{x}$ is the memory $\flat$ that is not defined on the name x, formally:

$$(\flat \setminus \texttt{x})(\texttt{y}) = \begin{cases} \flat(\texttt{y}) & \text{if } \texttt{y} \neq \texttt{x} \text{ and } \texttt{y} \in dom(\flat) \\ undefined & \text{otherwise} \end{cases}$$

- the *mandatory import operation* $\varphi \succ_{\texttt{M}} \ell, \flat$ returns a memory that is defined as follows:

$$\varphi \succ_{\texttt{M}} \ell, \flat \stackrel{\text{def}}{=} \begin{cases} \flat & if\ \ell = \varnothing \text{ and } \varphi\{^{\flat(\overline{\texttt{x}})}/_{\overline{\texttt{x}}}\} \text{ is satisfiable, with } \overline{\texttt{x}} = dom(\flat) \\ \ell'[\texttt{x} \mapsto v] & if\ \ell(\texttt{x}) = v \text{ and } \varphi\{^{v}/_{\texttt{x}}\} \text{ is satisfiable} \\ & \text{and } \ell' = \varphi \succ_{\texttt{M}} \ell \setminus \texttt{x}, \flat \setminus \texttt{x} \\ \ell'[\texttt{x} \mapsto \flat(\texttt{x})] & if\ \ell(\texttt{x}) = v \text{ and } \varphi\{^{v}/_{\texttt{x}}\} \text{ is not satisfiable} \\ & \text{and } \ell' = \varphi \succ_{\texttt{M}} \ell \setminus \texttt{x}, \flat \setminus \texttt{x} \text{ and } \texttt{x} \in dom(\flat) \end{cases}$$

  Notice that $\varphi \succ_{\texttt{M}} \ell, \flat$ is undefined if $\ell = \ell'[\texttt{x} \mapsto v]$ and $\varphi\{^{v}/_{\texttt{x}}\}$ is not satisfiable and $\texttt{x} \notin dom(\flat)$.
- the *default import operation* $\varphi \succ_{\texttt{D}} \ell, \flat$ returns a memory that is defined as follows (we assume that the baseline values in $\flat$ satisfy φ):

$$\varphi \succ_{\texttt{D}} \ell, \flat \stackrel{\text{def}}{=} \begin{cases} \flat & if\ \ell = \varnothing \text{ and } \varphi\{^{\flat(\overline{\texttt{x}})}/_{\overline{\texttt{x}}}\} \text{ is satisfiable, with } \overline{\texttt{x}} = dom(\flat) \\ \ell'[\texttt{x} \mapsto v] & if\ \ell(\texttt{x}) = v \text{ and } \ell' = \varphi \succ_{\texttt{D}} \ell \setminus \texttt{x}, \flat \setminus \texttt{x} \end{cases}$$

  We notice that, unlike the mandatory import operation, the default import operation is also defined when the values in ℓ do not satisfy φ.

Rule [Agree] is the standard agreement in *Stipula*. In this case, no norm is imported and the premises set the parties and the fields to the values instantiated during the agreement, which are reported in the label of the transition. In this case, the set of functions $\mathbb{F}$ is exactly those of the contract, *e.g.* F.

Table 5
The rule for invoking functions in *Stipula* with the import clause.

$$\text{[Function]}\quad \frac{\Psi, \mathtt{Q} \overset{\mathtt{t}}{\nrightarrow} \qquad @\mathtt{Q}\,\mathtt{A} : \mathtt{f}(\overline{\mathtt{y}})\,[\overline{\mathtt{k}}]\;(E)\;\{\,S\;W\,\} \Rightarrow @\mathtt{Q}' \in \mathbb{F}[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}} \qquad \ell(\mathtt{A}) = A \qquad \ell' = \ell[\overline{\mathtt{y}} \mapsto \overline{u}, \overline{\mathtt{k}} \mapsto \overline{v}]}{\mathbb{F} \vdash_{\mathtt{t}} \mathtt{Q}, \ell, \_, \Psi \xrightarrow{A : \mathtt{f}(\overline{u})[\overline{v}]} \mathbb{F} \vdash_{\mathtt{t}} \mathtt{Q}, \ell', S\,W \Rightarrow \mathtt{Q}', \Psi}$$

Rule [Agree-Mandatory] defines the mandatory import of a norm N. In this case, it is necessary to verify that the values agreed upon by the parties satisfy the constraints specified in φ. If this condition holds, the resulting memory ℓ' reflects the agreed values — see the definition of $\varphi\{^{\overline{z},\mathtt{t}}/_{\overline{z},\mathtt{now}}\} \succ_{\mathtt{M}} \ell, \flat\{^{\overline{z},\mathtt{t}}/_{\overline{z},\mathtt{now}}\}$ (notice that now is replaced by the value of the system's global clock). If, instead, there exists a field x whose agreed value violates a constraint in φ, two cases are distinguished: (*i*) the baseline value of x is undefined in $\flat$, or (*ii*) a baseline value for x is defined in $\flat$. In case (*i*), the agreement fails and no transition occurs. In case (*ii*), the memory is updated to store the baseline value $\flat(\mathtt{x})$ in place of the non-compliant value agreed by the parties. Notice that the declaration part of the configuration is $F'_{\mathtt{N}} \blacktriangleright F$, which gives precedence to the functions in the norm ($F'_{\mathtt{N}}$ is the instance of the functions) over those in the contract.

Rule [Agree-Default] models default imports. It differs from [Agree-MImport] in that the function $\varphi\{^{\overline{z}}/_{\overline{z}}\} \succ_{\mathtt{D}} \ell, \flat\{^{\overline{z}}/_{\overline{z}}\}$ is guaranteed to succeed: the constraints φ are not checked, and baseline values are used whenever the parties have not agreed on a specific value. In this case, the declaration component of the configuration is $F \blacktriangleright F'_{\mathtt{N}}$, which gives precedence to the functions defined in the contract over those in the norm.

Once an agreement has been reached – potentially also in compliance with the imported norm – it becomes necessary to define the rule for selecting a function from the available ones. In *Stipula*, this was straightforward: a function was simply chosen from the contract body. In the extended framework presented in this paper, however, the selection must follow the sequence $F \blacktriangleright F'$. In order to define the operational rule, we use the following auxiliary functions:

- the *evaluation function* $[\![E]\!]_\ell$ that returns the value of E in the memory ℓ. In particular:
  - $[\![v]\!]_\ell = v$ for values that are reals, booleans or strings, $[\![V]\!]_\ell = \ell(V)$ for names of assets, fields and parameters;
  - $[\![\mathtt{t}]\!]_\ell$, where $\mathtt{t}$ is a time value like "2025/1/1:00:15", is the difference between $\mathtt{t}$ and the current clock value;
  - let $\underline{uop}$ and $\underline{op}$ be the semantic operations corresponding to uop and op, then $[\![\mathtt{uop}\,E]\!]_\ell = \underline{uop}\ v$, $[\![E\,\mathtt{op}\,E']\!]_\ell = v\ \underline{op}\ v'$ with $[\![E]\!]_\ell = v, [\![E']\!]_\ell = v'$.
- the *selection operation* $\mathbb{F}[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}}$ be

$$\mathbb{F}[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}} = \begin{cases} \left\{ @\mathtt{Q}\,\mathtt{A} : \mathtt{f}(\overline{y})[\overline{k}](E)\{\,S\;W\,\} \Rightarrow @\mathtt{Q}' \;\middle|\; \begin{array}{l} @\mathtt{Q}\,\mathtt{A} : \mathtt{f}(\overline{y})[\overline{k}](E)\{\,S\;W\,\} \Rightarrow @\mathtt{Q}' \text{ in } F \\ \textit{and } [\![E\{^{\mathtt{t}}/_{\mathtt{now}}\}]\!]_{\ell[\overline{y}\mapsto\overline{u},\overline{k}\mapsto\overline{v}]} = \mathtt{true} \end{array} \right\} & \textit{if } \mathbb{F} = F \\ F[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}} & \textit{if } \mathbb{F} = F \blacktriangleright F' \textit{ and } F[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}} \neq \varnothing \\ F'[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}} & \textit{if } \mathbb{F} = F \blacktriangleright F' \textit{ and } F[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}} = \varnothing \end{cases}$$

  That is, the selection $\mathbb{F}[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}}$ returns a *set of functions* in $\mathbb{F}$ such that the corresponding guard E is true;
- the predicate $\Psi, \mathtt{Q} \overset{\mathtt{t}}{\nrightarrow}$, whose definition is

$$\Psi, \mathtt{Q} \overset{\mathtt{t}}{\nrightarrow} \overset{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } \Psi = \_ \\ \mathit{false} & \text{if } \Psi = \mathtt{t} \gg @\mathtt{Q}\{\,S\,\} \Rightarrow @\mathtt{Q}' \mid \Psi' \\ \Psi', \mathtt{Q} \overset{\mathtt{t}}{\nrightarrow} & \text{if } \Psi = \mathtt{t}' \gg @\mathtt{Q}'\{\,S\,\} \Rightarrow @\mathtt{Q}'' \mid \Psi' \text{ and } (\mathtt{t}' \neq \mathtt{t} \text{ or } \mathtt{Q}' \neq \mathtt{Q}) \end{cases}$$

Table 5 defines the rule for selecting a function in *Stipula* with import clauses. The label of the transition specifies the party A performing the invocation, the function name f and the actual parameters. The transition may occur provided (*i*) the contract is in the state Q that admits invocations of f from A, (*ii*) it is *idle* and no pending event can be triggered (in *Stipula* events preempt function invocations) – premise $\Psi, \mathtt{Q} \overset{\mathtt{t}}{\nrightarrow}$ –, and (*iii*) the code $\mathbb{F}$ contains a function $@\mathtt{Q}\,\mathtt{A} : \mathtt{f}(\overline{y})[\overline{k}](E)\{\,S\,\} \Rightarrow @\mathtt{Q}'$ such that E is true in the memory ℓ updated with the actual parameters and with now replaced by the current global clock. We notice that (*iii*) is defined by the selection operation $\mathbb{F}[@\mathtt{Q}\,\mathtt{A} : \mathtt{f}]^{\mathtt{t}}_{\ell,\overline{u},\overline{v}}$.

6.1. The import semantics in action

In order to illustrate the operational behavior of the rules presented in Tables 4 and 5, we apply them to the contracts VPN2 and VPN3 (Listings 3 and 5 respectively), introduced in Section 5. This exercise allows us to verify that the resulting semantics is consistent with, and faithfully captures, the intended meaning previously discussed.

Let $F_{\texttt{VPN2}}$ be the functions in Listing 3 and $F_{\texttt{m1373}}$ be the `withdraw` function in Listing 2. Let also $\overline{A} = \mathit{Consumer}, \mathit{Trader}$ and $\overline{v}$, containing the values of `tstart`, `tend`, `price`, `product`, `fee`, `twith`, is the tuple `0,12M,10,VPN,0.1,7D`; we assume that the initial global clock is 0 and that

$$
\begin{aligned}
F'_{\texttt{m1373}} &= F_{\texttt{m1373}}\{^{\texttt{Consumer,Trader}}/_{\texttt{\_Consumer,\_Trader}}\}\{^{\texttt{tstart,tend,wallet}}/_{\texttt{\_tstart,\_tend,\_wallet}}\}\\
\ell &= [\texttt{Consumer} \mapsto \mathit{Consumer}, \texttt{Trader} \mapsto \mathit{Trader}, \texttt{tstart} \mapsto 0, \texttt{tend} \mapsto \texttt{12M},\\
&\quad \texttt{price} \mapsto 10, \texttt{product} \mapsto \texttt{VPN}, \texttt{fee} \mapsto 0.1, \texttt{twith} \mapsto \texttt{7D}, \texttt{wallet} \mapsto 0, \texttt{good} \mapsto 0]\\
\ell' &= \ell[\texttt{type} \mapsto \texttt{VPN}, \texttt{v} \mapsto \mathit{adr}]\\
\ell'' &= \ell'[\texttt{v} \mapsto \mathit{adr}, \texttt{w} \mapsto 0, \texttt{tstart} \mapsto 0, \texttt{wallet} \mapsto 10]
\end{aligned}
$$

where adr is the internet address of a `VPN` service. A possible computation of `VPN2` is

$$
\begin{aligned}
&\_ \vdash_0 \_, \emptyset, \_, \_\\
&\xrightarrow{\overline{A}, \overline{A}:\overline{v}} && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_0 \texttt{Init}, \ell, \_, \_ && \text{(Agree-DImport)}\\
&\xrightarrow{\mathit{Trader}.\texttt{supply(VPN)}[\mathit{adr}]} && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_0 \texttt{Init}, \ell', S_{\texttt{spl}} \Rightarrow \texttt{Pay}, \_ && \text{(Function)}\\
&\longrightarrow && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_0 \texttt{Init}, \ell'[\texttt{good} \mapsto \mathit{adr}], \_ \Rightarrow \texttt{Pay}, \_ && \\
& && && \text{(Asset-Update), } cf. \text{ Appendix}\\
&\longrightarrow && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_0 \texttt{Pay}, \ell'[\texttt{good} \mapsto \mathit{adr}], \_, \_ && \\
& && && \text{(State-Change), } cf. \text{ Appendix}\\
&\xrightarrow{\mathit{Consumer}.\texttt{pay}()[10]} && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_0 \texttt{Pay}, \ell'[\texttt{v} \mapsto \mathit{adr}, \texttt{w} \mapsto 10], S_{\texttt{pay}} \Rightarrow \texttt{Run}, \_ && \text{(Function)}\\
&\longrightarrow^2 && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_0 \texttt{Pay}, \ell'', EV_{\texttt{pay}} \Rightarrow \texttt{Run}, \_ && \\
& && && \text{(Field-Update), (Asset-Update), } cf. \text{ Appendix}\\
&\longrightarrow && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_0 \texttt{Run}, \ell'', \_, EV_{\texttt{pay}} && \\
& && && \text{(State-Change), } cf. \text{ Appendix}\\
&\longrightarrow^* && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{6D}} \texttt{Run}, \ell'', \_, EV_{\texttt{pay}} && \\
& && && \text{sequence of (Tick), } cf. \text{ Appendix}
\end{aligned}
$$

where $S_{\texttt{spl}}$ and $S_{\texttt{pay}}$ are the bodies of the functions `supply` and `pay` in Listing 3, respectively, and $EV_{\texttt{pay}}$ is the event in $S_{\texttt{pay}}$. Notice that, in the configuration

$$F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{6D}} \texttt{Run}, \ell'', \_, EV_{\texttt{pay}}$$

there are two functions `@Run Consumer:withdraw` that can be invoked – the one in $F_{\texttt{VPN2}}$ and that in $F'_{\texttt{m1373}}$ – because both `now <= tstart+twith` and `now <= tstart+tend` hold. However, because of the definition of $(F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}})$[`@Run Consumer:withdraw`], the function that is returned is the one at lines 22–25 in $F_{\texttt{VPN2}}$ because the import is the default one. Therefore the computation continues as follows ($S_{\texttt{wtd}}$ is the body of `withdraw` at lines 22–25 of Listing 3):

$$
\begin{aligned}
&\xrightarrow{\mathit{Consumer}.\texttt{withdraw}()[]} && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{6D}} \texttt{Run}, \ell'', S_{\texttt{wtd}} \Rightarrow \texttt{End}, EV_{\texttt{pay}} && \text{(Function)}\\
&\xrightarrow{1 \multimap \mathit{Trader}} && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{6D}} \texttt{Run}, \ell''[\texttt{wallet} \mapsto 9], \texttt{wallet} \multimap \texttt{Consumer} \Rightarrow \texttt{End}, EV_{\texttt{pay}} && \\
& && && \text{(Asset-Send), } cf. \text{ Appendix}\\
&\xrightarrow{9 \multimap \mathit{Consumer}} && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{6D}} \texttt{Run}, \ell''[\texttt{wallet} \mapsto 0], \_ \Rightarrow \texttt{End}, EV_{\texttt{pay}} && \\
& && && \text{(Asset-Send), } cf. \text{ Appendix}\\
&\longrightarrow && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{6D}} \texttt{End}, \ell''[\texttt{wallet} \mapsto 0], \_, EV_{\texttt{pay}} && \\
& && && \text{(State-Change), } cf. \text{ Appendix}
\end{aligned}
$$

If, instead of invoking `withdraw`, we allow an additional two days to elapse, the system evolves according to the following transition:

$$
\begin{aligned}
&\longrightarrow && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{8D}} \texttt{End}, \ell''[\texttt{wallet} \mapsto 0], \_, EV_{\texttt{pay}} && \\
& && && \text{(State-Change), } cf. \text{ Appendix}
\end{aligned}
$$

In this configuration the `Consumer` may invoke `withdraw`. Also in this case, there is only one function `@Run Consumer:withdraw` that can be invoked — the one in lines 27–28 in $F_{\texttt{VPN2}}$ (of Listing 3) because the withdrawal deadline is elapsed. Therefore the computation continues as follows:

$$
\begin{aligned}
&\xrightarrow{\mathit{Consumer}.\texttt{withdraw}()[]} && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{8D}} \texttt{Run}, \ell'', \_ \Rightarrow \texttt{Run}, EV_{\texttt{pay}} && \text{(Function)}\\
&\longrightarrow && F_{\texttt{VPN2}} \blacktriangleright F'_{\texttt{m1373}} \vdash_{\texttt{8D}} \texttt{Run}, \ell'', \_, EV_{\texttt{pay}} && \text{(State-Change), } cf. \text{ Appendix}
\end{aligned}
$$

We now discuss `VPN2a` of Listing 5 that, unlike `VPN2`, uses the mandatory import, therefore the `withdraw` function in the norm will have precedence over the norm in the contract. Let us verify this behavior on an example of computation.

The functions in Listing 5 are the same of those in Listing 3, therefore we address them with the notation $F_{\texttt{VPN2}}$; let $F_{\texttt{m52}}$ be the `withdraw` function in Listing 4. Let also $\overline{A}$ and $\overline{v}$ as before and assume again that the initial global clock is 0. We also let

$$F'_{\texttt{m52}} = F_{\texttt{m52}}\{^{\texttt{Consumer,Trader}}/_{\texttt{\_Consumer,\_Trader}}\}\{^{\texttt{tstart,tend,wallet}}/_{\texttt{\_tstart,\_tend,\_wallet}}\}$$

and take the same values for ℓ, ℓ' and ℓ'' as before. The initial transition of the computations of `VPN2a` is

$$\_ \vdash_0 \_, \emptyset, \_, \_ \xrightarrow{\overline{A}, \overline{A}:\overline{v}} F'_{\texttt{m52}} \blacktriangleright F_{\texttt{VPN2}} \vdash_0 \texttt{Init}, \ell, \_, \_ \qquad \text{(Agree-MImport)}$$

where the reader may notice that the resulting $\mathbb{F}$ has the pattern $F'_{\texttt{m52}} \blacktriangleright F_{\texttt{VPN2}}$, meaning that the `withdraw` function in $F'_{\texttt{m52}}$ has precedence over the one in $F_{\texttt{VPN2}}$. Indeed, by repeating the same computation as in the previous case, we obtain the configuration $F'_{\texttt{m52}} \blacktriangleright F_{\texttt{VPN2}} \vdash_{\texttt{6D}} \texttt{Run}, \ell'', \_, EV_{\texttt{pay}}$.

Table 6
The declaration part for sequences of import clauses.

[Simple-M]

$$\texttt{stipula C \{ assets } \overline{\texttt{h}} \quad \texttt{fields } \overline{\texttt{x}} \quad \texttt{agreement}(\overline{\texttt{A}})\{\ \mathbb{d}\ \} \Rrightarrow @\texttt{Q} \quad F\ \texttt{\}}, \ell \Vdash \mathbb{F}, \ell$$

[Mandatory-M]

$$\frac{\begin{array}{c}\texttt{norm N <}\overline{\_\texttt{P}}\texttt{>(}\overline{\_\texttt{z}}\texttt{)[}\overline{\_\texttt{r}}\texttt{]/}\overline{\_\texttt{Q}}\texttt{/ constraints } \varphi \texttt{ baseline } \flat\ \texttt{\{ } F_{\texttt{N}}\ \texttt{\}} \\ \texttt{stipula C } \overline{\texttt{import-clause;}}\texttt{\{}\cdots\texttt{\}}, \ell \Vdash \mathbb{F}, \ell' \\ \ell'' = \varphi\{^{\overline{\texttt{z}},\texttt{t}}/_{\overline{\texttt{z}},\texttt{now}}\} \succ_{\texttt{M}} \ell', \flat\{^{\overline{\texttt{z}},\texttt{t}}/_{\overline{\texttt{z}},\texttt{now}}\} \qquad F'_{\texttt{N}} = F_{\texttt{N}}\{^{\overline{\texttt{P}}}/_{\overline{\texttt{P}}}\}\{^{\overline{\texttt{z}}}/_{\overline{\texttt{z}}}\}\{^{\overline{\texttt{r}}}/_{\overline{\texttt{r}}}\}\{^{\overline{\texttt{Q}'}}/_{\overline{\texttt{Q}}}\}\end{array}}{\texttt{stipula C } \overline{\texttt{import-clause;}}\ \texttt{mandatory import N <}\overline{\texttt{P}}\texttt{>(}\overline{\texttt{z}}\texttt{)[}\overline{\texttt{r}}\texttt{]/}\overline{\texttt{Q}'}\texttt{/ \{}\cdots\texttt{\}}, \ell \Vdash F'_{\texttt{N}} \blacktriangleright \mathbb{F}, \ell''}$$

[Default-M]

$$\frac{\begin{array}{c}\texttt{norm N <}\overline{\_\texttt{P}}\texttt{>(}\overline{\_\texttt{z}}\texttt{)[}\overline{\_\texttt{r}}\texttt{]/}\overline{\_\texttt{Q}}\texttt{/ constraints } \varphi \texttt{ baseline } \flat\ \texttt{\{ } F_{\texttt{N}}\ \texttt{\}} \\ \texttt{stipula C } \overline{\texttt{import-clause;}}\texttt{\{}\cdots\texttt{\}}, \ell \Vdash \mathbb{F}, \ell' \\ \ell'' = \varphi\{^{\overline{\texttt{z}},\texttt{t}}/_{\overline{\texttt{z}},\texttt{now}}\} \succ_{\texttt{D}} \ell', \flat\{^{\overline{\texttt{z}},\texttt{t}}/_{\overline{\texttt{z}},\texttt{now}}\} \qquad F'_{\texttt{N}} = F_{\texttt{N}}\{^{\overline{\texttt{P}}}/_{\overline{\texttt{P}}}\}\{^{\overline{\texttt{z}}}/_{\overline{\texttt{z}}}\}\{^{\overline{\texttt{r}}}/_{\overline{\texttt{r}}}\}\{^{\overline{\texttt{Q}'}}/_{\overline{\texttt{Q}}}\}\end{array}}{\texttt{stipula C } \overline{\texttt{import-clause;}}\ \texttt{import N <}\overline{\texttt{P}}\texttt{>(}\overline{\texttt{z}}\texttt{)[}\overline{\texttt{r}}\texttt{]/}\overline{\texttt{Q}'}\texttt{/ \{}\cdots\texttt{\}}, \ell \Vdash \mathbb{F} \blacktriangleright F'_{\texttt{N}}, \ell''}$$

If we allow an additional two days to elapse, the system evolves by using the rule [Tick] in Appendix in the following configuration

$$F'_{\texttt{m52}} \blacktriangleright F_{\texttt{VPN2}} \vdash_{\texttt{8D}} \texttt{Run}, \ell'', \_, EV_{\texttt{pay}}$$

At this point, if `Consumer` invokes `withdraw`, we have (we recall that the body of `withdraw` at lines 3–5 of Listing 4 is `wallet` $\multimap$ `Consumer`):

$$\begin{array}{ll} & F'_{\texttt{m52}} \blacktriangleright F_{\texttt{VPN2}} \vdash_{\texttt{8D}} \texttt{Run}, \ell'', \_, EV_{\texttt{pay}} \\ \xrightarrow[\textit{Consumer.withdraw()[]}]{} & F'_{\texttt{m52}} \blacktriangleright F_{\texttt{VPN2}} \vdash_{\texttt{6D}} \texttt{Run}, \ell'', \texttt{wallet} \multimap \texttt{Consumer} \Rrightarrow \texttt{End}, EV_{\texttt{pay}} \quad \text{(Function)} \\ \xrightarrow[10\multimap \textit{Consumer}]{} & F'_{\texttt{m52}} \blacktriangleright F_{\texttt{VPN2}} \vdash_{\texttt{6D}} \texttt{Run}, \ell''[\texttt{wallet} \mapsto 0], \_ \Rrightarrow \texttt{End}, EV_{\texttt{pay}} \\ & \qquad \text{(Asset-Send)}, \ cf.\ \text{Appendix} \\ \longrightarrow & F'_{\texttt{m52}} \blacktriangleright F_{\texttt{VPN2}} \vdash_{\texttt{6D}} \texttt{End}, \ell''[\texttt{wallet} \mapsto 0], \_, EV_{\texttt{pay}} \\ & \qquad \text{(State-Change)}, \ cf.\ \text{Appendix} \end{array}$$

that is exactly the behavior discussed in Section 5.

6.2. Conflicts among imports

The extension of *Stipula* presented in Table 1 admits imports of a single norm only. Conflicts between the functions defined in the contract and those provided by the imported norm are resolved through the operation $F \blacktriangleright F'$. This restriction is introduced for the sake of simplicity, so as to isolate and clarify the fundamental aspects of the import mechanism in a basic setting, before considering more general cases involving multiple norms. In fact, it turns out that the formalism introduced in Section 6 is also adequate for modeling *sequences of imports* defined by the syntax

$$\texttt{stipula C } \overline{\texttt{import\_clause;}}\ \texttt{\{}\cdots\texttt{\}}$$

where $\overline{\texttt{import\_clause;}}$ is a sequence of import clauses (when the sequence is empty we have a standard *Stipula* contract without imports of norms). That is, a generic contract may be

$$\texttt{stipula C import\_clause}_1\texttt{;}\cdots\texttt{;import\_clause}_n\texttt{;\{}\cdots\texttt{\}}$$

In this case, the declaration part $\mathbb{F}$ is defined by the operation

$$\texttt{stipula C } \overline{\texttt{import\_clause;}}\ \texttt{\{}\cdots\texttt{\}}, \ell \Vdash \mathbb{F}, \ell'$$

that takes a term `stipula C` $\overline{\texttt{import\_clause;}}$ `{` $\cdots$ `}` and a memory ℓ and returns a sequence of function declarations $\mathbb{F}$ and a memory ℓ'. The operation is specified in Table 6.

For example, if F are the functions in the contract `C` and F_1 and F_2 are the functions in the norms $\texttt{N}_1$ and $\texttt{N}_2$ after the renaming as in Table 6, respectively, then

- `stipula C mandatory import` $\texttt{N}_1 \cdots$ `; mandatory import` $\texttt{N}_2 \cdots$ `;` $\{\cdots\}, \ell \Vdash F_2 \blacktriangleright F_1 \blacktriangleright F, \ell'$, which highlights the precedence order F_2 then F_1 and finally F;
- `stipula C mandatory import` $\texttt{N}_1 \cdots$ `; import` $\texttt{N}_2 \cdots$ `;` $\{\cdots\}, \ell \Vdash F_1 \blacktriangleright F \blacktriangleright F_2, \ell'$, which specifies that the functions with lower precedence are those in F_2.

(with suitable ℓ and ℓ').

Using the operation `stipula C` $\overline{\texttt{import\_clause;}}$ `{` $\cdots$ `}`$, \ell \Vdash \mathbb{F}, \ell'$ we can collapse the three agreement rules in Table 4 into the rule

[Agree-Multiple]

$$\frac{\texttt{stipula C } \overline{\texttt{import\_clause;}}\ \texttt{\{ assets } \overline{\texttt{h}} \quad \texttt{fields } \overline{\texttt{x}} \quad \texttt{agreement}(\overline{\texttt{A}})\{\ \mathbb{d}\ \} \Rrightarrow @\texttt{Q} \quad F\ \texttt{\}}, \ell \Vdash \mathbb{F}, \ell' \qquad \mathbb{d} = \overline{\texttt{A}_1 : \texttt{x}_1} \ \cdots\ \overline{\texttt{A}_n : \texttt{x}_n} \qquad \ell = [\overline{\texttt{A}} \mapsto \overline{A}, \overline{\texttt{x}_i} \mapsto \overline{v_i}^{\,i\in 1..n}, \overline{\texttt{h}} \mapsto \overline{0}]}{\_ \vdash_{\texttt{t}} \_, \varnothing, \_, \_ \xrightarrow{(\overline{A}, \overline{A_i : v_i}^{\,i\in 1..n})} \mathbb{F} \vdash_{\texttt{t}} \texttt{Q}, \ell, \_, \_}$$

We conclude by observing that the *import* feature enables the attachment of protocol fragments to specific *Stipula* contracts, in a manner analogous to code libraries in programming languages. Depending on the type of import, it also establishes a priority order for function invocations. Our design facilitates the resolution of inconsistencies between contractual provisions and legal norms — not by removing or overwriting protocols, but by specifying a resolution order. This approach is particularly effective when conflicts involve only portions of the provisions, as illustrated in the use-cases.

6.3. Normative conflicts without a priority order

It is important to note that our framework imposes a strict total order (of imports) over the modules to be imported. This ensures that potential conflicts are resolved in a straightforward and deterministic manner. Such an approach results effective not only for solving conflicts among contract clauses and statutory provisions, but potentially also among norms imported through different types of import as seen in above.

In case, however, the conflicting norms are imported through the same type – i.e., both *mandatory* or *default* –, the resolution of conflicts requires a specific consideration. If a hierarchy among these norms is available – e.g., applying the principle *lex posterior derogat priori* – then conflicts can be resolved directly using the semantics in Section 6. The order of imports will reflect the legislative hierarchy and, accordingly, the higher module in the order will prevail over the lower one.

However, in certain cases, the hierarchy of norms imported may not be well established, and require legal interpretation. To extend our framework over cases in which no total order is available in advance, we shall provide ways to allow such an order to be imposed at runtime, by calling an external agent. The external reasoning for defining the order could be delegated to either a human agent, or to one of the legal reasoning models developed within the AI and Law domain rooted in formal logic and argumentation [44,45]. The integration of a similar feature, however, falls outside the scope of this work and is left as a direction for future research.

7. Future perspective for computable contracts in consumer protection

Stipula `Import` enables the drafting of computable agreements in compliance with specific statutory laws. In the context of consumer contract law, some authors [48] have argued that provisions establishing simple and time-sensitive rights, such as the right to withdraw, are well-suited for implementation through computable contracts running on a distributed ledger technology.

Given these premises, we argue that properly designed and implemented computable contracts – both within and outside distributed ledger environments – can enhance consumer protection by improving the effectiveness and enforceability of certain consumer protection rules.

Notwithstanding the legislative efforts, over the past two decades, to define mandatory contractual rights for consumers, such rights still remain largely ineffective [49]. One of the reasons for this has traditionally been traced back to consumers' lack of awareness about their rights or the steps to exercise them, together with their reluctance to engage in complex, time-consuming legal procedures [50,51]. The EU Fitness Check on Digital Fairness from 2024 showed that approximately 60% of consumers had not been able to use consumer law to ensure respect for their rights in the digital context [52]. It was also found that online consumers find it difficult to exercise their rights because of the intentionally complicated interfaces created by traders [52,53]. In such context, it is argued that technology advancements should play a proactive role in enhancing consumer protection [54]. Computable contracts, in particular, could lower transaction costs linked to the claim and enforcement of contract transactions in general and of consumers' prerogatives [35,55,56]. According to [48], computable contracts could be designed in such a way that they are compliant with the law, and do not pose risks for consumers. In particular, it is argued that simple, time-sensitive rights are particularly well-suited for implementation through such tools, due to the technology's capacity to enforce temporal constraints and the minimal interpretative effort required for their application. By contrast, the implementation of more complex rights and long-term contractual relationships poses greater challenges.

Building on this framework, we propose a further distinction that can help to assess the suitability of computable contracts in implementing consumer protection rules. In particular, some rules operate with a *subtractive effect* on contracts' content, while others have an *additive effect*. This distinction provides a lens to evaluate where the *Stipula* `import` mechanism can be most effectively applied.

Subtractive effect. An example of legislation having a subtractive effect is the Unfair Contract Terms Directive,[16] which established the invalidity of clauses not individually negotiated and that create a significant imbalance in the parties' rights and obligations at the disadvantage of the consumer, thereby leaving a gap in the contract. A similar effect cannot be effectively implemented through the `import` feature, as this is about attaching additional protocol to a contract. Theoretically, it could be implemented by identifying specific code patterns of unfairness, in the same fashion of inconsistency patterns in [11], and analyzing contract protocol against these patterns.

However, unfairness of a term is typically defined in relation to the contract context and overall balance between the parties. As a result, while a single pattern may capture one instance of an unfair clause, the same unfair effect can be achieved through multiple alternative formulations. This makes it difficult to identify and encode all possible unfair patterns. We believe that such a task would be carried out more effectively through the implementation of machine learning models (e.g., [57]) that are able to link patterns to an outcome more efficiently and taking context into consideration.

Additive effect. Examples of consumer contract law provisions having an *additive* effect are instead represented by the provisions of the Consumers' Rights Directive[17] and those imposing mandatory remedies for non-conformity of goods.[18]

The first directive imposes an obligation on traders to inform consumers about various aspects of the transaction before they can be legally bound[19] and defines the *right to withdraw*, which constitutes the object of the running example presented in Section 5. The latter ones grant consumers a set of mandatory remedies in case of non-conformity of the goods or services purchased, and defines a specific hierarchy among

[16] Directive 93/13/EC.
[17] Directive 2011/83/EU.
[18] Dir. 771/2019/EU on the sale of goods; Dir. 770/2019/EU on the sale of digital content and services.
[19] Arts 5 and 6 Dir. 2011/83/EU.

them [58] (e.g., a price reduction or termination can be demanded only if a prior attempt to seek repair or replacement has failed, been refused, or not completed within a reasonable time).

In sum, both legislations provide for some mandatory "procedures" to be implemented in contract relationships. The running example in Section 5 shows how the procedure related to the right to withdraw can be effectively formalized as computable contract code and integrated within specific agreements using *Stipula* `import` and we argue that a similar effectiveness can be achieved for other procedural norms provided by consumer contract law, such as the remedies for non-conformity.

Embedding such procedures into computable protocols could enhance consumers' ability to claim their rights, by guiding them through the available solutions without requiring them to have specific knowledge of the law and, when the contract concerns digital goods or services, support a degree of self-enforcement of such procedures, by adjudicating refunds, preclude certain actions, or returning goods. This procedural support would reduce the obstacles to the effectiveness of consumer contract prerogatives and increases the enforceability [52]

In light of these considerations, future research will explore the formalization and encoding of additional mandatory procedures in consumer contract law, with the aim of building comprehensive *Stipula* libraries that implement consumer protection rules and support the design of compliant-by-design contracts.

8. Conclusions

This paper introduced a novel approach to integrating statutory norms into computable contracts through the import feature of the *Stipula* domain-specific language. The feature supports two operational modes – `mandatory import` and `default import` – that correspond to the widely recognized legal distinction between mandatory rules (from which parties cannot deviate) and default rules (which apply in the absence of contrary agreement). This dual-mode design enables a more faithful representation of legal norms within computational contract logic.

We demonstrated the practical utility of this feature through a running example concerning the right of unilateral termination, grounded in the Italian contract law and EU consumer protection rules. Moreover, we formalized the semantics of `import`, highlighting how it resolves conflicts between contractual clauses and imported statutory rules in a legally consistent manner. Specifically, we showed how the mechanism can represent different interactions between statutory and contractual provisions: cases where statutory law fills contractual gaps, where it overrides conflicting contract clauses, and where contractual terms prevail either because statutory rules are only default or because the contract grants the consumer stronger protection. Future works shall deal with the possibility, already available through *Stipula* `import` but not showcased in the present contributions, of importing multiple norms at the same time and identify how to manage priorities among them.

Finally, we identified specific consumer contract provisions that are particularly well suited for implementation through the Stipula `import` feature. We argued that their computable implementation could enhance both the effectiveness and the enforceability of consumer rights. In particular, provisions with an *additive* impact on the content of contracts – especially those establishing mandatory procedures, such as the right of withdrawal or the remedies for non-conforming goods – stand out as the ones that could benefit the most from integration into computable contracts.

In this context, future developments will focus on specifying further the framework for the use of computable contracts to support consumer rights protection and, based on this, on expanding the repertoire of modeled statutory law provisions.

Declaration of competing interest

The author declares that there is no conflict of interest regarding the publication of this paper. The research was conducted independently, without any commercial or financial relationships that could be construed as a potential conflict of interest. This work contributes to the thematic focus of the Computer Law and Security Review special issue on Compliance, Processes, and AI Technologies for Legal Systems by proposing a methodology for the automatic integration of statutory legal norms – specifically consumer protection provisions – into computable contracts using the Stipula domain-specific language. It explores how formal logic and programmable semantics can enhance regulatory compliance and improve the effectiveness of legal safeguards in digital contracting environments.

Acknowledgment

This research was funded by the European Research Council (ERC) Project "CompuLaw" (Grant Agreement No 833647) under the European Union's Horizon 2020 research and innovation program.

Appendix. The operational semantics of *Stipula*

Table A.7 reports the definition of the transition relation of *Stipula* (the language in black of Table 1). In the table we use the *evaluation function* $[\![E]\!]_\ell$ as defined in Section 6.

As regards statements, the rule [Asset-Send] transfers part of an asset $\mathtt{h}$ to the party A. This part corresponds to the value of $[\![E]\!]_\ell$; we remark that the premise $0 \leq [\![\mathtt{h} - E]\!]_\ell \leq [\![\mathtt{h}]\!]_\ell$ constrains the value of E to be positive and smaller or equal to the value of $\mathtt{h}$. For example, if $\mathtt{h}$ holds $\mathtt{20D}$ then $\mathtt{25D} \multimap \mathtt{h}, \mathtt{A}$ would not be executed, while $\mathtt{h} \times 0.5 \multimap \mathtt{h}, \mathtt{A}$ would remove $\mathtt{10D}$ from $\mathtt{h}$ and send them to $\mathtt{A}$. In a similar way, [Asset_Update] moves a part a of an asset $\mathtt{h}$ to an asset $\mathtt{h}'$. When assets are tokens, the premise of the rule [Asset-Send] is well defined only when $\mathtt{h}$ is moved as a whole, that is when the statement under evaluation is $\mathtt{h} \multimap \mathtt{h}, \mathtt{A}$ (that we abbreviate $\mathtt{h} \multimap \mathtt{A}$). Similarly, the premises of the rule [Asset-Update] are well defined only when a is the token in $\mathtt{h}$ and $\mathtt{h}'$ is empty. For example, when the statement is $\mathtt{h} \multimap \mathtt{h}, \mathtt{h}'$ (and $\mathtt{h}'$ is empty). Rule [Value-Send] defines the sending of a value to a party. The value has to be either of type `real` or `string` or `bool`. The statement $\mathtt{1234D} \to A$ is admitted: it means that A will receive the `real` value `1234` (*c.f.* the meaning of $[\![E]\!]_\ell$). Rule [Field-Update] defines updates of fields.

Rule [State-Change] says that a contract changes state when the execution of the statement in the function's body terminates. At this stage, the sequence of events W is added to the multiset of pending events once their time expressions have been evaluated (`now` is replaced by the current value of the clock).

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

Rule [Tick] defines the elapsing of time. This happens when the contract has no statement to perform (*i.e.* no function or event to conclude), and no event can be triggered. As a consequence, the complete execution of a function or of an event cannot last more than a single time unit.

Table A.7
The transition relation of *Stipula*.

$$\text{[Value-Send]}\quad \frac{[\![E]\!]_\ell = v \quad \ell(\mathtt{A}) = A}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, E \to \mathtt{A}\ \Sigma, \Psi \xrightarrow{v \to A} \mathbb{F} \vdash_t \mathtt{Q}, \ell, \Sigma, \Psi}$$

$$\text{[Field-Update]}\quad \frac{[\![E]\!]_\ell = v \quad \ell' = \ell[x \mapsto v]}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, E \to x\ \Sigma, \Psi \longrightarrow \mathbb{F} \vdash_t \mathtt{Q}, \ell', \Sigma, \Psi}$$

$$\text{[Asset-Send]}\quad \frac{\ell(\mathtt{A}) = A \quad 0 \le [\![\mathtt{h} - E]\!]_\ell \le [\![\mathtt{h}]\!]_\ell \quad u = [\![\mathtt{h} - E]\!]_\ell \quad v = [\![\mathtt{h} - u]\!]_\ell}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, E \multimap \mathtt{h}, \mathtt{A}\ \Sigma, \Psi \xrightarrow{u \multimap A} \mathbb{F} \vdash_t \mathtt{Q}, \ell[\mathtt{h} \mapsto v], \Sigma, \Psi}$$

$$\text{[Asset-Update]}\quad \frac{0 \le [\![\mathtt{h} - E]\!]_\ell \le [\![\mathtt{h}]\!]_\ell \quad [\![\mathtt{h} - E]\!]_\ell = u \quad [\![\mathtt{h}' + E]\!]_\ell = u'}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, E \multimap \mathtt{h}, \mathtt{h}'\ \Sigma, \Psi \longrightarrow \mathbb{F} \vdash_t \mathtt{Q}, \ell[\mathtt{h} \mapsto u, \mathtt{h}' \mapsto u'], \Sigma, \Psi}$$

$$\text{[Cond-true]}\quad \frac{[\![E]\!]_\ell = \mathtt{true}}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, \mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}\ \Sigma, \Psi \longrightarrow \mathbb{F} \vdash_t \mathtt{Q}, \ell, S\ \Sigma, \Psi}$$

$$\text{[Cond-false]}\quad \frac{[\![E]\!]_\ell = \mathtt{false}}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, \mathtt{if}\,(E)\,\{\,S\,\}\,\mathtt{else}\,\{\,S'\,\}\ \Sigma, \Psi \longrightarrow \mathbb{F} \vdash_t \mathtt{Q}, \ell, S'\ \Sigma, \Psi}$$

$$\text{[State-Change]}\quad \frac{[\![W\{{}^{\mathtt{t}}/_{\mathtt{now}}\}]\!]_\ell = \Psi'}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, \_\ W \Rrightarrow \mathtt{Q}', \Psi \longrightarrow \mathbb{F} \vdash_t \mathtt{Q}', \ell, \_, \Psi' \mid \Psi}$$

$$\text{[Event-Match]}\quad \frac{\Psi = \mathtt{t} \gg @\mathtt{Q}\ \{\ S\ \} \Rrightarrow @\mathtt{Q}' \mid \Psi'}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, \_, \Psi \longrightarrow \mathbb{F} \vdash_t \mathtt{Q}, \ell, S \Rrightarrow \mathtt{Q}', \Psi'}$$

$$\text{[Tick]}\quad \frac{\Psi, \mathtt{Q} \overset{\mathtt{t}}{\not\to}}{\mathbb{F} \vdash_t \mathtt{Q}, \ell, \_, \Psi \longrightarrow \mathbb{F} \vdash_{t+1} \mathtt{Q}, \ell, \_, \Psi}$$

Data availability

Data will be made available on request.

References

[1] Surden H. Computable contracts. UCDL Rev 2012;46:629.
[2] Stazi A. Protection of weaker parties in smart contracts. J Eur Consum Mark Law 2023;12(6):265–6.
[3] Mik E. Smart contracts: terminology, technical limitations and real world complexity. Law Innov Technol 2017;9(2):269–300.
[4] ELI guiding principles and model rules on algorithmic contracts. Tech. rep., European Law Institute; 2023, URL https://www.europeanlawinstitute.eu/projects-publications/current-projects/current-projects/eli-guiding-principles-and-model-rules-on-algorithmic-contracts/. [Accessed 29 April 2025].
[5] Maugeri M. Smart contracts, consumer protection, and competing European narratives of private law. Ger Law J 2022;23(6):900–9.
[6] Crafa S, Laneve C. Programming legal contracts - A beginners guide to Stipula. In: The logic of software. a tasting menu of formal methods - essays dedicated to Reiner Hähnle on the occasion of his 60th birthday. Lecture notes in computer science, vol. 13360, Cham, Switzerland: Springer; 2022, p. 129–46.
[7] Crafa S, Laneve C, Sartor G, Veschetti A. Pacta sunt servanda: legal contracts in Stipula. Sci Comput Program 2023;225:102911.
[8] Laneve C, Parenti A, Sartor G. Legal contracts amending with Stipula. In: Proceedings of coordination'23. Lecture notes in computer science, vol. 13908, Springer; 2023, p. 253–70.
[9] Laneve C, Parenti A, Sartor G. Programming contract amending. In: JSAI international symposium on artificial intelligence. Springer; 2023, p. 19–34.
[10] The stipula project, Available on github: URL https://github.com/stipula-language.
[11] Laneve C, Parenti A, Sartor G. Draft better contracts. In: Legal knowledge and information systems. IOS Press; 2024, p. 95–106.
[12] Rodotà S. Le fonti di integrazione del contratto, vol. 20, Roma TrE-Press; 2024.
[13] Collins H. Implied terms: The foundation in good faith and fair dealing. Curr Leg Probl 2014;67(1):297–331.
[14] Sprague S, Steadman J. Codice civile in inglese. Wolters Kluwer Italia; 2021.
[15] Cohen GM. Implied terms and interpretation in contract law. Encycl Law Econ 2000;3:78–99.
[16] Bongiovanni V, et al. Integrazione del contratto e clausole implicite. Giuffrè; 2018.
[17] Iamiceli P, et al. Nullità parziale e integrazione del contratto nel diritto dei consumatori tra integrazione cogente, nullità 'nude'e principi di effettività, proporzionalità e dissuasività delle tutele. Giust Civ 2020;2020(4):713–53.
[18] Reich N, Micklitz H-W, Rott P, Tonner K. European consumer law. Intersentia; 2014.
[19] Weatherill S. EU consumer law and policy. In: EU consumer law and policy. Edward Elgar Publishing; 2005.
[20] Ben-Shahar O, Bar-Gill O. Regulatory techniques in consumer protection: a critique of European consumer contract law. Common Mark Law Rev 2013;50(Special).
[21] Milosevic Z, Gibson S, Linington P, Cole J, Kulkarni S. On design and implementation of a contract monitoring facility. In: Proceedings. First IEEE international workshop on electronic contracting, 2004.. 2004, p. 62–70.
[22] Governatori G. Representing business contracts in RuleML. Int J Coop Inf Syst 2005;14(02n03):181–216.
[23] Grigg I. The ricardian contract. In: Proceedings. First IEEE international workshop on electronic contracting, 2004. IEEE; 2004, p. 25–31.
[24] Clack CD, Bakshi VA, Braine L. Smart contract templates: foundations, design landscape and research directions. 2016, CoRR arXiv:1608.00771 URL http://arxiv.org/abs/1608.00771.
[25] Clack CD. Smart contract templates: Legal semantics and code validation. J Digit Bank 2018;2(4):338–52.
[26] Cervone L, Palmirani M, Vitali F, et al. The intelligible contract. In: HICSS. 2020, p. 1–10.
[27] Clack CD. Languages for smart and computable contracts. In: smart legal contracts: computable law in theory and practice. Oxford University Press; 2022, http://dx.doi.org/10.1093/oso/9780192858467.003.0013.
[28] Kowalski R, Datoo A. Logical English meets legal English for swaps and derivatives. Artif Intell Law 2022;30(2):163–97.
[29] Watt SJ, Goodenough O, Wong MW. Deontics and time in contracts: An executable semantics for the L4 DSL. In: Legal knowledge and information systems. IOS Press; 2023, p. 119–24.
[30] Lexon language. 2022, URL http://lexon.org/. [Accessed 13 September 2025].

[31] Governatori G, Milosevic Z, Sadiq S. Compliance checking between business processes and business contracts. In: 2006 10th IEEE international enterprise distributed object computing conference. EDOC'06, IEEE; 2006, p. 221–32.
[32] Parvizimosaed A, Sharifi S, Amyot D, Logrippo L, Roveri M, Rasti A, Roudak A, Mylopoulos J. Specification and analysis of legal contracts with Symboleo. Softw Syst Model 2022;21(6):2395–427.
[33] Prisacariu C, Schneider G. A formal language for electronic contracts. In: International conference on formal methods for open object-based distributed systems. Springer; 2007, p. 174–89.
[34] Fenech S, Pace GJ, Schneider G. Automatic conflict detection on contracts. In: International colloquium on theoretical aspects of computing. Springer; 2009, p. 200–14.
[35] Cummins J, Clack CD. Transforming commercial contracts through computable contracting. J Strat Contract Negot 2022;6(1):3–25.
[36] Baquero PM, Amariles DR, Amyot D, Anda AA, Bayirli M, Logrippo L, Lopes A, Mylopoulos J, Parvizimosaed A, Rasti A, et al. The compliance gap in data supply chains: contract specification languages and smart contracts as compliance technologies: P. Baquero et al.. Artif Intell Law 2025;1–50.
[37] Sartor G. Defeasibility in law. In: Handbook of legal reasoning and argumentation. Springer; 2018, p. 315–64.
[38] Boella G, Pigozzi G, van der Torre L. AGM contraction and revision of rules. J Log Lang Inf 2016;25:273–97.
[39] Sartor G. Argumentation in AI and law. In: Elgar research handbook on legal argumentation. Elgar; 2025.
[40] Governatori G, Rotolo A, Sartor G. Logic and the law: Philosophical foundations, deontics, and defeasible reasoning. In: Handbook of deontic logic and normative systems, volume 2. College Publications; 2022, p. 657–764.
[41] Dung PM. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming, and *n*–person games. Artificial Intelligence 1995;77:321–57.
[42] Prakken H, Sartor G. A dialectical model of assessing conflicting arguments in legal reasoning. Artif Intell Law 1996;4:331–68.
[43] Prakken H. An abstract framework for argumentation with structured arguments. Argument Comput 2010;1:93–124.
[44] Dung PM, Sartor G. The modular logic of private international law. Artif Intell Law 2011;19(2):233.
[45] Prakken H, Sartor G. A formal framework for combining legal reasoning methods. In: Proceedings of the nineteenth international conference on artificial intelligence and law. 2023, p. 227–36.
[46] Vitaletti A, Zecchini M. A tale on decentralizing an app: the case of copyright management. In: DLT. 2023.
[47] Etherisc. Etherisc — Pioneer in parametric blockchain insurance. 2025, URL https://etherisc.com/. [Accessed 10 September 2025].
[48] Durovic M, Willett C. A legal framework for using smart contracts in consumer contracts: Machines as servants, not masters. Mod Law Rev 2023;86(6):1390–421.
[49] Micklitz H-W, Saumier G. Enforcement and effectiveness of consumer law. Springer; 2018.
[50] Benöhr I. EU consumer law and human rights. OUP Oxford; 2013.
[51] Ben-Shahar O. One-way contracts: Consumer protection without law. Walter de Gruyter GmbH & Co. KG; 2010.
[52] Commission E. Digital fairness - fitness check on EU consumer law. 2024, URL https://tinyurl.com/h5xnnj78. [Accessed 24 March 2025].
[53] Riefa C. Protecting vulnerable consumers in the digital single market. Eur Bus Law Rev 2022;33(4).
[54] Riefa C. Transforming consumer law enforcement with technology: from reactive to proactive? J Eur Consum Mark Law 2023;12(3):97–101.
[55] Borgogno O. Usefulness and dangers of smart contracts in consumer and commercial transactions. In: DiMatteo L, Cannarsa M, Poncibo C, editors. A modified version is forthcoming in "Smart contracts blockchain technology: role of contract law". Cambridge University Press; 2019.
[56] Forbes L. Consumer protection in the face of smart contracts. Loy Consum L Rev 2022;34:45.
[57] Lippi M, Pałka P, Contissa G, Lagioia F, Micklitz H-W, Sartor G, Torroni P. CLAUDETTE: an automated detector of potentially unfair clauses in online terms of service. Artif Intell Law 2019;27:117–39.
[58] De Franceschi A. Consumer's remedies for defective goods with digital elements. J Intell Prop Inf Tech E Commer Law 2021;12:143.