



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

in cotutela con University of Luxembourg - Université du Luxembourg

DOTTORATO DI RICERCA IN
LAW, SCIENCE AND TECHNOLOGY

Ciclo 38

Settore Concorsuale: 12/H3 - FILOSOFIA DEL DIRITTO

Settore Scientifico Disciplinare: IUS/20 - FILOSOFIA DEL DIRITTO

DRAFTING COMPUTABLE CONTRACTS: EXECUTABLE REPRESENTATIONS OF
LEGAL AGREEMENTS

Presentata da: Alessandro Parenti

Coordinatore Dottorato

Antonino Rotolo

Supervisore

Giovanni Sartor

Supervisore

LEON VAN DER TORRE

Co-supervisore

Cosimo Laneve

DISSERTATION
Defence held on 14 April 2026 in Bologna
to obtain the degree of
DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE
AND
DOTTORE DI RICERCA
IN LAW, SCIENCE AND TECHNOLOGY
by
Alessandro PARENTI
Born on 16 June 1996 in Poggibonsi (SI) (Italy)
Drafting Computable Contracts: Executable
Representations of Legal Agreements

Dissertation defence committee

Prof. Dr Alberto De Franceschi, Chair
Professor, University of Ferrara

Prof. Dr Giovanni Sileno
Assistant Professor, University of Amsterdam

Prof. Dr Silvia Crafa
Associate Professor, University of Padova

Prof. Dr Leon van der Torre, Supervisor
Professor, University of Luxembourg

Prof. Dr Giovanni Sartor, Co-Supervisor
Professor, University of Bologna

Abstract

Computable contracts aim to express the terms of legal agreements in a form that computers can interpret and execute. While their roots predate distributed ledger technologies, the rise of blockchain has renewed interest in this domain through the concept of “smart contracts”, although this term has been misinterpreted and has lost its original connection with the legal dimension. After providing a conceptual background that clarifies common misinterpretations and disambiguates overlapping terms in this domain—such as smart contracts, e-contracts, or computable contracts—this thesis ultimately focuses on the latter.

This dissertation investigates how contract terms can be represented in a computable form such that the behaviour of the computable contract remains faithful to the legal semantics of the agreement. This research question is addressed along two complementary lines of inquiry. First, the thesis explores the design of a domain-specific language for computable contracts that preserves legal meaning while supporting execution in digital environments. This results in the implementation of the *Stipula* language, whose specific features reflect key contract elements such as obligations, amendment clauses or statutorily implied terms. Second, it examines how different programming paradigms—particularly logic-declarative and object-oriented imperative approaches—perform in modelling computable contracts, including *Stipula*, through a comparative framework grounded in a real-world case study on data-sharing agreements under the EU Data Act.

The findings show that no single paradigm fully captures the complexity of computable contracting: declarative approaches align more closely with normative reasoning, whereas imperative approaches provide stronger operational control. Although domain-specific solutions offer important benefits, the thesis suggests that hybrid and layered language architectures are a promising direction for unifying legal semantics and computational automation.

Acknowledgments

The work has been supported by the “CompuLaw” project, funded by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (Grant Agreement No. 833647).

Contents

Abstract	1
1 Introduction	1
1.1 Motivations and Context	1
1.2 Research Question and Methodology	3
1.3 Thesis Structure	4
I Background	7
2 The Path to Computable Contracts	9
2.1 The Evolution of Digital Contracting	9
2.1.1 EDI: digital representation of contract data	9
2.1.2 Ricardian Contracts	11
2.1.3 E-contracts	13
2.1.4 Smart Contracts and DLTs	14
2.1.5 Computable Contracts	19
2.2 The Legal Framework	20
2.2.1 The Legal Validity of Computable Contracts	20
2.2.2 Legislative Interventions	22
2.3 Key Findings	25
3 From Law to Code: The Role of Languages	27
3.1 Legal Rules as Code	28
3.1.1 Deontic Positions: Representing Normative Force	29
3.1.2 Defeasibility of Legal Reasoning	30
3.1.3 Isomorphism and Validation	31
3.2 Imperative and Declarative Paradigms: Design Principles	31
3.2.1 Imperative vs Declarative Computable Contracts	33
3.3 Domain-specific languages for Legal Contracts	34
3.4 Key Findings	37

II	Stipula DSL	39
4	Stipula Foundations	41
4.1	Patterns of Legal Contracts	41
4.2	Semantics	45
4.3	Integrated Environment	50
5	Amendments	51
5.1	Amending Contracts	52
5.2	<i>Stipula</i> with Amendments	54
5.2.1	The Syntax of <i>H-O Stipula</i>	55
5.3	Amendments in Practice	56
5.3.1	Additive Amendment	58
5.3.2	Overriding Amendment	61
5.4	The Semantics of <i>higher-order Stipula</i>	63
5.5	Amendments Supplements: agreements and constraints	65
5.6	Key Findings	66
6	Contract Analysis	69
6.1	Incompatible clauses as a source of problems	70
6.2	Incompatibilities in <i>Stipula</i> and their analyses	74
6.3	Key Findings	79
7	Statutory Integration	81
7.1	The Role of Law in Contracts	82
7.2	Formal Approaches to Conflicting Legal Sources	84
7.3	The import mechanisms in <i>Stipula</i>	86
7.4	Semantics of the import extension	94
7.4.1	Conflicts Among Imports	95
7.4.2	Normative Conflicts without a Priority Order	96
7.5	Key Findings	97
III	Survey	101
8	Setting the stage for Comparison	103
8.1	Evaluation Criteria	104
8.1.1	Legal Fidelity	104
8.1.2	Accessibility	106
8.1.3	Operational Expressiveness	108
8.2	The Case Study: A Data Use Agreement under the EU Data Act	109
8.2.1	Computable Contracts for Data Sharing	110

8.2.2	A Data Use Agreement in the Car Rental sector	111
8.2.3	From Contract Clauses to Normative Components	113
9	Comparison	115
9.1	Scope of the Evaluation	116
9.2	Logic-Based Declarative Languages	117
9.2.1	Prolog and First-Order logic	117
9.2.2	Logical English	127
9.2.3	Defeasible Deontic Logic on ASP	132
9.2.4	Logic Production Systems (LPS)	140
9.3	Object-Oriented Imperative Languages	149
9.3.1	Solidity	149
9.3.2	Stipula	157
9.4	Discussion	163
9.4.1	Legal Fidelity	164
9.4.2	Accessibility.	167
9.4.3	Operational Expressiveness.	167
9.5	Trade-offs and Conclusions.	169
	Conclusions	173
10	Final Reflections and Future Directions	173
10.1	Summary of Key Findings	173
10.2	Limitations of the Present Work	175
10.3	Concluding Remarks	176
A	Full Code Implementations	179
A.1	Logical English	179
A.2	Defeasible Deontic Logic on ASP	182
A.3	Logic Production System	184
A.4	Solidity	186
A.5	Stipula	191
I		193
	Bibliography	193

CONTENTS

List of Figures

5.1 Take-or-pay amendment	59
5.2 Price-revision clause amendment	61
5.3 Substitution of Provider	62
5.4 Static constraints on Amendments	66
9.1 The Outcome of an LPS program	142
9.2 Execution of the LPS D.U.A	146

LIST OF FIGURES

List of Listings

5.1	The Gas Supply contract in <i>Stipula</i>	56
6.1	Bakery warranty	74
6.2	Timber purchase	76
6.3	Mortgage	77
7.1	<i>Stipula</i> VPN service Agreement with termination clause	88
7.2	Norm m 1373cc: Article 1373 of the Italian Civil Code	89
7.3	<i>Stipula</i> VPN service Agreement with termination clause	90
7.4	Norm m 52cons: Article 52 of Italian Consumer Code	92
7.5	<i>Stipula</i> VPN service Agreement with termination clause 4(b)	93
9.1	Prolog Formalisation of DUA	119
A.1	Logical English Formalisation of DUA	179
A.2	DDL Formalisation of DUA	182
A.3	LPS Formalisation of DUA	184
A.4	Solidity Formalisation of DUA	186
A.5	Stipula Formalisation of DUA	191

Chapter 1

Introduction

1.1 Motivations and Context

In recent years, the advent and rapid evolution of blockchain technology have attracted growing attention across technological, economic, and legal domains. As of October 2025, the total market capitalisation of the cryptocurrency ecosystem exceeds four trillion dollars¹, reflecting the enormous financial value now circulating within the blockchain ecosystem. This unprecedented growth has inevitably fuelled widespread enthusiasm for the potential applications of this technology, among these *smart contracts*. While blockchain technology has popularised this notion, the present thesis approaches the topic from a broader perspective, focusing specifically on the computable representation of legal agreements.

The concept of “smart contract” was originally coined by Nick Szabo in 1996, with a precise legal dimension. Szabo described them as “contracts embedded in the world” whose terms, expressed as lines of code, could be automatically executed without human intervention, an idea he illustrated through the example of vending machines. Such a design, he argued, could reduce enforcement costs and make contractual breach less attractive, since violating the technological layer (for instance, the machine) would be more burdensome than complying with the agreed terms.

This conception, combined with an often partial understanding of the actual

¹Data available at coinmarketcap.com

functioning of distributed ledger technology (DLT), has led to several misconceptions about what blockchain based smart contracts are and what they can realistically achieve. In both public and academic discourse, smart contracts have often been portrayed as fully autonomous instruments capable of enforcing agreements independently, a view that tends to overstate their technical feasibility as well as their legal significance [32, 178].

Within the blockchain environment, in fact, smart contracts simply denote any computer program deployed on a distributed ledger capable of managing digital resources and to automatically execute specific operations when predefined conditions are met². Whether such programs can also represent and automate legally binding agreements, however, concerns only a specific subset of their possible applications. To capture this distinction, the term “smart legal contract” has been increasingly adopted to refer to those smart contracts that represent or give effect to a legal agreement, as opposed to other types of protocols running on distributed ledger technologies.

Yet, the idea of expressing contractual terms in a computable form long predates blockchain technology. Since the 1970s, scientific communities, such as Artificial Intelligence and Law or Business Management, have explored how contractual terms could be modelled in ways that are both human readable and machine processable, in order to improve communication efficiency, monitoring, compliance, and automation. Early systems for Electronic Data Interchange (EDI) standardised the exchange of business documents among enterprises, and from the 1980s onwards, the AI and Law field began to model legal knowledge in computational terms. By the late 1990s and early 2000s, these efforts extended specifically to contracts, giving rise to the notion of *electronic contracts* or *e-contracts*.

This whole body of research can today be framed within the broader field of *computable contracts*. The notion of computable contracts was introduced by Sarden in 2012 [162], to describe agreements that are expressed, in whole or in part, in a computer readable form and equipped with mechanisms that enable a certain degree of automated verification of compliance. This definition is specific enough

²According to the Ethereum Whitepaper, which first introduced smart contracts in the blockchain environment, they are described as “cryptographic boxes that contain value and only unlock it if certain conditions are met” [25].

to clearly identify the technology’s function, in contrast to, for example, *e-contract* that can also be used loosely to refer to contracts concluded online. On the other hand, it is sufficiently general to remain independent of any particular technological implementation, unlike *smart legal contract* which necessarily remains tied to blockchain technology.

The present work is therefore centred on the study of computable contracts and examines, from a legal informatics perspective, how programming languages can model the main concepts and patterns that shape legal contracts. In doing so, it builds upon the extensive knowledge developed before the advent of blockchain technology, while also acknowledging the role that blockchain has played in renewing interest in the computational representation of legal agreements.

1.2 Research Question and Methodology

The development of computable contracts centres on expressing the terms, clauses, and, more generally, the set of rules defined by a contract in a form that can be processed and executed by a computer. This requires translating concepts that are usually expressed in natural language—an inherently open-ended and ambiguous medium—into a formal language that is, by nature, precise and deterministic. This transposition is far from trivial.

Therefore, the present dissertation focuses on the following research question: **How can we represent contract terms in a computable form in such a way as to ensure that the behaviour of the computable contract is faithful to the legal semantics of the agreement?**

This question is explored by focusing on programming languages as the main means through which the rules and provisions constituting a contract can be conveyed to a computational system. This overarching research question is addressed through two complementary sub-questions:

- **RQ1:** To what extent can a domain-specific language for computable contracts be designed to preserve legal meaning while providing the operational capabilities required for execution in digital environments?
- **RQ2:** What are the benefits and limitations inherent to different program-

ming paradigms—specifically logic-declarative and imperative/object-oriented—for the representation and implementation of computable contracts?

The first sub-question is embodied in the development of *Stipula*, a domain-specific language for computable contracts. In this context, the thesis develops and analyses language features whose semantics are intended to capture fundamental contractual elements and recurring patterns.

The second sub-question is addressed by analysing logic-declarative and imperative/object-oriented programming paradigms. The former has historically been used in the field of AI and Law to model legal knowledge due to its expressiveness and declarative nature, whereas the latter underpins most of today’s blockchain-based smart contract development. The thesis provides a comparative analysis of selected formal languages belonging to these paradigms and evaluates them against a common set of criteria.

Ultimately, given the availability of several existing languages for modelling legal rules, this thesis is guided by the expectation that different approaches exhibit complementary strengths and limitations, and that a dedicated domain-specific language may better address the multiple dimensions of computable contracts.

1.3 Thesis Structure

The thesis is structured into three main parts. The first part establishes the conceptual and legal foundations of the research. It provides a historical overview of the evolution of digital contracting, clarifying the terminology and distinguishing computable contracts from related concepts such as electronic and smart contracts. It then examines the legal framework applicable to computable contract technologies, including recent European regulatory initiatives, and introduces the core principles of legal knowledge representation relevant to computational modelling. Finally, it reviews existing formalisms and design paradigms proposed in the field.

Having established the grounds of computable contracts development, the dissertation focuses on the actual design of computable contracts by presenting the development of the *Stipula* language, a domain-specific solution for modelling and

executing computable contracts. The second part, in fact, describes the language’s main primitives, its formal syntax and semantics, and the relationship with legal concepts. Then introduces the language extensions developed in the thesis, i.e., ad hoc features for amendments, contract analysis, and statutory integration, demonstrating how *Stipula* supports contractual evolution, detects structural inconsistencies, and incorporates external legal sources.

Having introduced a concrete design solution, the third part aims to evaluate it against other relevant approaches. To this end, it conducts a structured comparative analysis. Using a case study based on a Data Use Agreement under the EU Data Act, it assesses representative languages from both logic-based declarative and imperative/object-oriented paradigms. The comparison highlights the legal fidelity, accessibility, and operational expressiveness of each solution, shedding light on the trade-offs that arise when translating legal semantics into executable behaviour. The discussion culminates in reflections on future hybrid architectures for computable contracting.

Part I

Background

Chapter 2

The Path to Computable Contracts

The ambition to express the terms of legal agreements in a form that machines can interpret, execute, and possibly enforce has a long and multifaceted history. Over the decades, this line of research has taken various directions, often conducted under overlapping or imprecise terminology. Recently, the emergence of distributed ledger technologies (DLTs) and, in particular, their use for implementing so-called smart contracts, have significantly boosted the interest in this field. While this has prompted both academic and institutional effort aimed at defining the legal status of these new digital form of contracting, it also contributed to create conceptual confusion concerning terminology and the realistic boundaries of technological automation in contracting.

This chapter aims to provide the necessary background for understanding the context in which computable contracts came about, clarify the definitional scope of this technology and illustrate how these can fit in the current legal framework.

2.1 The Evolution of Digital Contracting

2.1.1 EDI: digital representation of contract data

The idea of representing a contract in digital form to streamline transactional processes originated in the 1960s with the rise of Electronic Data Interchange

(EDI), i.e., the development of common standards for the digital representation of and communication of trade documents. This technology enabled the computer-to-computer communication of legal transactions such as invoices, bills of lading, or purchase orders rather instead of the paper-based one, aiming to improve the speed, reliability, and efficiency of business operations [49].

EDI can be defined as “the electronic transfer, from computer to computer, of commercial and administrative data using an agreed standard to structure messages”¹.

Just as a natural language has grammar and vocabulary, EDI messages follow a defined syntax but, relying on a common data format, it allows different information systems to receive and interpret external data and process it autonomously for internal business process. This development introduced several key advantages.

First, this reduces both administrative and transaction costs by eliminating the need for paper-based communication, manual form handling, filing, and other office-related expenses[17, 49]. For instance, an EDI system can support an ongoing supplier-buyer relationship in which prices and quantities are adjusted based on market conditions, provided such changes fall within the parameters of pre-agreed templates [157]. Moreover, EDI improves the quality of information by minimizing ambiguity and reducing errors. The strict structure of EDI messages significantly lowers the risk of misinterpretation and disputes regarding the meaning of contractual terms. This is reflected in the minimal number of legal disputes reported in the first forty years of EDI’s use [50].

The realization of these advantages depended on a crucial precondition that shaped the EDI system: *trust* among participants. The very notion of a ‘common standard’, in fact, implies a prior agreement or understanding on how data is to be exchanged. Even before legislators formally recognized the legal status of EDI, its use was driven by a shared understanding among companies, which often regarded EDI transactions as legally binding despite the absence of explicit legal regulation [173].

In fact, the integration of this standard into businesses’ infrastructures, while offering numerous benefits, also introduced a set of risks inherent to its digital

¹Definition taken from 94/820/EC: *Commission Recommendation of 19 October 1994 relating to the legal aspects of electronic data interchange.*

nature that could have repercussions across the entire network [144]. For instance, the disclosure of sensitive information to unauthorized entities can occur if one party lacks adequate security controls. Similarly, the use of unreliable third-party software by one participant might lead to system-wide malfunctions. Other concerns include the repudiation of the origin or receipt of EDI messages (i.e., when a party denies having sent or received a given message). Due to these vulnerabilities, the existence of strong trust among business partners was essential not only to ensure that such failures were rare, but also to resolve potential errors cooperatively rather than escalating them into legal disputes.

EDI represents an early and influential example of the digital representation of contractual information, enabling structured communication between businesses and the internal automation of transaction-related processes. However, EDI did not aim to automate the enforcement of contracts, and its success was largely embedded within trusted commercial relationships operating under agreed standards and templates.

2.1.2 Ricardian Contracts

Over time, not only contractual data, but also the very objects of contracts themselves became digital. This evolution required a step beyond the mere digitalisation of contractual information: it called for a connection between traditional, paper-based contracts and their digital—or executable—objects.

The Ricardian Contract was introduced by Ian Grigg around 1996 to address this need, in the context of a system for the digital issuance and management of financial instruments, such as bonds and digital currencies [85]².

The structural separation between the instrument being transacted and the legal terms and conditions governing it made the identification of the digital instrument challenging. Traditional schemes relying on numeric identifiers, issuer names, or tuples proved insufficient. These raised practical questions regarding whether a central registry would be required for validation or whether multiple issuers could define their own identifier namespaces [86]. Within this uncertainty produced by the digital instrument alone, the specific legal significance remained

²The Ricardo Payment System: <https://www.systemics.com/docs/ricardo/>

buried in external documents such as user agreements, disconnected from the core technical infrastructure.

Grigg’s proposal inverted this paradigm by making the contract itself the primary unit of issuance. From a technical perspective, a Ricardian contract is structured as a triple:

- *Prose*: legal semantics written in natural language;
- *Code*: operational semantics implemented in a programming language;
- *Parameters*: deal-specific values such as dates, prices, or deadlines [88]. Parameters serve as a critical interface between the legal and technical components. They define the specific terms of the agreement and are used to instantiate variables in the smart contract code³.

This combined artifact is cryptographically signed by the issuer and linked to every related transaction through a secure hash, ensuring traceability and authenticity of every transaction [86].

The Ricardian model is recognized as a foundational design pattern for computable agreements, and has informed substantial subsequent research and development in the field [38, 36, 88, 33]. Among the most relevant, Clack et al. [38, 36] developed a methodology for serialising computable contracts by defining legal prose templates and corresponding operational code models that can be instantiated through a shared set of parameters. The process is grounded in the definition of an abstract core specification, which allows contract components to be modular and reusable.

In this architecture, a crucial role is played by markup languages and legal ontologies. The first are needed to enrich contract documents with embedded data needed by computers to process them. Depending on the markup language, it is possible to describe the document’s structure (e.g., using Akoma Ntoso language [135]) or even the legal semantics for computational reasoning (e.g., using LegalRuleML [12]). Second, the adoption of shared legal ontologies—such as the

³The concept is flexible and can be tailored to the complexity and design goals of a contract—for example, to capture quantities, costs, or timeframes.

FIBO (Financial Industry Business Ontology)⁴ or PRONTO [134]—is key to ensuring semantic consistency and interoperability across systems and jurisdictions. Ontologies define the meaning of legal and technical terms in a standardised way, helping to avoid ambiguity and ensuring that all systems involved interpret the contract consistently.

2.1.3 E-contracts

As business processes became increasingly automated (e.g., receiving, preparing and shipping orders) the need arose to integrate contract logic directly into the operational system so as to ensure the legal compliance of the automated system. Such integration is, on the one hand, necessary to avoid critical failures caused by a mismatch between the contractual and operational layers—ensuring that systems can detect and respond to transaction events that may lead to contract violations. On the other hand, it enables a greater degree of automation in contract monitoring and enforcement, thereby improving the overall efficiency and reliability of business operations [81, 167].

In this context, the formalisation of contractual terms into executable logic has been studied under term *e-contracts* since the early 2000s, especially applying formal logic. As defined by Krishna et al., “An e-contract is a contract modeled, specified, executed and enacted (controlled and monitored) by a software system (such as a workflow system)” [109]. Unlike EDI technology, which primarily focused on representing contracts’ *data* in a machine readable form, e-contracts capture contracts’ *semantics* enabling monitoring and compliance checking [123].

Work in this area has identified several key requirements for adequately representing legal agreements.

Several contributions have shown the importance of a state-aware design for formal contract models [53, 129, 142]. This enables the tracking of the various stages of the contract life cycle, from formation to execution, potential violation etc. The life-cycle orientation is crucial for evaluating compliance over time rather than in isolated snapshots.

Another foundational requirement is the ability to represent deontic concepts

⁴<https://spec.edmcouncil.org/fibo/>

such as obligations, permissions, and prohibitions, which define the normative expectations placed on the contracting parties [115, 124, 75]. Closely related is the need to model contract violations and the associated contrary-to-duty (CTD) obligations, which specify the secondary duties that arise in response to such violations [80, 75].

A further requirement is the support for defeasible reasoning [72], which allows the system to handle incomplete or conflicting information through non-monotonic logic. This is particularly important in dynamic environments where new information may override prior conclusions, and where normative conflicts may require prioritization and resolution (infra Section 3.1).

Finally, some contributions have applied temporal logic to e-contracts to allow the expression of deadlines, durations, and time dependencies between actions [81, 115] natively in the specification of the contract. This allows systems to evaluate not only whether actions occur, but when they occur in relation to contractual commitments.

This line of research is foundational for the development of computable contracts. However, the term e-contract is rather generic and has been used across multiple disciplines with varying degrees of specificity. In legal contexts, for instance, it is often employed broadly to denote any contract concluded through electronic means (such as online or via email) without further qualification [73]. This broad usage may lead to conceptual ambiguity.

2.1.4 Smart Contracts and DLTs

The term *smart contract* was originally introduced by computer scientist and cryptographer Nick Szabo in 1994 [166] and defined as “a computerized transaction protocol that executes the terms of a contract”. Although this definition is not conceptually far from that of e-contracts, and indeed the two largely overlap in terms of implementation methods, the distinction lays in the purpose and scope of application.

While e-contracts identify practical tools to support business management and process control, the notion of smart contracts carries a more philosophical dimension. As Szabo explains, “many kinds of contractual clauses (such as collateral,

bonding, delineation of property rights, etc.) can be embedded in the hardware and software they deal with, in such a way as to make breach of contract expensive [...] for the breacher” [164].

This idea is exemplified by Szabo through the case of a vending machine: the machine consists of hardware and software that together embody a legal contract. This is expressed through a series of *if-then* conditions (*if the right price is inserted, then dispense the product*) and the cost of tampering with the machine would likely exceed the value of its contents. Such a mechanism enables contracting between unknown parties who don’t need to trust each other but can instead rely on the automated process of the machine.

Especially in the context of global online commerce, where contract enforceability is difficult and expensive, in fact, securing transactions through mathematical and technological solutions can allow for a higher predictability of risks and allow even smaller businesses to engage in multinational transactions [165].

In essence, smart contracts aim to reduce the risk of non-compliance by delegating execution to computer code that automatically enforces the agreed terms. While fascinating, this concept remained *de facto* under the radar until it was re-proposed by Vitalik Buterin within the framework of Distributed Ledger Technologies (DLTs), to describe the core infrastructure of blockchain applications and the so-called Web 3.0.

The rise of Blockchain

In 2008, the pseudonymous Satoshi Nakamoto introduced Bitcoin, the first ever blockchain [127]. The Bitcoin whitepaper described a peer-to-peer payment system where each participant to a peer to peer network keeps an identical copy of the ledger recording the transactions. In such a system, different cryptographic techniques are employed to ensure consensus among participants on the order of transactions in the ledger (Proof-of-Work), security and tamper resistance of the ledger (asymmetric cryptography and hashing) [70]. This architecture ensures transparency, security, and integrity while eliminating the need for a trusted third party. For this reason, blockchain is often described as a *trustless* system: trust is placed in the protocol and its cryptographic rules rather than in any human

intermediary.

Building upon Bitcoin’s technical foundations, Ethereum [25] extended blockchain functionality beyond payments. Ethereum introduced a decentralized execution environment—the *Ethereum Virtual Machine*—capable of running general-purpose code known as *smart contracts*. In this context, however, the term “smart contract” does not have the specific legal dimension given by Szabo, but simply refer to self-executing computer programs whose operations are validated and recorded on the blockchain. As described in the Ethereum whitepaper, these are “cryptographic boxes that contain value and only unlock it if certain conditions are met” [25].

To clarify this ambiguity, scholars have introduced a useful distinction between *smart contract code* and *smart legal contracts* [159]. The former refers to autonomous code that executes digital transactions, while the latter designates those instances where such code also fulfils the substantive and formal requirements of a legal agreement within a given jurisdiction [66]. Smart legal contracts therefore aim to bridge the technical notion of self-executing software with the legal notion of enforceable agreement, reflecting more closely Szabo’s original idea. What blockchain added to Szabo’s original vision was a secure computational environment where smart contracts could be run on, in a trust-less way.

Realizing the Limitations

The rapid expansion of the blockchain ecosystem (particularly in the financial sector) has fuelled growing enthusiasm for its potential applications, including smart contracts. Promoted as a disruptive innovation, smart contracts were often portrayed as a means to revolutionize commercial and legal practices by enabling breachless, self-enforcing agreements capable of reducing both transaction and litigation costs [178, 32].

However, as scholars from both the legal and computer science domains examined the phenomenon more closely, it became evident that much of this early enthusiasm rested on some misconceptions about both the technical functioning of smart contracts [29, 120] and their legal implications [96, 157]. These two categories are addressed below.

Technical Misconceptions. The most common technical misconceptions concern the supposed *trustless* and *self-enforcing* nature of smart contracts.

The first assumes that blockchain technology completely removes the need for trust. In this view, execution is fully automated and guaranteed by code and cryptography, without relying on third parties such as banks or courts. While blockchains indeed replace traditional intermediaries, they do not eliminate trust altogether; rather, they *reallocate* it. Users must still rely on a variety of actors—core developers, miners, or validator nodes—whose influence over the protocol’s evolution and operation is not always transparent or formally governed [174, 55].

This trustless narrative is further challenged by the so-called *oracle problem* [29]. Because blockchains are closed systems, they lack native access to external information. Whenever the execution of a smart contract depends on real-world data (for example, stock prices or weather conditions), such data must be introduced through *oracles*—external entities, services, or programs that collect off-chain information and feed it into the blockchain [51]. By serving as interfaces between on-chain and off-chain environments, oracles necessarily reintroduce an element of trust, as the system depends on their reliability and correctness. In practice, they are also used to delegate various tasks off-chain, such as privacy management or computationally intensive operations (e.g., randomness generation), further diluting the idea of complete autonomy [28].

The second misconception concerns the belief that smart contracts can autonomously enforce all contractual obligations. This assumption must be significantly qualified.

First, many contractual provisions—such as those invoking “good faith” or “reasonable efforts”—are inherently unsuitable for formalization into deterministic code [162, 60]. Such open-textured clauses often require human interpretation or judicial assessment, making them incompatible with purely computational execution.

Secondly, even when contractual logic is fully specified, automation is limited by the scope of blockchain control. Smart contracts can directly manage only *on-chain* assets (cryptocurrencies, tokens, NFTs), while obligations concerning *off-chain* assets—physical goods, fiat currency, or real-world services—cannot be

enforced autonomously [179]. Execution in these cases ultimately depends on external cooperation or hybrid mechanisms such as escrow arrangements and pre-funded deposits [121, 122]. Without such structures, performance again relies on human behaviour and intention, reintroducing precisely the trust and discretion smart contracts were meant to eliminate.

In short, the promise of self-enforcement depends not only on the expressibility of legal terms in code, but also on whether control over the relevant resources can be embedded within the computational environment itself. True automation is possible only when both the *logic* of the obligation and the *objects* it governs exist within the same digital domain. Beyond this boundary, smart contracts remain powerful tools for automating certain operations, but not substitutes for the broader institutional and interpretative functions of law.

Legal Misconceptions. A further misconception often associated with smart contracts concerns their presumed ability to eliminate ambiguity. Because computability requires that contractual terms be expressed through precise and discrete conditions, smart contracts are believed to reduce interpretative uncertainty and the risk of disputes.

While such precision may indeed enhance certainty, it is not necessarily desirable from a legal standpoint [157, 120]. Legal vagueness and flexibility are not flaws but deliberate drafting tools that allow contracts to adapt to unforeseen circumstances [18]. Since smart contracts cannot handle indeterminacy, parties must attempt to anticipate every possible scenario *ex ante*—an unrealistic task that both increases drafting complexity and limits adaptability [136].

These difficulties are amplified by the immutable nature of blockchain technology. Immutability, while often celebrated as a guarantee of integrity, can conflict with fundamental principles of contract law, which allow parties to amend, renegotiate, or terminate their agreements [136]. Moreover, legal systems must retain the ability to nullify or modify agreements that are unlawful, contrary to public policy, or entered into without proper capacity [121, 120]. The irreversible character of blockchain transactions thus clashes with the requirement that contracts remain subject to legal oversight and potential revision by competent authorities.

In short, the assumption that smart contracts can remove ambiguity or replace

legal flexibility overlooks the essential role of interpretation, renegotiation, and judicial review in ensuring fairness and adaptability within contractual relationships.

2.1.5 Computable Contracts

The evolution traced throughout this chapter illustrates how the ambition to digitalise and automate contractual relationships has developed along multiple, partly overlapping lines of research. From Electronic Data Interchange (EDI) to Ricardian contracts, e-contracts, and smart contracts, these diverse approaches share a common goal: enabling the expression and performance of contractual obligations in a form that computers can process and, to some extent, execute automatically.

Today, these research efforts converge under the concept of *computable contracts*. The term, introduced by Henry Surden in 2012 [162], has gained broad acceptance due to its conceptual precision and technological neutrality. Unlike earlier expressions such as “smart contract”, which denote a specific implementative solution, or “e-contracts”, that has a too broad applicability, Surden’s term provides a clear and shared terminology for describing the automation of contractual processes without committing to any specific technological implementation.

A computable contract is characterised by two main features. First, it is *data-oriented*: the agreement, in whole or in part, is represented as structured, machine-processable data. This reflects the principles of EDI systems and underlies many subsequent developments in digital contracting.

Second, it is *computable* in the sense that it enables the automatic assessment—and, in some cases, the enforcement—of contractual obligations. At a basic level, computability allows the system to monitor the state of the contractual relationship and detect non-compliance, for instance, by verifying whether a payment was made on time. At a more advanced level, it may support the automated enforcement of certain terms, such as preventing the exercise of a right once a deadline has expired.

The data-oriented dimension allows machines to extract and process contractual information, while the computable dimension adds a semantic layer that enables the system to interpret and act upon legal obligations. Therefore a computable contract presupposes the creation of a single artefact that encapsulates

both legal semantics and executable logic. This represents a step beyond the Ricardian contracts architecture, for example, where the legal and technical layers were kept distinct but linked through shared parameters. Pursuing this integration has led to the development of *domain-specific languages* for contract drafting and execution. These languages are designed to be expressive enough for legally trained professionals to encode contractual logic directly, without the need for a later translation from natural language to code. Such an approach reduces the risk of inconsistencies between legal intent and operational behaviour, thereby bringing the ambition of truly computable contracting one step closer to reality [37].

2.2 The Legal Framework

The idea of representing agreements in a computable form has raised the question of whether code-based expressions of consent can satisfy the traditional elements of contractual validity. This section examines the legal status of computable contracts according to current laws. While early analyses and contributions mentioned here were largely motivated by the emergence of blockchain and smart contracts, their findings extend to computable contracts in general, since both involve assessing the legal significance of computer-executable representations of agreements [3].

2.2.1 The Legal Validity of Computable Contracts

Rather than focusing on one jurisdiction, here we reference principles of contract law that are shared across different legal system and codified in instruments such as the *Principles of European Contract Law* (PECL) and the *Draft Common Frame of Reference* (DCFR)⁵. These identify two essential elements for contract formation: (1) a sufficient agreement (offer and acceptance), and (2) the intention of the parties to be legally bound.

Offer and Acceptance. An *offer* is any expression of willingness to be bound by certain terms addressed to another party or to the public at large. A computable

⁵The DCFR, supported by the European Commission, synthesises the core principles of European contract law [95].

contract entails the expression of an offer and its acceptance, albeit through a digital medium. This may correspond to making code publicly accessible, deploying a contract on a shared infrastructure, or communicating a code-based proposal to identified parties [59, 66]. For the offer to be valid, it must specify all essential terms (*essentialia negotii*); otherwise, it remains an invitation to treat.

Acceptance, conversely, occurs when the other party manifests consent—either through an explicit statement or through conduct consistent with the proposed terms.⁶ In digital systems, acceptance may be expressed by performing an action (for example, executing a transaction or invoking a function) that signals agreement per *factia concludentia* [32]. These arrangements often resemble unilateral contracts, where acceptance occurs through performance rather than a formal declaration.

Contractual Intention. In addition to mutual assent, the parties must intend to be legally bound. According to PECL, intention is determined objectively, based on whether a reasonable observer would interpret the parties’ actions as showing intent to create legal obligations.⁷ In computable contracts, the act of authorising code execution, deploying a program, or committing digital resources generally supports this inference [178]. Nonetheless, questions remain regarding meaningful consent when the code’s functioning is opaque to non-technical users. As with on-line “clickwrap” agreements, best practices suggest that a human-readable version of the terms should accompany the code to ensure effective notice and informed consent [66].

Form and Electronic Signatures. European contract law recognises the principle of informality: a contract need not be in writing unless required by law.⁸ Accordingly, computable contracts can validly exist in electronic form. Within the EU, the *eIDAS Regulation* establishes that electronic documents and signatures cannot be denied legal effect solely because of their electronic nature.⁹ Digital signatures used in computable contracts—such as cryptographic keys or token-based

⁶Art. II-2:204(1) PECL.

⁷Art. II-2:102 PECL.

⁸Art. II-2:101(2) PECL.

⁹Art. 25(1) Regulation (EU) 2014/910.

authentication—generally qualify as *simple electronic signatures* and as *advanced* or *qualified* signatures when the connection between a signature and a physical person is certified by an authority [94, 65].

Fitting Computable Contracts within Contract Law. Beside the applicability of general conditions of contract validity, both scholars and institutions have extended the analysis of the various categories and institutes of contract law.

A prominent example is that of the UK Law Commission which published an *Advice to Government on Smart Legal Contracts*. This found that “current legal principles can apply to smart legal contracts in much the same way as they do to traditional contracts, albeit with incremental and principled development of the law where necessary” [44].

Comparative studies across European jurisdictions confirm this adaptability [169, 117, 21, 116]. In short, computable contracts represent a new technological form of contracting, but can operate within the current legal framework.

2.2.2 Legislative Interventions

Although current legal frameworks seem sufficient to regulate support computable contacts, the sudden hype on DLTs has led several countries to enact legislation to explicitly recognise the validity of blockchain smart contracts.

In Europe, Italy was among the first to introduce legal definitions of Distributed Ledger Technologies and smart contracts through Law No. 12/2019, although these were criticised for linking the concept too narrowly to blockchain infrastructures [20, 116]. Malta and France have also recognised specific uses of DLTs, while the UK has approached the topic through interpretative guidance rather than legislation [168].

In the United States, the *ESIGN Act* and *UETA* provide general legal validity for electronic contracts, and several states—such as Arizona, Tennessee, and New York—have expressly acknowledged the enforceability of blockchain-based smart contracts [44].

The EU Data Act

On a different note, the European legislator has recently addressed computable contracts not to recognise them as a legal category, but rather to promote one of their potential applications. In doing so, it adopted a technologically neutral definition—although still employing the term *smart contracts*—and established a set of requirements for their use.

These provisions are contained in Regulation (EU) 2023/2854, i.e., *Data Act*. Article 2(39) provides a legal definition of smart contracts as:

“a computer program used for the automated execution of an agreement or part thereof, using a sequence of electronic data records and ensuring their integrity and the accuracy of their chronological ordering.”

Although in the author’s view, the term “smart contract” remains prone to misunderstanding and confusion due to its association with the blockchain ecosystem, the definition adopted by the Data Act remains technologically neutral and focuses on the technology’s functions and effect, such as the legal nature, automation, data integrity, and chronological traceability. This interpretation aligns with Surden’s concept of *computable contracts*, while adding an emphasis on robustness and reliability.

The Data Act represents a key component of the EU “data strategy” and aims to foster a functioning data economy by promoting data sharing and clarifying who may create value from data and under what conditions.^[10] It encourages the contractualisation of data exchanges, explicitly acknowledging that data law should be mainly governed by contract law principles [89]. Within this framework, *smart contracts* are proposed as a technical tool to implement and enforce data-sharing agreements. These are mentioned throughout the text as means to foster compliance with the regulation’s obligations (notably in Article 11) [31].

Given the digital and repetitive nature of data-sharing operations, in fact, smart contracts appear particularly suitable for this context: they can automate

¹⁰<https://digital-strategy.ec.europa.eu/en/policies/data-act>

simple, recurring exchanges that take place each time data is accessed or transferred [132]. As discussed in Section 2.1.4, this represents one of the most viable and desirable use cases for smart—or, more precisely, computable—contracts.

Article 30 sets out the technical and legal requirements that smart contracts must meet when used to execute data-sharing agreements. These provisions are designed to ensure legal certainty and operational reliability in the implementation of automated, data-based contractual relationships.

1. **Robustness.** As any piece of software, smart contracts remain vulnerable to programming errors and unintended behaviour. Ensuring robustness requires formal verification methods and secure coding practices to prevent bugs and vulnerabilities in smart contract logic.
2. **Safe Termination.** clearly deriving from the self-executing nature of bc based contracts, this requirement aims at ensuring that a method exists to terminate the continued execution of transaction, in case of material error or agreement *contra legem*, for example. On this point, it has been noted that the *halting problem* is non-decidable for non-trivial programs written in Turing-complete languages [24], therefore it is not generally possible to guarantee that execution will terminate for each possible input. Specific safeguards or constraints shall be implemented to allow for controlled deactivation [131].
3. **Data Archiving and Continuity.** Smart contracts must preserve transactional data and logic to maintain an auditable history, especially in cases of deactivation or termination.
4. **Access Control.** Proper access control must be enforced at both the governance and technical layers [132].
5. **Consistency with Agreements.** Smart contract behaviour must faithfully reflect the terms of the underlying data-sharing agreement. To achieve this, domain-specific languages and formal methods should be employed to ensure semantic consistency between code and contractual obligations [37]

In connection with the data act’s scope, some commentators have noted potential issues related to the use of smart contracts when deployed on blockchain systems, such as reliance on oracles and immutability [31], and privacy concerns [131]. These challenges are manageable in permissioned blockchain systems, where governance and accountability can be more clearly established. However, again, the Data Act maintains a technologically neutral position.

2.3 Key Findings

This chapter outlined the historical and legal foundations of *Computable Contracts*. From the early developments of electronic data exchange to the emergence of blockchain and distributed ledger technologies, a common ambition can be traced throughout: to express the content of legal agreements in a form that machines can interpret and act upon. Although technological infrastructures have evolved considerably, the core challenge remains the same—reconciling the precision required by computation with the interpretative flexibility inherent in legal reasoning.

Current legal frameworks are generally considered capable of accommodating computable contracts as valid sources of obligations, provided that the traditional requirements of contract formation are fulfilled. At the same time, recent legislative initiatives, such as the EU *Data Act*, confirm this compatibility and explicitly promote the use of programmable contracts to enhance the efficiency and reliability of data-driven business relations within the information society.

Together, these analyses highlight the dual nature of computable contracts: they are both legal artefacts and computational systems. Capturing this duality is a demanding task that revolves around the design and use of *formal languages* capable of bridging legal semantics and executable logic. The following chapter builds on this foundation by examining the main paradigms for such representations, introducing declarative and imperative approaches to legal modelling, and reviewing the principal domain-specific languages developed for the drafting and execution of computable contracts.

Chapter 3

From Law to Code: The Role of Languages

Expressing contractual content in a computable form is, to some extent, an act of translation: from the legal language, expressed in natural language, into a computational one, usually a programming language. Translation generally concerns preserving the semantics of a text while modifying its syntax. The goal is therefore not to alter the legal meaning but to express it in a structure suitable for automated processing and execution.

Since the 1980s, scholars in the field of Artificial Intelligence (AI) and Law have explored how to formalise legal semantics to enable automated reasoning [77]. Their work has relied heavily on formal logic as a means of representing legal concepts in ways that can be interpreted and manipulated by computational systems. However, when the ultimate goal is to build a computable contract, the sole “translation” of legal semantics is not sufficient. A working implementation also requires the formalisation of an additional semantic layer, that captures the behaviour of the contract and its parties at a lower, operational level. This layer specifies how contractual obligations, permissions, and prohibitions are to be executed or monitored in practice, thus defining the behavioural semantics of the contract. Such aspects are usually not described in traditional natural-language contracts, which makes their computational representation particularly challenging.

In such a complex framework, the choice of the programming language becomes

crucial, as it must be suitable to express these different semantic layers. To this end, one approach is to rely on a single language specifically designed to encompass all the required levels of abstraction. Alternatively, one can adopt multiple, interconnected languages, each dedicated to a specific layer or function. This latter architecture is described by Clack [37] as the *language stack*.

Programming languages differ in many respects and can be grouped into broad categories according to their fundamental design principles. Among these, two paradigms are particularly relevant for computable contracts and constitute one of the main objects of analysis in this dissertation: the *Declarative* and the *Imperative* paradigms. These paradigms embody distinct design philosophies, each offering specific strengths and limitations when applied to the representation and execution of legal concepts.

The present chapter introduces these distinctions and relates them to the specific objective of developing a computable contract. The following sections first analyse the main principles of legal knowledge modelling that have been developed since the 1980s within the field of AI and Law. Then they introduce the Declarative and Imperative paradigms, highlighting their main differences in terms of programming approach. Finally, the chapter provides an overview of the principal formal languages designed for the legal domain that constitute the current state of the art.

This chapter is therefore intended to serve as a foundation for Parts II and III of the dissertation.

3.1 Legal Rules as Code

In the 17th century, philosopher and mathematician Gottfried Wilhelm Leibniz (1646–1716) highlighted the logical structure present in Roman law, and started drawing parallels with propositional and modal logic [10]. He described legal conditions as composed of two parts: a hypothesis ("if", or *conditio*) and a conclusion ("then", or *conditionatum*), echoing the classical logical form of modus ponens.

If $p \implies q$ and p is true, then q must also be true.

This vision inspired early AI and Law researchers to formalise legal rules in logic to improve legal drafting and avoid unintended ambiguities [4, 5].

More recently, researchers have started using Logic Programming (LP) to develop computable representations of legal statutes and allow automated reasoning over them. A landmark attempt in this direction was presented in the paper “The British Nationality Act as a logic program” by Sergot et al. in 1986 [156]. Here, the authors argue that a substantial amount of statutory law is definitional, i.e., rules that specify the conditions under which a certain legal qualification applies [156, 97]. As such, statutory rules can be expressed as a combination of relations among entities in the form of a logical conditional, an *if, then* rule.

To this end, the authors employ the Prolog programming language which allows rules to be expressed as *Horn clauses*. For example, the provision “*A person born in the United Kingdom [...] shall be a British citizen if, at the time of birth, his father or mother is (a) a British citizen; [...].*” can be formalized as a pseudo horn clause form as follows:

```
X is a British citizen :-
    X was born in the U.K
    and Z is a parent of X
    and Z is a British citizen.
```

This structure consists of a head (before the `:-` symbol), and a list of conditions that must hold true in order to verify the head statement. When expressed in Prolog syntax and combined with a reasoning engine, such a representation enables the automated assessment of the rule on specific scenarios.

The work of Sergot and Kowalski remains a foundational reference for the formalisation of legal rules as code. Building on their contribution, subsequent research has evolved to identify additional elements that must be considered in order to achieve a faithful computational representation of the law.

3.1.1 Deontic Positions: Representing Normative Force

While early work focused on *definitional* or *constitutive* rules, such as those addressed in the encoding of the British Nationality Act [156], legal systems also

rely heavily on *prescriptive* or *regulative* rules. In this context, a widely accepted view in computational legal modeling is that *deontic modalities* [172] expressing obligations, permissions, and prohibitions are essential for representing normative aspects of the law.

Jones and Sergot [97] present the importance of incorporating deontic positions into formal legal representations, especially when modeling violations and the duties triggered by them, i.e., contrary-to-duty (CTD) obligations. To exemplify the cases where deontic concepts are required, they use the dichotomy between what is the ideal and actual: obligations (and other deontic notions) describe what is ideal (according to a normative system) and it is not necessarily accomplished in actual cases.

Therefore deontic operators are particularly useful when building systems intended either to enforce compliance with norms or to provide normative guidance to users in decision-making contexts. A substantial body of literature has further developed this approach, offering both foundational insights and advanced techniques for incorporating deontic logic into computable legal models [83, 128, 90, 6].

3.1.2 Defeasibility of Legal Reasoning

As Sartor notes [150, 151], legal systems are full of exceptions, special cases, and competing principles, so that adding a new premise to a case may activate an exception clause and reverse a previous decision. This characteristic of law is called *defeasibility*.

Accordingly, any faithful computable representation shall be able to accommodate [151] the expression of rule exceptions, conflicts, and priority relations among competing norms.

To this end, researchers in computational law have explored a range of approaches aimed at faithfully modeling these features [151, 79, 8, 30]. Some approaches extend general-purpose logic programming environments—such as Prolog and s(CASP)—by introducing meta-level constructs for deontic and defeasible reasoning. Others have developed domain-specific languages with built-in support for legal reasoning constructs, explicitly encoding normative positions, defeasible rules, and rule hierarchies [110, 76, 154].

3.1.3 Isomorphism and Validation

Another significant contribution to the computational modeling of legal knowledge bases is the concept of *isomorphism*, introduced by Bench-Capon in 1992 [15]. This concept refers to the need for a clear and structured correspondence between specific portions of the legal source and their computable representations. One of the primary motivations behind this idea is to facilitate *maintenance*—a critical concern given the frequency of legislative amendments. Isomorphic design helps address a longstanding limitation of knowledge-based systems: their tendency to become outdated or inconsistent as legal texts evolve.

This insight also highlights a broader challenge in the computable representation of law: the intrinsic tension between three core requirements of a legal knowledge base. Ideally, a computable model should: (1) be a *faithful* representation of the legal text; (2) be *computationally adequate*, enabling all relevant inferences to be derived by machine; and (3) be *maintainable* and *validatable*, i.e., easy to update and verify. However, efforts to satisfy one of these requirements often undermine the others. For example, increasing the semantic richness of a representation may improve faithfulness but make it harder to reason over automatically. Likewise, complex or opaque representations may hinder validation and ongoing maintenance [147].

3.2 Imperative and Declarative Paradigms: Design Principles

Imperative and declarative programming represent two distinct ways of conceptualising and implementing computation. While imperative programming dominates most industrial and commercial applications because of its straightforward logic, fine-grained control of execution, and ease of debugging [143, 98], declarative programming remains central in fields such as knowledge representation, databases, and automated reasoning. In particular, declarative approaches have played a key role in legal knowledge modelling, as discussed in the previous section [163, 154, 9, 76]. Understanding their core characteristics is essential to evaluate their suitability for modelling legal agreements.

3.2. IMPERATIVE AND DECLARATIVE PARADIGMS: DESIGN PRINCIPLES

Imperative programming languages are built around the notion of a program as a *sequence of instructions or procedures* executed by the machine to achieve a specific outcome [155]. The programmer specifies not only the desired result but also *how* each step should be carried out to reach it. This approach provides explicit control over execution flow and system behaviour, typically structured around functions or methods, which makes it intuitive for developers. However, imperative programs tend to be verbose and detached from high-level specifications, and their correctness is usually established through testing rather than formal proofs [1, 155].

Declarative programming languages, by contrast, describes *what* the program should accomplish without prescribing the steps to achieve it [155, 114]. Instead of a sequence of commands, a declarative program defines relations among entities, leaving the task of deriving outcomes to inference mechanisms or querying systems. The focus is thus not on controlling the execution flow but on specifying the *truth of statements* and letting the system determine how these can be satisfied [108].

A key distinction between the two paradigms concerns the notion of *symmetry*. Imperative languages are inherently *asymmetric*, as instructions define a one-way flow from inputs to outputs. For example, in Solidity the function:

```
function add(uint a, uint b) public pure returns (uint) {  
    uint c = a + b;  
    return c;  
}
```

computes the value of `c` given `a` and `b`, but not the reverse. To derive the values of `a` or `b` from a given `c`, one would need to define a separate inverse function.

Declarative programming, on the other hand, is often *symmetric*, since the same rule can be used both to compute results from inputs and to infer the inputs that satisfy a given result [108, 114]. This bidirectionality allows for a more flexible style of programming, particularly suited to reasoning tasks. In Prolog, for instance, addition can be defined declaratively using the Peano representation of natural numbers¹.

¹Giuseppe Peano (1858–1932) was an Italian mathematician who formulated the *Peano axioms*, a formal foundation for the natural numbers based on a constant 0 and a successor function $s(\cdot)$ [138].

3.2. IMPERATIVE AND DECLARATIVE PARADIGMS: DESIGN PRINCIPLES

```
sum(A, 0, A).  
sum(A, s(B), s(C)) :-  
    sum(A, B, C).
```

Here numbers are constructed from the constant 0 and the successor function $s/1$ (e.g., 0 represents zero, $s(0)$ one, and $s(s(0))$ two) [61]. The rules do not specify a sequence of steps but a relation among the three arguments of the predicate `sum/3`. As a result, the same definition can be used in different directions: a query such as `?- sum(s(s(0)), s(0), X).` (i.e., `sum(2,1,X)`) will infer $X = s(s(s(0)))$ ($X = 3$), while `?- sum(X, s(0), s(s(s(0)))).` (i.e., `sum(X,1,3)`) will return $X = s(s(0))$ ($X = 2$). The rule is therefore *symmetric*: it defines a relation that can be verified or explored from multiple perspectives rather than encoding a one-way computation. This property also explains why declarative approaches are widely used in database systems, where data can be queried in many different ways [155, 11].

In summary, imperative programming prioritises operational clarity, efficiency, and explicit control of execution, whereas declarative programming provides a high-level, semantically transparent means of representing and reasoning about knowledge. This distinction also shapes the way in which *computable contracts* are designed and implemented.

3.2.1 Imperative vs Declarative Computable Contracts

Most computable contracts deployed today are developed using *imperative programming languages*², particularly within the blockchain ecosystem³. Imperative approaches are preferred because they allow explicit control of execution and state management—essential features in environments where deterministic behaviour and performance are critical.

A preliminary analysis of the use of *declarative, logic-based languages* for computable contracts is provided by Governatori et al. [78], with specific reference to blockchain-based smart contracts. The authors argue that declarative, logic-based

²See [170]; Etherisk <https://etherisc.com/>; former Axa Fizzy <https://www.businessinsider.com/axa-turns-to-smart-contracts-for-flight-delay-insurance-2017-9>

³For instance, Solidity on Ethereum, Java and Go on Hyperledger, and PyTeal on Algorand.

formalisations offer several advantages. They enable a more direct alignment between the structure and semantics of legal norms, and are often more intelligible to legally trained practitioners. This facilitates both transparency and the formal verification of compliance, since contractual conditions can be expressed in a form closer to normative reasoning. At the same time, the study acknowledges that declarative languages face practical challenges: limited familiarity among developers and potential computational overhead when executed in resource-constrained distributed systems.

3.3 Domain-specific languages for Legal Contracts

Because of the specific needs that the representation of a computable contracts requires, instead of relying on general purpose languages such as Solidity, Java or Prolog mentioned above, relevant scientific efforts have been put in the development of *domain-specific languages*.

In this direction, several formal languages differing in terms of design approach, level of abstraction and target domain. Below, some of the most relevant proposals are outlined.

A first approach has focused on *markup languages* and standardisation initiatives that allow for the representation of legal documents in a structured, machine-readable form, following the ricardian concept Section 2.1.2. Among these, *Legal-RuleML* [12] extends the *RuleML* framework by incorporating legal-specific constructs such as temporal validity, jurisdiction, and defeasibility. It provides a standardized way to encode legislation and contractual clauses for reasoning and interoperability across legal information systems.

On a similar vein, the *Accord Project*⁴ provides a comprehensive framework for drafting computable contracts starting from natural language clauses. Natural language templates are associated with a corresponding template of executable logic written in the domain-specific language *Ergo*. The two templates are connected through a shared data model (the *parameters* of the Ricardian architecture) which instantiates the contractual variables.. and the two templates are linked through a

⁴accordproject.org

data model which instantiate the templates. At present, the framework’s compiler targets Java and JavaScript⁵. Although technically advanced and well-structured from an implementation perspective, the framework does not explicitly capture the legal semantics of contractual terms within any of its layers.

A different line of research has explored the design of *domain-specific specification languages*. Such languages provide an abstract, implementation-independent means of describing a contract’s intended behaviour, supported by a rigorous semantics that enables validation of specific properties and formal reasoning. In this family, *Symboleo* [137] represents a prominent example. It adopts a temporal logic framework to model contractual obligations, powers, and permissions, and provides a runtime environment implemented in *Prolog* for formal verification. Two kinds of analysis are supported: (i) the verification of the contract’s state after a given sequence of events, and (ii) the checking of liveness and safety properties via model checking tools. Although *Symboleo* offers rigorous mechanisms for consistency analysis and has been applied to compliance modelling in data supply chains [13], its focus remains on verification and monitoring rather than on direct automation of contractual performance.

A conceptually similar yet architecturally distinct approach is offered by the *Contract Specification Language (CSL)* developed by Deon Digital⁶. Unlike most blockchain-based smart legal contract frameworks, CSL separates the logical and operational dimensions of a contract. It distinguishes between three conceptual layers: (1) the *contract logic*, specifying rights, obligations, and conditions; (2) a *control layer*, which monitors compliance; and (3) a *resource layer*, managing the transfer of assets and other external effects⁷. This design promotes modularity and semantic clarity by delegating the execution of resource transactions to external managers. While CSL provides a mature and well-structured framework for digital contract specification, its execution model depends on external components, thus limiting its ability to realise fully autonomous contracts.

Another significant contribution comes from *Rosetta DSL*⁸, developed within the International Swaps and Derivatives Association (ISDA) to formalise and au-

⁵docs.accordproject.org/docs/accordproject.html

⁶www.deondigital.com/technology/

⁷docs.deondigital.com/latest/

⁸rosetta-technology.io/

tomate financial contracts. Rosetta provides a unified data and process model that allows the unambiguous translation of contractual terms into executable specifications for financial instruments. Its scope remains centred on derivatives and financial products and thus less generalisable to broader categories of legal agreements.

Still in the financial sector, Flood and Goodenough [67] propose a pre-linguistic model to express financial contracts, namely a Deterministic Finite Automaton (DFA). In this framework, a contract is described as a system of *states*, and *transitions rules*, that describe how the contract moves from one state to another during the lifecycle.

Other contributions have focused instead on improving the accessibility and readability of legal programming languages, seeking to make formal representations directly intelligible to non-technical users. The *L4 language* [176] exemplifies this trend. It is a domain-specific language for the legal domains that combines a rule-based logical structure with a quasi-natural syntax. L4 and related projects such as *Catala* [119] aim to capture the reasoning patterns of legal texts while maintaining a syntax that remains close to natural language. Its design promotes clarity and traceability, although their adoption in operational settings is still limited.

Finally, *Lexon* ⁹ adopts a radically different strategy based on a *controlled natural language* (CNL). Lexon seeks to make smart contracts directly readable by non-programmers, using a constrained subset of English that can be compiled into blockchain code. This approach lowers the barrier to entry for legal practitioners and enhances intelligibility, but it also introduces ambiguity risks due to its quasi-natural syntax [92]. In practice, the system largely acts as syntactic sugar for Solidity, since no formal semantics has been defined to guarantee the alignment between legal meaning and code behaviour. As with other CNL-based initiatives, its expressiveness is limited by the trade-off between readability and formal precision.

In summary, the current landscape of computable contract languages shows a wide spectrum of approaches: declarative, logic-based specification frameworks, template-driven environments, and controlled natural languages. Each of these approaches contributes valuable insights into the representation and operational-

⁹lexon.org

isation of legal meaning, yet none fully reconciles legal semantics, executability, and accessibility. The following chapter builds upon these efforts to introduce a domain-specific language that aims to integrate these dimensions within a unified, executable framework.

3.4 Key Findings

This chapter has outlined the main foundational approaches to the development of computable contracts. Early research in Artificial Intelligence and Law demonstrated that formal logic provides a powerful basis for modelling legal reasoning, but also revealed the limitations of purely declarative representations when applied to dynamic and operational contexts. The subsequent analysis of imperative and declarative paradigms highlighted their distinct design principles and their respective relevance for computable contract modelling.

As discussed, the current landscape encompasses a wide spectrum of programming approaches for legal agreements, including declarative, logic-based specification frameworks, template-driven environments, and controlled natural languages. Each of these contributes valuable insights into the representation and operationalisation of legal meaning, yet none fully reconciles legal semantics, executability, and accessibility within a single framework.

Building on these findings, the next chapter introduces *Stipula*, a domain-specific language designed to integrate these dimensions—linking legal abstraction and computational execution—into a coherent, executable model of computable contracts.

Part II

Stipula DSL

Chapter 4

Stipula Foundations

This chapter brings *Stipula*, a domain-specific language designed for drafting computable contracts. Its development follows a structured methodology that begins with the identification of simple legal patterns, proceeds with their analysis, and culminates in the design of computationally faithful implementations. As a result, *Stipula* is built around primitives and features that capture directly concepts of contract law, such as agreements, obligations, or permissions. Its aim is to work as an intermediate domain-specific language: more concrete than specification language, and with a higher level of abstraction than a full-fledged programming language [47].

In the following sections, the main primitives are presented together with the legal patterns they are intended to represent, as well as the corresponding formal syntax and semantics.

4.1 Patterns of Legal Contracts

Stipula is based on a few main primitives that abstractly capture the core building blocks of contracts. These are presented below together with their specific syntax. To allow a clearer reading, the explanation is integrated with brief code examples.

The formal syntax of a *Stipula* contract is presented in Table 4.1. In the syntax, the parts in brackets $[\dots]$ are optional.

Stipula uses different types of names to indicate different entities: C , C' , $\dots$

<pre> stipula C { assets $\bar{h}$ fields $\bar{x}$ agreement($\bar{A}$){ $\bar{A}_1 : \bar{x}_1$... $\bar{A}_n : \bar{x}_n$ } $\Rightarrow @Q$ F } </pre>		$// \bigcup_{i \in 1..n} \bar{A}_i \subseteq \bar{A}, \quad \bigcup_{i \in 1..n} \bar{x}_i \subseteq \bar{x}, \quad \bigcap_{i \in 1..n} \bar{x}_i = \emptyset$	
<i>Functions</i>	$F ::=$	$-$	$ @Q A : f(\bar{y})[\bar{k}](E)\{S W\} \Rightarrow @Q' F$
<i>Prefixes</i>	$P ::=$	$E \rightarrow x \mid E \rightarrow A \mid E \multimap h, h' \mid E \multimap h, A$	
<i>Statements</i>	$S ::=$	$- \mid P S \mid \text{if}(E)\{S\} \text{else}\{S\} S$	
<i>Events</i>	$W ::=$	$- \mid E \gg @Q\{S\} \Rightarrow @Q' W$	
<i>Expressions</i>	$E ::=$	$v \mid X \mid \text{now} \mid E \text{ op } E \mid \text{uop } E$	
<i>Values</i>	$v ::=$	$n \mid \text{false} \mid \text{true} \mid s \mid \mathbb{t}$	

 Table 4.1: Syntax of *Stipula*

denote *contract names*; $A, A', \dots$ denote *parties*; $f, g, \dots$ 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 $x, y, \dots$; the latter ones are stored in *assets* denoted by $h, k, \dots$. Finally, we will use $Q, Q', \dots$, to range over contract *states*. To simplify the syntax, we often use the vector notation $\bar{x}, \bar{h}, \bar{A}$ to denote possibly empty sequences of elements.

A contract in *Stipula* has a name, which is C in Table 4.1; its body contains the *assets* and *fields*, the *agreement code*, and the sequence F of functions.

Agreement. A contract enters into force at the moment of the ‘meeting of the minds’ of the parties. 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 binding¹. This is represented by the *agreement* operator through which parties are asked to agree on the terms of the

¹Depending 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).

contract.

This element is represented by the `agreement` constructor². This specifies the set $\bar{A}$ of involved parties and the initial values of a subset of the contract's *fields* (i.e., certain terms of the contract). These do not include assets, that are initially empty. The agreement also specifies the initial state of the contract. For example, the agreement

```
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`.

Permissions and Prohibitions Legal contracts may create, extinguish or regulate the parties' normative positions such as permissions, prohibitions or obligations.

The actions that parties are able to take are represented by the functions in F (see Table 5.1). In particular, every function defines the caller party A (indicating the party that is enabled to invoke it) and a state Q when the invocation is admitted. *States* are used to represent the different stages of the contract life cycle, e.g., initial activation, the payment of instalments, the occurrence of a breach etc. At each state only certain functions may be invoked while others are precluded. In this way, states provide a mechanism to model and automatically enforce prohibitions and permissions. In the foregoing code, the state enabling the invocation of the `update_price` function by `PriceProvider` is `@Start`.

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

Moreover, records the value p in the field `price` and sends it to the parties `Supplier` and `Buyer`. After these operations, the contract transits to the state

²A constructor is a special type of method used in object-oriented programming languages to initialize objects. The constructor is called automatically every time when an object is created, allowing the object to set initial values for its attributes or perform other setup tasks [23].

Waiting_order.

Assets and Fields As shown in Table 4.1 (F definition), functions' parameters are split in two lists: *formal parameters* $\bar{y}$ in brackets and *asset parameters* $\bar{k}$ in square brackets. The *precondition* E constrains the execution of the body of f . The *body* $\{ S W \} \Rightarrow @Q'$ specifies the *statement part* S , the *event part* W , and the state Q' of the contract when the function execution terminates. The *statements* S include the empty statement $_$ and different prefixes followed by a continuation. Prefixes P use the two symbols $\multimap$ and $\rightarrow$ to differentiate operations on assets and on fields, respectively.

In practice, computable contracts frequently need to manage currencies or digital goods. In *Stipula*, these entities are identified by *assets*, and operations involving them (such as transfers or escrows) are expressed through dedicated syntactic primitives. This specific design choice guarantees that assets cannot be arbitrarily created or destroyed, but only transferred between parties (see Section 4.2). The prefix $E \multimap h, h'$ subtracts the value of E to the asset h and adds it to h' , $E \multimap h, A$ subtracts the value of E to the asset h and transfers it to A . The operational semantics will prevent assets with negative values. We will always abbreviate $h \multimap h, h'$ and $h \multimap h, A$ (which are very usual, indeed) into $h \multimap h'$ and $h \multimap A$, respectively.

In contrast to assets, *fields* store standard values (e.g., price, deadline) and are not diminished by the transferred value. For example, the prefix $E \rightarrow x$ updates the field or the parameter x with the value of E .

Statements also include a *conditional* $\text{if } (E) \{ S \} \text{ else } \{ S' \}$ with the standard semantics.

Obligations Obligations in *Stipula* are captured in a different way compared to permission and prohibitions. While the latter are enforced indirectly through states, obligations rely on a more direct modelling, through *Events*, W in Table 4.1. 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 $E \gg @Q \{ S \} \Rightarrow @Q'$ 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 following operation (that is defined inside the body of a function)

```
now + time_due >> @Payment_Due{
  10  $\multimap$  wallet, Seller
} $\Rightarrow$  @Breach
```

models the obligation to pay within a certain deadline. The event triggers a transition at time `now + time_due` that may take place if, at that time, the contract is in the state `Payment_Due`. In particular, the event deducts a penalty fee of €10 from the contract wallet and transfers it to the Seller. After the operation the contract transitions to the state `@Breach`

Expressions E include constant values ranged over by $v, u, \dots$, 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 t that represent the global system clock and are expressed in the format “2025/1/1:00:15”, with minute-level precision.

4.2 Semantics

In contrast to other general purpose programming languages used for computable contracts, (e.g., Solidity, Java, Go), Stipula relies on precise semantics, mathematically defined. This means that the behaviour of a Stipula program can be anticipated with mathematical precisions and this allows reasoning on it, such as proving certain properties, analysing its content and ensure it is consistent with certain properties.

A Stipula contract can be seen as a finite-state automata in the sense of [67] (see Chapter 3). As such, at any point in of its lifecycle, a Stipula contract is represented by a *configuration*, i.e., a complete mathematical description of the system, including its current state and all the information required to determine

its subsequent evolution. A Stipula configuration is expressed by the tuple $\mathbb{D} \vdash_{\mathfrak{t}} \mathbb{Q}, \ell, \Sigma, \Psi$ where

- $\mathbb{D}$ is the declaration of the configuration.
- $\mathfrak{t}$ is the time value of the system's global clock;
- $\mathbb{Q}$ 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 $A, A', \dots$ (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 $S \Rightarrow \mathbb{Q}$. We assume that $_ S \Rightarrow \mathbb{Q}$ is equal to $S \Rightarrow \mathbb{Q}$;
- 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

$$\mathbf{k}_1 \gg \mathbb{Q}_1 \{ S_1 \} \Rightarrow \mathbb{Q}'_1 \mid \dots \mid \mathbf{k}_h \gg \mathbb{Q}_h \{ S_h \} \Rightarrow \mathbb{Q}'_h$$

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

The semantics of a Stipula contract is defined by a set of rules describing how the system evolves from one configuration to another. These are the transition rules or *transition relations*. The general transition relation in *Stipula* is

$$\mathbb{D} \vdash_{\mathfrak{t}} \mathbb{Q}, \ell, \Sigma, \Psi \xrightarrow{\mu} \mathbb{D} \vdash_{\mathfrak{t}'} \mathbb{Q}', \ell', \Sigma', \Psi'$$

where,

$$\mu ::= _ \mid (\overline{A}, \overline{A_1} : \overline{v_1}, \dots, \overline{A_n} : \overline{v_n}) \mid A : \mathbf{f}(\overline{u})[\overline{v}] \mid v \rightarrow A \mid v \multimap A.$$

That is, the label μ identifies the specific occurrence that affects the contract configuration. This is either empty, or denotes an initial agreement, or a function call, or a value send, or an asset transfer. For example, the initial state of a contract, before it is initialised by the invocation of the **agreement constructor** is represented by the following empty configuration

$$- \vdash_{\mathbb{E}} -, \emptyset, -, -$$

where no state, memory ℓ pending event or residual function body is present. The invocation of the agreement triggers a transition that is defined by the following relation:

$$- \vdash_{\mathbb{E}} -, \emptyset, -, - \xrightarrow{(\bar{A}, \bar{A}_i; \bar{v}_i^{i \in 1..n})} F \vdash_{\mathbb{E}} \mathbf{Q}, \ell, -, -$$

The formula above the arrow specifies the operation that triggers the transition—namely, the invocation of the agreement. During this step, the actual parties $(\bar{A}, \bar{A}_1)$ instantiate the contract fields with concrete values $(\bar{v}_1)$. The resulting configuration $(\mathbb{D})$ represents the newly initialised contract, which is now in state $\mathbf{Q}$ with memory ℓ . In particular, the memory is defined as follows

$$\ell = [\bar{\mathbf{A}} \mapsto \bar{A}, \bar{\mathbf{x}}_i \mapsto \bar{v}_i^{i \in 1..n}, \bar{\mathbf{h}} \mapsto \bar{0}]$$

The set of formal parties $\bar{\mathbf{A}}$ (e.g., *seller* and *borrower*) is instantiated with the corresponding actual parties, denoted in italics as $\bar{A}$ (e.g., *Alice* and *Bob*). Likewise, the set of formal fields $\bar{\mathbf{x}}_i$ is instantiated with the actual values $\bar{v}_i$. Finally, the assets $\bar{\mathbf{h}}$ are initialised as empty (0).

For each operation that can be performed through a *Stipula* contract, a specific transition relation is defined (update of an asset, of a field, the invocation of a function, triggering of an event and so on). For the definition of these rules, two auxiliary functions are used.

One is the *evaluation function*. This is used to evaluate expressions E , e.g., mathematical operations used to subtract a certain amount from an asset, to calculate a deadline and so on. The function $\llbracket E \rrbracket_{\ell}$ returns the value of E in the memory ℓ . For example, the update of a given contract field x , is defined by the

relation

$$\frac{[\text{FIELD-UPDATE}] \quad \llbracket E \rrbracket_\ell = v \quad \ell' = \ell[x \mapsto v]}{\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, E \rightarrow x \Sigma, \Psi \longrightarrow \mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell', \Sigma, \Psi}$$

In the lower part of the rule, the memory ℓ is updated to ℓ' between the two configurations. Specifically, the field x is updated with the value v , which corresponds to the evaluation of the expression E under the current memory ℓ , i.e., $\llbracket E \rrbracket_\ell$.

The second auxiliary function is the *selection operation* $\mathbb{D}[\text{@Q A} : \mathbf{f}]_{\ell, \bar{u}, \bar{v}}^{\mathbb{t}}$. For each function invocation, this operation returns a set of functions that satisfy the function guard E (see Table 4.1). Such a guard is always defined as an expression and thus evaluated through the evaluation function.

Stipula also features an internal clock. In the configuration, $\mathbb{t}$ represents the current point in time. This internal clock advances only when pending events are present. Specifically, the progression of time (a *tick*) is governed by the predicate $\Psi, \mathbf{Q} \nrightarrow$, which holds true only when there are pending events in Ψ . When a tick occurs, the corresponding transition relation is:

$$\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, -, \Psi \longrightarrow \mathbb{D} \vdash_{\mathbb{t}+1} \mathbf{Q}, \ell, -, \Psi$$

The rest of the transition relations are presented in Table 4.2. As regards statements, the 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 $1234\mathbf{D} \rightarrow A$ is admitted: it means that A will receive the **real** value **1234** (*c.f.* the meaning of $\llbracket E \rrbracket_\ell$).

Rule [ASSET-SEND] transfers part of an asset $\mathbf{h}$ to the party A . This part corresponds to the value of $\llbracket E \rrbracket_\ell$; the premise $0 \leq \llbracket \mathbf{h} - E \rrbracket_\ell \leq \llbracket \mathbf{h} \rrbracket_\ell$ constrains the value of E to be positive and smaller or equal to the value of $\mathbf{h}$, so as to ensure that the asset stores the minimum amount enough to be transferred. For example, if $\mathbf{h}$ holds $20\mathbf{D}$ then $25\mathbf{D} \multimap \mathbf{h}, A$ would not be executed, while $\mathbf{h} \times 0.5 \multimap \mathbf{h}, A$ would remove $10\mathbf{D}$ from $\mathbf{h}$ and send them to A . In a similar way, [ASSET-UPDATE] moves a part E of an asset $\mathbf{h}$ to an asset $\mathbf{h}'$. consequently the both assets $\mathbf{h}$ and $\mathbf{h}'$ are updated accordingly.

4.2. SEMANTICS

$ \begin{array}{c} \text{[VALUE-SEND]} \\ \hline \frac{\llbracket E \rrbracket_\ell = v \quad \ell(\mathbf{A}) = A}{\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, E \rightarrow \mathbf{A} \Sigma, \Psi} \\ \xrightarrow{v \rightarrow A} \mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, \Sigma, \Psi \end{array} $	$ \begin{array}{c} \text{[ASSET-SEND]} \\ \hline \frac{\ell(\mathbf{A}) = A \quad 0 \leq \llbracket \mathbf{h} - E \rrbracket_\ell \leq \llbracket \mathbf{h} \rrbracket_\ell \quad u = \llbracket \mathbf{h} - E \rrbracket_\ell \quad v = \llbracket \mathbf{h} - u \rrbracket_\ell}{\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, E \multimap \mathbf{h}, \mathbf{A} \Sigma, \Psi} \\ \xrightarrow{v \multimap A} \mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell[\mathbf{h} \mapsto u], \Sigma, \Psi \end{array} $
$ \begin{array}{c} \text{[ASSET-UPDATE]} \\ \hline \frac{0 \leq \llbracket \mathbf{h} - E \rrbracket_\ell \leq \llbracket \mathbf{h} \rrbracket_\ell \quad \llbracket \mathbf{h} - E \rrbracket_\ell = u \quad \llbracket \mathbf{h}' + E \rrbracket_\ell = u'}{\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, E \multimap \mathbf{h}, \mathbf{h}' \Sigma, \Psi} \\ \longrightarrow \mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell[\mathbf{h} \mapsto u, \mathbf{h}' \mapsto u'], \Sigma, \Psi \end{array} $	
$ \begin{array}{c} \text{[COND-TRUE]} \\ \hline \frac{\llbracket E \rrbracket_\ell = \mathbf{true}}{\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, \text{if}(E) \{ S \} \mathbf{else} \{ S' \} \Sigma, \Psi} \\ \longrightarrow \mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, S \Sigma, \Psi \end{array} $	$ \begin{array}{c} \text{[COND-FALSE]} \\ \hline \frac{\llbracket E \rrbracket_\ell = \mathbf{false}}{\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, \text{if}(E) \{ S \} \mathbf{else} \{ S' \} \Sigma, \Psi} \\ \longrightarrow \mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, S' \Sigma, \Psi \end{array} $
$ \begin{array}{c} \text{[STATE-CHANGE]} \\ \hline \frac{\llbracket W \{ \mathbb{t} / \mathbf{now} \} \rrbracket_\ell = \Psi'}{\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, -W \Rightarrow \mathbf{Q}', \Psi \longrightarrow \mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}', \ell, -, \Psi' \mid \Psi} \end{array} $	
$ \begin{array}{c} \text{[EVENT-MATCH]} \\ \hline \frac{\Psi = \mathbb{t} \gg \mathbb{Q} \{ S \} \Rightarrow \mathbb{Q}' \mid \Psi'}{\mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, -, \Psi \longrightarrow \mathbb{D} \vdash_{\mathbb{t}} \mathbf{Q}, \ell, S \Rightarrow \mathbf{Q}', \Psi'} \end{array} $	

Table 4.2: The transition relations of *Stipula*

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 .

4.3 Integrated Environment

Stipula has been prototyped, therefore one can experiment enforcement of contractual conditions, traceability, and outcome certainty. The prototype is a Java application that is available on the github website of the project, together with a number of sample contracts [48]. The development has taken three months and ~ 3000 lines of Java code. In order to use it, it is necessary to clone the repository and install the prototype following the instructions provided on the website. Once installed, it is possible to write a brand new contract and run it to test its execution.

The prototype is also provided with the *higher-order Stipula* extension, supporting runtime amendments. At the implementation level, this functionality is achieved by updating the contract instance with changes contained in the *higher order*. Once the changes are overwritten, the new version is deployed. A technical explanation of the prototype design is contained in [47].

Although *Stipula* features an intuitive syntax, the project is provided with a user-friendly graphical interface to guide the drafting. This is clearly useful for first-time users as well as for users that are already familiar with the language in that avoids potential syntactic mistakes.

Chapter 5

Amendments

The language presented so far defines *Stipula* contracts that are immutable, i.e., they cannot be modified at runtime once the execution has been started. However, there may be several reasons to modify a contract, ranging from the fact that the parties have changed their mind, to the occurrence of an unexpected event that affects the contractual relationship. The latter case is usually dealt with in contracts by *hardship* clauses, and represents a crucial issue, especially in long-term commercial agreements.

In the current version of the language, in order to model contract amendments, one would have to anticipate the potential modifications caused at the time of contract formation, and provide for them accordingly, in the contract code. Besides being hardly feasible, such a practice would significantly raise the drafting costs, thus nullifying one of the main purposes for digitalizing legal contracts. Therefore, in the present work, we discuss the addition of a new feature to *Stipula* in order to support for future amendments during contract execution. This extension, called *higher-order Stipula*, features functions that may carry computer code as an input parameter. The code is run when the function is invoked, thus possibly modifying the original contract protocol. This solution allows us to manage situations where the amendment affects the whole body of the contract, so that only the new provisions are operational, as well as situations where only some parts are changed, so that the rest of the previous code remains operational.

In the following sections, we first analyse the most common scenarios requiring

amendments, and discuss the legal basis for amending contracts in legal systems. In Section 5.3, we test the new feature on a real-world example directly taken from contract practice. In Section 5.5 we discuss two methods for restricting amendments. The first method requires the parties' agreement. With the second method, parties may pre-define constraints against which future amendments can be verified at run-time.

5.1 Amending Contracts

The principle of *freedom of contract* allows parties to modify contracts at their will, provided that there is an agreement and that the new content is not against the law. Occasionally, one party may yield to the other the power to change some parts of the agreement unilaterally [27]. This is a common practice for consumer contracts and standard terms of service, where the right to modify is usually tied to certain requirements, such as notifying the other party of the change and giving them the possibility to withdraw. In specific cases, the right to unilateral modification (*jus variandi*) may be directly conferred by the legislator (e.g., in Italian law, the employer's right to change employees' tasks ¹).

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

Contracts are entered into with the expectation that both parties will fulfil their obligations as agreed upon. The roman brocard *pacta sunt servanda* (agreement have to be respected), constitutes a foundational principle of contract theory: the contract is a mutual promise in which each party can hold the other one to the promised performance. However, parties accepted to be bound by those promises under the particular set of circumstances standing at the time of stipulation: if these circumstances change, this commitment may need to be revised. For exam-

¹Art. 2103 Codice Civile

ple, the beginning of a war could drastically raise the price of commodities needed for production or the outbreak of a pandemic could halt factories' activity. Such changes of circumstances may make performance of contractual obligations impossible, excessively onerous or even deprive the performance of its original utility for the counterparty. To address these situations, a legal basis to justify non-performance or to legitimately request an amendment of the contract is provided by the principle of *clausola rebus sic stantibus*: a contract is binding only as far as the relevant circumstances remain the same as they were at the time of conclusion of the agreement [181].

The matter is known to most legal systems but it is addressed in different ways. In common law systems, courts have elaborated the doctrine of *frustration*². Frustration represents an excuse when an unforeseen change in circumstances deprives the contract of all utility for one party, even though the material capacity to perform the obligation is not affected. In the United States, one can also find the notion of *impracticability* which, recognised by the Uniform Commercial Code (§ 2-615), offers a defense in case performance became impractical due to a contingency that the parties, at the time of stipulation, assumed would not take place. Unlike under *frustration*, impracticability also applies where performance has become extremely difficult or onerous for one party [133].

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

In contract practice, especially in international context, the eventuality of unexpected changes in circumstances that might affect the agreement is usually dealt with by specific clauses defining the conditions and procedures to be followed in such cases. By writing such clauses, parties can avoid the uncertainty of being at the mercy of the relevant national legislation and adjudication [68]. The main ex-

²The *frustration* doctrine was originally developed by English courts as a consequence to the famous *Coronation* cases in 1902-1904. The cancellation of King Edward VII's coronation frustrated the purpose of the defendants who leased apartments to witness the procession from a privileged spot. See *Krell v Henry* (1903).

amples in this sense are *force majeure* and hardship clauses. While *force majeure* occurs when performance becomes impossible and usually leads to suspension or termination, hardship cases take place where the equilibrium of a contract is altered making compliance significantly more onerous for one party³. In these cases, the burdened party is usually entitled to request an amendment of the contract, or its termination.

First-order *Stipula* does not provide ways to deal directly with modifications during its execution. In order to model either *force majeure* or hardship, one should anticipate all the appropriate amendments for each possible circumstance at the time of first drafting. While this is easy for termination clauses (it is enough to include a transition to a final state), it is clearly impossible for other kinds of amendments [120]. Even an attempt to do that would raise drafting costs and introduce huge complexities in the contract, thus nullifying one of the main objectives of *Stipula*, which is to have a simple and intelligible code. For these reasons, *Stipula* provides a specific feature to handle such situations, *higher-order Stipula* that allows parties to remove or amend the effects of a contract in a direct and intelligible way.

5.2 *Stipula* with Amendments

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

Technically, amendments are runtime adjustments to the contract's behavior. In programming languages, these runtime adjustment are usually expressed by *higher-order*, i.e., functions that may also take code as an input parameter. This

³Art. 6.2.2 UNIDROIT Principles

$$\begin{aligned}
 F &::= _ \mid @Q\ A : f(\bar{y})[\bar{k}](E)\{S\} \Rightarrow @Q'\ F \mid @Q\ A : F(\bar{X})\{H\}\ F \\
 P &::= E \rightarrow x \mid E \rightarrow A \mid E \times h \multimap h' \mid E \times h \multimap A \mid \text{if}(E)\{S\}\ \text{else}\{S\} \\
 S &::= _ \mid P\ S \\
 E &::= v \mid V \mid E\ \text{op}\ E \mid \text{uop}\ E \\
 H &::= (\text{remove } X)?\ (\text{add } X)?\ \text{run } X
 \end{aligned}$$

 Table 5.1: Syntax of light-weight *Stipula* and *higher-order Stipula*

code is run when the higher-order function is invoked, thus possibly modifying the function's behaviour. *higher-order Stipula* introduces *higher-order functions*. Its syntax and semantics are designed to faithfully capture the amending behaviour in contract practice. In the following sections, the formal syntax is introduced and then its functioning demonstrated on a running example. Finally the formal semantics is presented.

5.2.1 The Syntax of *H-O Stipula*

For simplicity, we present the higher-order extension on a *lightweight* version of the language presented in Chapter 4, without the *agreement clause* and *events*; this allows us to showcase the extension without unnecessary overcomplications.

The code of a *Stipula* contract is

$$\text{stipula } C \{ \text{parties } \bar{A} \quad \text{fields } \bar{x} \quad \text{assets } \bar{h} \quad \text{init } @Q \quad F \}$$

where C identifies the *contract name*; $\bar{A}$ are the *parties* that can invoke contract's functions, $\bar{x}$ and $\bar{h}$ are the *fields* and the *assets*, respectively, and the *initial state* is set to $@Q$. The contract body also includes the sequence F of functions, whose syntax is defined in Table 5.1 (the terms in black). In particular, *first-order* functions highlight who is the caller party A , the state $@Q$ when the invocation is admitted and the name of the function. The invocation has two lists of parameters: the *formal parameters* $\bar{y}$ in brackets and the *asset parameters* $\bar{k}$ in square brackets. The *precondition* E constrains the execution of the body; the *body* $\{S\} \Rightarrow @Q'$ specifies the *statement part* S and the state $@Q'$ of the contract when the function execution terminates.

The syntax of a *higher-order* function in *Stipula* is instead represented by the red parts of Table 5.1. In particular, we use function names ranged over $F, G, \dots$ (in general, function names that start with an upper-case letter). According to the formal syntax, a *higher-order* function in *Stipula* is

```
@Q Party: amendment ( $\langle X, Y, Z \rangle$ ) {remove  $X$  add  $Y$  run  $Z$ }
```

This function carries three parameters in brackets ($\langle \cdot \rangle$), whose roles are indicated by the directives **remove** X **add** Y **run** Z . X is a sequence of function names that will be removed from the contract (the terms of the sequence may be either f or $A:f$ or $Q A:f$); Y represents the new code added and may include declarations of new parties, fields, assets as well as new functions that will amend the contract; Z is the body of the higher-order functions. Therefore Z is defined in the form $\{..\} \Rightarrow @Q$, and may also include the new elements defined in Y . It is worth mentioning that, according to the syntax in Table 5.1, while X and Y are potentially empty sequences (i.e., optional parameters), Z is mandatory (it is, in fact, necessary to at least define the state that the contract will transit to after the function invocation).

5.3 Amendments in Practice

In order to showcase the practical effect of *higher-order Stipula* on the execution of a computable contract we will apply it to a real-world scenario.

Consider the following contract example directly derived from a contractual dispute arose before the Court of Arbitration of the International Chamber of Commerce⁴. The dispute concerned a long-term contract for the purchase of liquid natural gas. In order to set the cost of oil, parties define a price formula (that, in our case, for simplicity, is a multiplicative factor) and agree to refer to a specific pricing agency's publications to retrieve the necessary parameters (the market price, in our case). They also provide for a price revision clause regulating the procedures and conditions for updating the formula. Table 5.2 presents a simplified version of the agreement in its main points. The contract can be represented in *Stipula* as shown in Listing 5.1.

⁴ICC case n. 10351/2001

5.3. AMENDMENTS IN PRACTICE

1. **Agreement.** *Supplier* and *Buyer* stipulate a long term agreement for the supply of liquified natural gas. Parties indicate *Price Provider* as the reference price agency and fix a *price formula*.
2. **Price.** Gas price is calculated by applying the *price formula* to the price provided by *Price Provider*.
3. **Purchase.** *Buyer* can place an order for gas by paying in advance the corresponding price, as resulting from the *price formula*. *Supplier* shall deliver the gas ordered and payed without undue delay.
4. **Formula Revision.** Parties can, upon agreement, decide to revise the *price formula*, in order to match market needs.
5. **Amendment.** This Agreement may only be amended by a written instrument executed by each of the parties hereto.

Table 5.2: The Gas Supply contract in natural language

```
1 stipula GasSupply {
2   parties: Supplier, Buyer, PriceProvider
3   fields: price, formula, order
4   assets: wallet
5
6   agreement (Supplier, Buyer, PriceProvider) {
7     Supplier, Buyer : formula
8   } init => @Start
9
10  @Start,@Waiting_order PriceProvider: update_price(p)[] {
11    p -> price
12    p -> Supplier, Buyer
13  } => @Waiting_order
14
15  @Waiting_order Buyer: place_order()[w] {
16    w/(price*formula) -> order
17    order -> Supplier
18    w -> wallet
19  } => @Order
20
21  @Order Supplier: send_gas()[g] (g == order){
22    g -> Buyer
23    wallet -> Supplier
24  } => @Waiting_order
25
26  @Waiting_order Supplier,Buyer: formula_revision(f)[] {
27    f -> formula
28  } => @Waiting_order
29
30  @Waiting_order Seller,Buyer: amendment(X,Y,Z){remove X add Y run Z}
31 }
```

Listing 5.1: The Gas Supply contract in *Stipula*

Lines 1-4 define parties, fields, and assets of the contract called *GasSupply*. The contract includes the third party *PriceProvider* representing the reporting agency to which *Supplier* and *Buyer* refer to retrieve the necessary market data to calculate the (discounted) price. The *formula* stores the multiplicative factor to be applied to the market price; the field *order* stores the gas order made by

the buyer each time.

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

Finally, in order to cope with possible unforeseen circumstances, the parties include an amendment clause (n.5). This is represented by the **amendment** HO function in line 30. This can be called by both **Seller** and **Buyer** to introduce a modification. For the purposes of this section, we assume that parties find an agreement on the amendment outside the contract. We discuss in Section 5.5 how to constrain amendments in *higher-order Stipula*.

5.3.1 Additive Amendment

From the history of the dispute⁵ we can see that parties included in their contract a so-called take-or-pay clause. A *take-or-pay* clause is a provision in a contract stating that a buyer has the obligation of either taking delivery of goods from a Supplier or paying a specified penalty amount to the Supplier for not taking them. This kind of provision benefits both parties because reduces the risk of the

⁵Available at <https://tinyurl.com/Case10351>

`Supplier: amendment( $\varepsilon, \mathbb{D}, \{2y \rightarrow t\_thre \quad 100 \rightarrow threshold\} \Rightarrow @Restart)$`

where $\mathbb{D}$ is

```

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

```

Figure 5.1: Take-or-pay amendment

investment on the supplier side, and allows the buyer to negotiate a lower price⁶. For the purposes of our example, we can assume that an external event (*e.g.*, a war) affected the gas market as to increase the contractual risk on all market participants. Therefore, parties agree to introduce a take-or-pay clause to reduce their exposure for the following two years of their business relationship, providing a penalty in case the purchase threshold is not reached. To do so, **Supplier** invokes the **amendment** function as shown in Figure 5.1 from the top.

At this stage, all the function's parameters (**remove X add Y run Z**) are instantiated. The first parameter ε specifies that there are no functions to be removed from the old code. The second parameter, $\mathbb{D}$, defines a list of new functions.

⁶investopedia.com, available at <https://tinyurl.com/5xxjp997>

They generally reflect the original code presented in Listing 5.1 but with some additions necessary to implement the *take-or-pay* clause.

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

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

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

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

Notice that $\mathbb{D}$ introduces a whole new set of contract states and none of the new functions provide transitions to previous states from Listing 5.1. This means that the old code is completely deactivated, even though it is not definitively removed. In fact, it will be enough to invoke again the `amendment` function and make a transition to an old state to render it operational again. For example, after the two years negotiated by parties, they will be able to get to the old version of the contract. We call this kind of amendment *additive*: it introduces a whole new piece of protocol replacing the old one, and the two never overlap.

5.3.2 Overriding Amendment

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

In order to introduce such a possibility, the `amendment` function is invoked by either party as shown in the first line of Figure 5.2. Also in this case, there is no

```
Supplier, Buyer: amendment( $\varepsilon$ ,  $\mathbb{D}'$ , {"Provider_revision"  $\rightarrow$  ~
}  $\Rightarrow$  @Waiting_Order)
```

Where $\mathbb{D}'$ is:

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

Figure 5.2: Price-revision clause amendment

function to remove (the first argument is ε). The short body of the amendment function, the third parameter, is indicated directly in the first line. It simply communicates the title of the modification to all parties and makes a transition to the `Waiting_Order` state, already existing in the original contract.

In contrast to the previous example, here the new code $\mathbb{D}'$ does not substitute completely the old set of states, therefore old functions are kept operational. $\mathbb{D}$ introduces a new version of the `price_revision` function from Listing 5.1 and implements the possibility of calling a parties meeting in case the provider becomes unreliable. Having the two functions the same name, the same callers and same state, the new and the old `price_revision` function clearly overlap with each other and enter into conflict once they are invoked. *Higher-order Stipula* handles conflicts though priorities and by leveraging constraints. As a general rule, conflicts are resolved by giving priority to functions of the newest code. However, this can be

```
Buyer: define_Provider(update_price,  $\mathbb{D}''$ , {"New_Price_Provider"  $\rightarrow$  ~
}  $\Rightarrow$  @Waiting_order)
```

where $\mathbb{D}''$ is:

```
1 parties: NewPriceProvider = NewPriceProvider
2
3 @Waiting_order NewPriceProvider: update_price(p) [] {
4   p  $\rightarrow$  price
5   p  $\rightarrow$  Supplier, Buyer
6 }  $\Rightarrow$  @Waiting_order
```

Figure 5.3: Substitution of Provider

handled more smoothly by providing specific constraints that have to be satisfied. In this case, the newest `price_revision` requires the input `x` to be equal to the string `"dispute_provider"` (line 1). This means that, whether parties want to use that function to change provider, they will have to fulfill that condition, otherwise they will just call the old version from Listing 5.1 and simply modify the price formula. Once the contract has transited to the `@Meeting` state (line 3), parties are enabled to call the `define_provider` function. This is modeled as a *higher-order* one in that it is necessary to modify the code to add a new party to the contract.

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

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

As shown in Figure 5.3, the first parameter of `define_Provider` indicates `update_price`, meaning that the function is removed from the old code of Listing 5.1. At the same time, $\mathbb{D}''$ introduces the new price provider as a party to the contract and defines a new `update_price` function that can be only accessed by the

new provider. Lastly, the body of the function in line 1 communicates the name of the new provider to all parties and moves the contract to `@Waiting_Order`.

At this stage, the provisions regulating the parties' relationship results from the combination of the original agreement, plus the two following amendments. This is represented by the code from Listing 5.1, the code $\mathbb{D}'$ from Figure 5.2 and $\mathbb{D}''$ from Figure 5.3, which are all operational at the same time. Just like the previous one, this amendment does not introduce a new set of states: old and new code uptimes are preserved simultaneously. However, no overlaps occur. The old and new `update_price` differ as to the party that is able to invoke it. For this reason, the removal of the old function is necessary to prevent the old provider from interacting with the contract. We call this type of amendment *overriding with removal*.

5.4 The Semantics of *higher-order Stipula*

As mentioned above, we discuss the semantics of the *higher-order* extension on a lightweight *Stipula*, represented by the syntax in Table 5.1

In *higher-order Stipula* the declaration part of a configuration has the form $\mathbb{D} \triangleleft \mathbb{D}_1 \triangleleft \dots \triangleleft \mathbb{D}_n$, where $\mathbb{D}$ is the declaration of the initial contract and $\mathbb{D}_1, \dots, \mathbb{D}_n$ is a sequence of amendments. Amendments $\mathbb{D}_i$ have shape

$$\text{parties } \overline{A'} = \overline{A'} \quad \text{fields } \overline{z} \quad \text{assets } \overline{k} \quad F$$

This configuration admits the modification contract parties by initialising new parties names ($\overline{A'}$), like in the scenario presented in Figure 5.3.

The *selection operation* (presented in Section 4.2) is extended accordingly to include the declaration parts representing amendments' configurations ($\mathbb{D}'$ is a single amendment). In particular, the operation

$$(\mathbb{D} \triangleleft \mathbb{D}')[\text{QQ } \mathbf{A} : \mathbf{f}]_{\ell, \overline{u}, \overline{v}}$$

returns the newest set of functions in the list of amendments whose guard E is

true. Specifically

$$\begin{cases} \mathbb{D}'[\textcircled{Q} A : \mathbf{f}]_{\ell, \bar{u}, \bar{v}} & \text{if } \mathbb{D}'[\textcircled{Q} A : \mathbf{f}]_{\ell, \bar{u}, \bar{v}} \neq \emptyset \\ \mathbb{D}[\textcircled{Q} A : \mathbf{f}]_{\ell, \bar{u}, \bar{v}} & \text{otherwise} \end{cases}$$

This means that in case two identical function exist in the original contract code $\mathbb{D}$ and in the amendment code $\mathbb{D}'$, the newest function will be invoked.

When the function is higher-order, the selection returns $\mathbb{D}[\textcircled{Q} A : \mathbf{F}]_{\ell, \varepsilon, \varepsilon}$ (*i.e.* fields and asset parameters are empty). The transition relation triggered by the invocation of a *higher-order* function, is defined by the Rule [HO-FUNCTION]

Such a function is of the form $A : \mathbf{F}(\mathbf{P}, \mathbb{D}', B)$. $\mathbf{P}$ is a *sequence* of items $\mathbf{p}$ representing the list of functions that are to be deleted from the original contract code. These can be $\mathbf{f}$ or $\mathbf{F}$, $A : \mathbf{f}$ or $A : \mathbf{F}$, $\textcircled{Q} A : \mathbf{f}$ or $\textcircled{Q} A : \mathbf{F}$. $\mathbb{D} \setminus \mathbf{P}$ is the new version of the configuration $\mathbb{D}$ where the following protocol is erased:

- every function $\mathbf{f}$, if $\mathbf{p} = \mathbf{f}$;
- every function $\mathbf{f}$ that is invoked by A , if $\mathbf{p} = A : \mathbf{f}$;
- every function $\mathbf{f}$ that is invoked by A in a state $\textcircled{Q}$, if $\mathbf{p} = \textcircled{Q} A : \mathbf{f}$;
- similarly for higher-order functions.

The final configuration therefore represents the amended contract where the set of function $\mathbf{P}$ has been erased and where new protocol ($\mathbb{D}'$) was added. ℓ' is the new memory updated with the amendments' assets and fields and B is the protocol to be run within the HO function.

[HO-FUNCTION]

$$\frac{\begin{array}{l} \textcircled{Q} A : \mathbf{F}(X, Y, Z) \{ \text{remove } X \text{ add } Y \text{ run } Z \} \in \mathbb{D}[\textcircled{Q} A : \mathbf{F}]_{\ell, \varepsilon, \varepsilon} \\ \mathbb{D}' = \text{parties } \bar{A}' = \bar{A}' \text{ fields } \bar{z} \text{ assets } \bar{k} \text{ } F \quad \ell(A) = A \quad \ell' = \ell[\bar{k} \mapsto \bar{0}, \bar{A}' \mapsto \bar{A}'] \end{array}}{\mathbb{D} \Vdash \textcircled{Q}, \ell, - \xrightarrow{A : \mathbf{F}(\mathbf{P}, \mathbb{D}', B)} \mathbb{D} \setminus \mathbf{P} \triangleleft \mathbb{D}' \Vdash \textcircled{Q}, \ell', B}$$

Once the function $\mathbf{F}$ has been chosen, the actual arguments $\mathbf{P}$, $\mathbb{D}'$ and B' are used as follows: functions in $\mathbf{P}$ are removed from the declaration part $\mathbb{D}$, which is then amended with the code $\mathbb{D}'$ (provided this operation is well-defined, *c.f.* the

foregoing constraint about names in $\mathbb{D}'$) and the memory ℓ is updated with the binding of party names and the initialization of new asset names to 0. We observe that new fields are not initialized: in case, the initialization must be explicitly performed in the body B (*c.f.* the example in Figure 5.1).

5.5 Amendments Supplements: agreements and constraints

So far, one basic condition for modifying a legal contract was left out of the representation: the *mutual consent of the parties*. In fact, excluding the cases where the legislator confers to one party the power to modify the contract unilaterally, in all other cases, the manifestation of mutual agreement is required⁷. In order to deal with this principle within the contract, it is necessary to allow parties to express the consent to amendments at runtime.

The *Stipula* language already features an agreement clause which is triggered at the deployment of the contract and that corresponds to the “meeting of the minds”: every one must accept the terms of the contract in order for the legal bound to arise. By following the same pattern, the *higher-order Stipula* can provide such a constructor to be invoked whenever a HO function is called.

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

In order to specify and implement such possibilities, in [113] we have studied a technique for defining constraints for amendments. This allows parties to agree on the type of amendments they might accept in the future when a contract is

⁷Art. 2.1.1 UNIDROIT Principles

```
constraints [ (parties: fixed;)? (fields:  $\bar{z}$  constant;)?
              (assets:  $\bar{k}$  not-decrease;)? (reachable states:  $\bar{Q}$ )? ]
```

Figure 5.4: Static constraints on Amendments

stipulated. To this aim, we extend *higher-order Stipula* with the syntactic clause in Figure 5.4. This clause allows us to define an amendment verifier that automatically checks whether amendments comply or not with the restrictions in the clause.

Every constraint in the clause may be missing (when all the constraints are empty then “constraints []” is omitted and we are back to the basic syntax). The constraint “parties: fixed” specifies that amendments cannot modify the set of parties. If this constraint was present in the *gas supply* contract, then the amendment $\mathbb{D}'$ of Figure 5.3 would have been rejected. The constraint “fields: $\bar{z}$ constant” disables updates of fields in $\bar{z}$. The constraint “assets: $\bar{k}$ not-decrease” protects private assets to be drained by unauthorised parties. Finally, the constraint “reachable states: $\bar{Q}$ ” guarantees that, whatever contract update is performed, the states in $\bar{Q}$ can be reached from the ending state of the amendment. This is because, for example, the corresponding functionalities cannot be disallowed forever. The foregoing clause has not yet been integrated in the toolchain: the prototyping work is ongoing.

5.6 Key Findings

The development of *higher-order Stipula* demonstrates a mechanism to represent effectively and in a legally meaningful way contract amendments within computable contracts. By introducing higher-order functions capable of carrying executable code as parameters, *higher-order Stipula* enables contracts to be amended dynamically during execution, while maintaining the clarity and traceability of the original agreement.

Through the presented examples, we have shown that the new feature can model different amendment scenarios commonly found in legal practice, including complete substitutions of the contract protocol (additive amendments) and partial

modifications where old and new provisions coexist (overriding amendments). The formal semantics defined for *higher-order Stipula* ensures that these operations are both coherent and verifiable, preserving the logical consistency of the evolving contract.

Moreover, the language extension provides mechanisms to reflect essential legal safeguards, such as requiring mutual consent for amendments or enforcing predefined constraints that restrict how and when modifications can occur. These additions strengthen the correspondence between computational and legal reasoning, moving *Stipula* closer to the expressive and procedural richness of real-world contract law.

Chapter 6

Contract Analysis

The implementation of contractual provisions becomes problematic when such provisions do not clearly indicate the arrangements agreed upon by the parties. For example, open-ended, vague terms can engender uncertainties, possibly leading to legal disputes. However, these elements may be intentionally employed by the authors as a source of flexibility in the contract (e.g., to enable the parties to address future uncertainties or to conclude the contract even if not all details have been agreed upon) [35]. A different situation occurs when uncertainties are caused by the presence of *incompatible terms*, i.e., clauses expressing incompatible provisions. Their inclusion in a contract is rarely intentional and, when this happens, the joint application of such clauses is impossible or leads to absurd results. The reasons of incompatibilities can be multiple; for example, a misunderstanding or mistake at the time of drafting, or a mismatch between specifically agreed provisions and a standard contract template. Whatever the cause, incompatible terms almost never represent a feature, but rather a bug, leading to time-consuming disputes and, possibly, to highly expensive resolutions [56]. Therefore, it is crucial in legal practice to effectively identify incompatible clauses at the time of contract drafting. This applies to both contracts written in natural language, and to computable contracts consisting of executable code. Incompatible clauses expressed in natural language can often be addressed through legal interpretation, i.e., by assigning to the apparently incompatible clauses consistent meanings (if the parties can agree on such meanings). This is not possible in computable contract, whose clauses can

be automatically executed according to the semantics of the underlying computer language.

Recent advances in AI, Machine Learning, and NLP have greatly improved the capabilities to automatically analyse legal documents [34, 161], but they typically address issues like clause completion or drafting assistance rather than semantic incompatibilities [180, 111]. By contrast, notable contributions in this sense have been made through logic-based methods. These [64, 99] can detect contradictions through deontic or first-order logics, but depend on heavy formalisations and manual encoding. The closest contribution to our work is the Symboleo specification language [137], which allows the verification of contracts through model checking. This enables one to verify whether the execution of the contract matches the parties' intentions, based on predefined models of contractual behaviour. Although this approach is limited to the models defined for a specific contract, it shows that when contracts are expressed in formal languages, their precise syntax and semantics enable the application of powerful analysis techniques that have been developed and successfully applied in the domain of software verification.

In *Stipula*, we identified general code patterns indicating potentially incompatible clauses in *Stipula* contracts and proposed techniques to detect them at time of drafting. The patterns are categorised as *unreachability*, *non-determinism* and *security* issues. While not exhaustive, i.e., other cases of incompatible clauses may produce different patterns, these serve as indicators of potential incompatibilities when a contract is formalized in *Stipula*.

In what follows, a legal analysis of incompatible terms is provided in Section 6.1 and three illustrative legal cases are presented. In Section 6.2, we analyze those cases in *Stipula* and explore techniques for detecting incompatibilities.

6.1 Incompatible clauses as a source of problems

If every clause in a contract was precise and exhaustive, no dispute would ever arise around the correct application of the contract. However, reality is quite different. Typically, a contract may lack provisions addressing possible future circumstances that may occur during its life cycle; or feature provisions that are general and vague, thus requiring specifications to be applied. When this happens,

lawyers need to use interpretation to integrate or specify the contract (according to different rules or doctrines) and be able to apply the provisions to the particular scenario. Now, while it is true that these situations may induce conflicting interpretations among parties and, therefore, legal disputes, it is also true that drafters often intentionally leverage these elements – incompleteness, vagueness or open-endedness – as features to achieve various contractually-relevant objectives. These range from obtaining flexibility and adaptability, to reducing transaction costs during negotiations, and to enacting at the right time a precise economical strategy [18].

Another scenario that necessitates legal interpretation to be untangled occurs when a provision or group of provisions can generate multiple distinct interpretations that are individually clear and precise, yet incompatible with each other [63]. A clause is *incompatible* with a contract or part of it when (i) there is no situation in which it can apply or (ii) there are situations where it can apply together with other clauses (i.e., the conditions for each clause are satisfied), but their simultaneous application is impossible or leads to an absurd result. In such cases, the contract becomes problematic because its execution hinges on deciding which of the conflicting provisions to enforce.

Let us now illustrate a practical instance of incompatible clauses with the following fictional example, adapted from [99], and with two case laws.

Example 1. (*Bakery Sell* [99]) A contract for the purchase of a bakery includes a warranty that the bakery can bake 10,000 pretzels daily. The warranty defines the following procedure and deadlines: (1) the buyer must assert any breach within 14 days of contract closing; (2) the seller has 28 days to address the issue after a breach is asserted; (3) if the issue isn't resolved, compensation must be paid within 14 days otherwise a failure occurs; and (4) any claim under the warranty has a limitation of 42 days from the closing.

Deadlines (1), (2) and (3), add up to a total of 56 days from the closing of the contract to potentially claim a compensation. However this is clearly inconsistent with the general deadline of 42 days given in (4) and generates an ambiguity within the agreement. To resolve the conflict, a court could interpret provision (4) as a cumulative deadline, meaning that while all individual deadlines remain

valid, they must collectively fit within the 42-day limit. However, the lack of such a specification may generate conflicting interpretations and disputes among parties.

Beyond academic examples, these situations frequently occur in contractual practice and often lead to legal disputes. When faced with such cases, courts attempt to resolve ambiguities by applying interpretation doctrines or rules, which can vary based on the legal system and the specifics of the case. We will now examine two legal disputes to illustrate the factors that can cause inconsistencies and how courts address them.

Case Law 1. (*Timber Sale*¹) In a purchase agreement for timber cut and removed from a specific area, two clauses regulate the price. The first one sets *a price of \$1.25 per thousand feet of wood* (1), while the second states that “*the entire sale and purchase price of said timber is \$1,400.00*” (2). After the buyer paid the \$1,400 as per clause (2), the seller took the matter to court, seeking the annulment of such clause for inconsistency and claiming a higher amount in accordance to clause (1).

The first instance court indeed found the two clauses to be clearly inconsistent. Following the doctrine according to which, in cases of conflicting clauses, the first clause prevails, the court disapplied clause (2). However, when the matter was taken on appeal, the court’s approach changed radically. Instead of disregarding one of the conflicting clauses, the court adhered to the doctrine according to which no part of the contract should be disregarded if it can be reasonably interpreted to be consistent with the overall intent. Therefore, the court sought and found a consistent interpretation: the price per unit specified in clause (1) was intended to apply only up to the maximum total price of \$1,400 outlined in (2).

Case Law 2. (*Mortgage*²) A mortgage contract features one document that is specifically agreed with the Borrower and one document of standard mortgage conditions applying to all the clients of the lender. The specifically agreed clauses state that (1) *the interest rate would be variable and following the Bank of England Base Rate*. At the same time, one clause of the standard mortgage conditions

¹Hardin v. Dimension Lumber Co., 13 P.2d 602,602 (Or. 1932)

²This is a simplification of Alexander v West Bromwich Mortgage Company [2016] EWCA Civ 496.

allows (2) *the lender to modify the interest rate for one of a number of different listed reasons*, beyond just adjustments tied to the Bank of England Base Rate (e.g., changes in market conditions). An inconsistency clause in the standard conditions states that (3) *in the event of any conflict between the offer document and the standard conditions, the terms of the offer document would prevail*. When the Lender increased the interest rate according to clause (2), the borrower claimed the inconsistency of that clause with the specific agreed terms, and sought its removal according to clause (3).

In contrast to Case Law [1](#), here the inconsistency is not immediately evident from the text. At first glance, in fact, one would say that clauses (1) and (2) are not really inconsistent: the former does not explicitly forbid the possibility of rate variation by the bank, and the latter provides for determined circumstances, thus prevailing over a more general clause. However, the court noted that inconsistency is not limited to cases in which clauses present literal contradictions, but also extends to situations where they “*cannot ‘fairly’ or ‘sensibly’ be read together*”. According to the court, the specific agreed terms defined clearly the main object of the agreement and a general provision such as (2) would not simply specify those terms but rather transform or negate them. The clauses were therefore found inconsistent and the priority clause in (3) was applied.

Through the examples presented, we aim to highlight two key points. First, that the causes of incompatibilities can vary: some are obvious in the contract text, while others are hidden and require careful interpretation. Second, that even when incompatibilities are clear, resolving them is not straightforward and courts may interpret differently the same situation. The combined effects of these two factors produce uncertainty in contractual relationships and, potentially, costly and time-consuming litigation. When contracts are expressed in a programming language, their formal structure can assist in identifying incompatibilities during the drafting process. Therefore, developing a methodology to detect and address these issues is highly valuable for ensuring clarity and preventing disputes.

6.2 Incompatibilities in *Stipula* and their analyses

This section showcases how it is possible to detect inconsistent terms in *Stipula* contracts. We will take into consideration the real world scenarios presented in Section 6.1 and transpose them in *Stipula*. Through code analysis, we will be able to identify the symptoms of inconsistency in *Stipula* contract as specific code patterns. Additionally, we propose possible verification techniques that can be employed to automate such analyses in a toolchain.

We distinguish three categories of inconsistency code pattern: *unreachability*, *anomalies*, *privacy*.

Unreachability. Consider the scenario described in Example 1, concerning a warranty for the sale of a bakery. In this case, the inconsistency affects the temporal structure of a contract: there is a clear mismatch between the general deadline (clause 4) and the sum of all the specific deadlines of the contract described by the warranty (clauses 1, 2 and 3). The warranty of Example 1 can be implemented in *Stipula* as follows:

```

1 stipula Bakery_Warranty{
2   fields price, comp
3
4   agreement ( Seller , Buyer ) {
5     Seller , Buyer : price, comp
6   } \tostate @Start
7
8   @Start Buyer : buy()[w] (w == price) {
9     w ← Seller
10    now + 14 >> @Closing {} ⇒ @End
11    now + 42 >> @Repair {} ⇒ @End
12    now + 42 >> @Comp {} ⇒ @End
13  } ⇒ @Closing
14
15  @Closing Buyer: assert_breach() [] {
16    "Non_conformity" → Seller
17    now + 28 >> @Repair {} ⇒ @Comp
18    now + 42 >> @Comp {} ⇒ @Fail
19  } ⇒ @Repair
20
21  @Repair Seller: repair() [] {
22    "Breach solved" → Buyer
23  } ⇒ @End
24
25  @Comp Seller: compensate()[w] (w == comp) {
26    w ← Buyer
27  } ⇒ @End

```

Listing 6.1: Bakery warranty

The relevant deadlines are defined by the events at lines 11, 12, 17 18. Let us analyse the flow of execution assuming that the contract starts at a time 0.

The invocation of `buy()` at time $t_{buy} = 0$ makes the buyer complete the payment for the bakery and the contract transit to the state `@Closing`. The function's body triggers the three events at lines 10, 11 and 12 that correspond to the deadlines of the clauses (1) and (4) of Example 1. In particular, the event at line 10, noted `ev10`, represents the 14-day period for asserting a breach of the warranty (that will occur at time $t_{buy} + 14$), and the events at line 11 and 12, noted `ev11` and `ev12` represent the 42-days general deadline for advancing any claim under the warranty (that will occur at time $t_{buy} + 42$). If either of these deadlines expires, the contract transits to the `@End` state, indicating termination without further issues.

If a breach occurs, the buyer invokes the function `assert_breach()` at time $t_{buy} < t_{breach}$. This action triggers additional deadlines: in particular, `ev17`, which requires the seller to solve the breach within 28 days, therefore within $t_{breach} + 28$, or, according to `ev18`, to compensate within 42 days from t_{breach} and thus expiring at $t_{breach} + 42$. As we said in Example 1, the breach+compensation procedure may take longer than the 42 days stated in the clause (4) of the contract. In *Stipula* this is highlighted by the fact that `ev18` is unreachable (the code can never be executed) because it is preempted by `ev12`. In fact, it turns out that $t_{buy} < t_{breach}$ implies that the time when `ev12` is executed the ($t_{buy} + 42$) is less than the time of `ev18` ($t_{breach} + 42$).

The work in [112] develops an algorithm for highlighting *unreachable code* in *Stipula*. The problem is difficult because of the interplay between states, time and nondeterminism in the language. In particular, the algorithm integrates the standard control flow analysis that is performed by compilers [2] with a constraint solver that verifies the consistency of time constraints of events. The running prototype has been integrated in the *Stipula* toolchain [48]. For example, in the above case the analyzer returns

`"unreachable_code": [ Comp.ev_18.Fail ]`. Removing these clauses,

which are vacuous according to the operational semantics of the program, is essential, on the one hand to facilitate the formation of the agreement (where the parties have opposing views on the unreachable clauses) and, on the other hand to ensure that the contract fits the intention of the parties (where they wrongly

assumed that the unreachable clauses would be executed).

Non-determinism. Consider now Case Law [1](#). Here, two terms regulating the price of timber seem to contradict each other. Clause (1) sets a price per measure (\$1.25 per thousand feet) and Clause (2) sets a fixed total amount (\$1400). In *Stipula*, prices are stored into **fields** and the compliance with them is verified when the purchase functions are invoked. In particular, fields are used to define constraints that need to be satisfied for the function to be called. When multiple constraints for the same operation are defined, the runtime system nondeterministically chooses one of the instances whose constraints are true. By transposing the contract of Case Law [1](#), in *Stipula* we obtain the following code:

```

1 stipula Timber_Purchase {
2   assets timber
3   fields price_u, price_t
4
5   agreement ( Seller , Buyer ) {
6     Seller, Buyer : price_u, price_t
7   } ⇒ @Start
8
9   @Start Seller: transfer_timber()[t] {
10    t ⇝ timber
11  } ⇒ @Run
12
13  @Run Buyer: buy()[u] (u == timber×price_u) {
14    u ⇝ Seller
15    timber ⇝ Buyer
16  } ⇒ @End
17
18  @Run Buyer: buy()[u] (u == price_t) {
19    u ⇝ Seller
20    timber ⇝ Buyer
21  } ⇒ @End

```

Listing 6.2: Timber purchase

The agreement clause sets **price_u** and **price_t** to 1.45 and 1,400, respectively. According to this code, after the Seller has made the timber available by invoking **transfer_timber** (line 9), the Buyer can perform the actual purchase through two different functions **buy**. These functions in lines 13 and 18 have the same prototype (same name, same initial state and same invoking party) therefore the runtime system decides to execute one of them according to the value of the corresponding boolean guards. The problem with the code of Listing [6.2](#) is that the two guards “**u == timber×price_u**” at line 13 and “**u == price_t**” at line 18 may be both satisfiable. In these cases, the *Stipula* runtime *non-deterministically* chooses to make the contract transit either to **@Run** or to **@End**.

The state machine programming style of *Stipula* allows one to easily identify clauses (*i.e.* functions and events) that have the same prototypes. The additional analysis that corresponding guards are not satisfiable can be performed by an off-the-shelf constraint solver such as [7]. In this case, the solver is more powerful than the one used in the reachability analysis because constraints have more expressive patterns. The prototype implementation of the analyzer is already in our agenda; we expect that, in the foregoing code, the analyzer will output

```
"nodeterminism_warning": [ @Run Buyer:buy : line 13 AND line 18 ].
```

We observe that the message is a “warning”. Indeed, in general, it is possible that two satisfiable constraints are always unsatisfiable at runtime because the flow of execution reduces value ranges of values of fields. Therefore a closer analysis by the designer of the contract is required.

As explained in case Law [1], this incompatibility was resolved by the court interpreting the two clauses as expressing a general rule (*i.e.*, price per measure) and an exception (total fixed price in case the price per measure exceeds \$1400). This can be represented in *Stipula* by defining appropriate function constraints. In particular, function in line 13 would feature the guard (`u = timber×price_u && timber×price_u ≤ price_t`), and function in line 18 the guard (`u = price_t && timber×price_u > price_t`).

Security. In the mortgage contract of Case Law [2], the regulation of the interest rate is ambiguous because one clause of the contract provides that this power is exclusive prerogative of one party (the Bank of England adjusting the Base Rate) and another clause breaks the “exclusivity” by granting access to another party.

In *Stipula*, terms such as interest rates are defined by *fields* that are initialized by the agreement clause and the power to modify them are implemented by clauses that update the field. The relevant parts of the contract of Case Law [2] can be written in *Stipula* as follows (for code simplicity, we overlook the constraint that every rate must be paid within a given deadline; this is specified in *Stipula* by means of events):

```
1 stipula Mortgage {
2   fields: rate, amount, num_r, ret_r
3
4   agreement(Bank, Lender, Borrower){
```

```

5      Bank: rate
6      Lender, Borrower: amount, num_r
7  } ⇒ @Start
8
9  @Start Lender: advance()[u] (u == amount){
10     u ↦ Borrower
11     0 → ret_r
12 } ⇒ @Run
13
14 @Run Bank: update_rate(x) []{
15     x → rate
16 } ⇒ @Run
17
18 @Run Borrower: repay()[u]
19 (u=amount/num_r+amount×rate && ret_r < num_r){
20     u ↦ Lender
21     ret_r+1 → ret_r
22 } ⇒ @Run
23
24 @Run Borrower: repay()[u]
25 (u=amount/years+amount×rate && ret_r=num_r){
26     u ↦ Lender
27 } ⇒ @End
28
29 @Run Lender: update_rate(x) []{           //General T&C
30     x → rate
31 } ⇒ @Run                                   }

```

Listing 6.3: Mortgage

We list, among the parties of the agreement, not only the parties to the contract but also any agent (such as the bank in this case) who contributes to determine the content of the agreement³.

The functions **advance** and **repay** (lines 9, 18, and 23) handle standard mortgage activities, such as advancing the principal amount and repaying it. Additionally, the program includes clauses to adjust the interest rate according to the Central Bank’s Base Rate: the function in line 14 allows the central bank to update the **rate** field (according to the specifically agreed terms), the function in line 27 allows the Lender to modify the interest rate under general mortgage conditions. It is exactly the presence of these two functions that caused problems in Case Law [2](#); in particular, the problematic clause is the function at line 27 that the court found to conflict with the true intentions of the parties. Said otherwise, it is not admissible that the field **rate** can be updated by a party that is not the Central Bank. In Computer Science, this problem is known as *secure information*

³The party **Bank** in this context does not represent a human agent actively participating in the contract. Instead, it should be understood as a software agent responsible for retrieving relevant information from institutional sources, such as an oracle. We use the name **Bank** here for simplicity and clarity.

flow whereby only certain parties are only allowed to access to given fields. There are several types of accesses, such as read-write (**rw**), read-only (**r**), no-access (**na**). In the foregoing case, the Bank has read-write access to **rate**, while Borrower and Lender have read-only access. The *Stipula* language may be extended in order to support security information declarations in the agreement clause as follows

```

agreement(Bank, Borrower){
  Bank: rate
  Lender, Borrower: amount, num_r
  rate: Bank [rw]; Lender, Borrower [r]
} ⇒ @Start

```

Then we can use a standard technique that has been proposed for the analysis of the secure information flow: *type checking* [171]. This technique amounts to define a number rules – one for every syntactic term – that enforce the notion of *security correctness*. For example, in *Stipula*, we have the rule

$$\frac{\Gamma + \bar{u} : \mathbf{rw} + \bar{v} : \mathbf{rw} \vdash E \quad \Gamma + \bar{u} : \mathbf{rw} + \bar{v} : \mathbf{rw} \vdash S}{\Gamma \vdash @Q \text{ A } f(\bar{u})[\bar{v}](E)\{S\} \Rightarrow @Q'}$$

stating that, if E and S are correct with respect to the security constraints specified in $\Gamma + \bar{u} : \mathbf{rw} + \bar{v} : \mathbf{rw}$ (u and v are the formal parameters of f ; the body of f can access to them in **rw** mode), then the function $@Q \text{ A } f(\bar{u})[\bar{v}](E)\{S\} \Rightarrow @Q'$ is correct with respect to the constraints in Γ . The map Γ is defined by the (refined) agreement clause: according to line 4 of the above code, we have $\Gamma = [\mathbf{rate} \mapsto (\mathbf{Bank} : \mathbf{rw}), \mathbf{rate} \mapsto (\mathbf{Lender} : \mathbf{r}), \mathbf{rate} \mapsto (\mathbf{Borrower} : \mathbf{r})]$. It turns out that, for the code in Listing 6.3 (with the refined agreement) $\Gamma + \bar{u} : \mathbf{rw} + \bar{v} : \mathbf{rw} \vdash S$ cannot be verified for the function at line 27; therefore the function is not secure.

The prototype implementation of the security analyzer is also in our agenda; when fed with Listing 6.3 we expect that the analyzer will output

```

"security_alert": [ @Run Lender:update.rate at line 27 is not secure ].

```

6.3 Key Findings

The analysis conducted in this section leads to several key observations. First, contractual incompatibilities are not rare and can take different forms: some are

evident, such as conflicting deadlines or conditions that can never be met, while others are subtler and stem from overlapping rights or powers among the parties. In all cases, they introduce uncertainty, increase negotiation costs, and may ultimately result in litigation.

Second, the patterns that these incompatibilities produce in computable contract code are often similar and can be grouped into a few categories. The object-oriented syntax and the state-machine semantics of *Stipula* proved suitable for the application of program analysis techniques, which identified three main code patterns associated with incompatibility — *unreachability*, *non-determinism*, and *security* — and allowed us to design dedicated analyses for each. In particular, the unreachability analysis has already been implemented and integrated into the *Stipula* workbench, while the non-determinism and security analyses have been designed and are currently under development.

Overall, these findings suggest that integrating program analysis techniques into the drafting process of computable contracts can significantly improve their reliability.

Chapter 7

Statutory Integration

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 is shared and recognised across many legal systems, albeit implemented in different forms.¹ Civil law jurisdictions generally refer to this doctrine 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.

While computable contracts enhance predictability in contractual relations and may contribute to the reduction of transaction costs, they also raise concerns because of their potential to operate in violation of applicable legal norms. Moreover, the absence of standardized or regulated computational infrastructures poses significant challenges for institutional oversight, rendering it difficult for authorities to intervene in or suspend the execution of such contracts, especially when these are deployed within distributed ledger environments [160, 120]. It is therefore im-

¹This 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.

perative to develop methodologies that ensure compliance with applicable statutory provisions from the outset of contract formation [93, 118]. 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.

In order to capture this specific legal pattern, we developed an ad hoc feature for *Stipula*, termed `import`. This construct enables the incorporation of statutory provisions into specific contractual instances.

In the next section, we provide a general overview of contract integration mechanisms across different legal systems and introduce the use case of unilateral contract termination. Then, section 7.3 demonstrates the practical use of the `import` feature in *Stipula* through a running example, and Section 7.4 formalizes the semantics of the construct.

7.1 The Role of Law in Contracts

The role of statutory law in shaping the content of contracts is recognised 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 [146]. A similar provision is found in Article 1194 of the French Code Civil.

In common law jurisdictions, the same principle is recognised through the notion of *implied terms*. In English law, courts have long accepted the possibility of implying terms within contracts² 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 contracts³ [41]. A similar recognition, although with differences in the specific scope of implication in fact and in law, can be found in

²See *Liverpool City Council v Irwin* [1977] AC 239; *Scully v Southern Health and Social Services Board* [1992] 1 AC 294.

³For example, the Sale of Goods Act 1979 provides standard rules applying to commercial contracts.

the U.S. legal system,⁴ where the concept of implied terms is also reflected in the Restatement (Second) of Contracts⁵.

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 [158]. 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* [41, 40, 22].

Default integration occurs for gap-filling purposes and to preserve contracts' validity in the context of incomplete contracts [40]. 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 [91]. 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 performed⁶. However, the article's final paragraph states that this applies without prejudice to any contrary agreement⁷. This indicates the default nature of such provision and that any specific agreement between parties

⁴See *Henningsen v. Bloomfield Motors, Inc.*, 161 A.2d 69 (N.J. 1960).

⁵Sections 204–205.

⁶Art. 1373 par. 2 *Codice Civile*.

⁷Art. 1373 par. 4 *Codice Civile*.

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 freedom of contract and establish boundaries to protect consumers, by defining prohibited or necessary contents [145, 177]. The boundaries defined serve as mandatory pro-consumer arrangements [14] that cannot be derogated by parties unless in a way that is more favourable 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 time⁸ 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.

To conclude, mandatory and default statutory provisions have different, specific effects on contracts. While legal doctrine and courts provide interpretive criteria to resolve potential inconsistencies among the former and the latter when contracts are expressed in computational terms, these interactions must be explicitly modelled through mechanisms that capture priorities and exceptions. The next section reviews how such issues have been addressed in the field of AI and Law through formal approaches.

7.2 Formal Approaches to Conflicting Legal Sources

The modelling and analysis 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, through logic-based approaches. Various formal

⁸The 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.

approaches have been proposed, including non-monotonic and defeasible reasoning [152], belief revision [19], and argumentation frameworks [153].

In particular, in formal argumentation [57, 140, 139], 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 the purposes of this section, approaches that combine multiple legal sources within a single reasoning process are particularly relevant. One prominent example is the work of Sartor and Dung [58], 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 sub-modules 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 [141] 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 way.

Although these afore mentioned contributions provide general mechanisms to model conflicts and priorities among norms, these have not been implemented within the contract field. In particular, no existing language for computable contracts yet provides a native mechanisms for contract 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. To fill this gap we developed such a feature for *Stipula* and it is presented below.

	<pre> stipula C [import_clause] { assets h fields x̄ agreement(Ā){ Ā₁ : x̄₁ ... Ā_n : x̄_n // ∪_{i∈1..n} Ā_i ⊆ Ā, ∪_{i∈1..n} x̄_i ⊆ x̄, ∩_{i∈1..n} x̄_i = ∅ } ⇒ @Q F } </pre>
	<pre> norm N <P>(Z)[r]/Q/ [constraints φ] [baseline b] { F } import_clause ::= [mandatory] import N <P>(Z)[r]/Q'/ </pre>
<i>Functions</i>	$F ::= - \mid @Q A : f(\bar{y})[\bar{k}](E)\{S W\} \Rightarrow @Q' F$
<i>Prefixes</i>	$P ::= E \rightarrow x \mid E \rightarrow A \mid E \multimap h, h' \mid E \multimap h, A$
<i>Statements</i>	$S ::= - \mid P S \mid \text{if}(E)\{S\} \text{else}\{S\} S$
<i>Events</i>	$W ::= - \mid E \gg @Q\{S\} \Rightarrow @Q' W$
<i>Expressions</i>	$E ::= v \mid X \mid \text{now} \mid E \text{ op } E \mid \text{uop } E$
<i>Values</i>	$v ::= n \mid \text{false} \mid \text{true} \mid s \mid \mathbb{t}$
<i>Constraints</i>	$\varphi ::= x \text{ rop } w \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$
<i>Baselines</i>	$b ::= x = w \mid x = w, b$
<i>Data</i>	$w ::= v \mid \text{now} \mid \text{now} + k \quad (k \text{ is a natural number})$

 Table 7.1: Syntax of *Stipula* with import features (in red)

7.3 The import mechanisms in *Stipula*

The extension of *Stipula*'s syntax for importing statutory provisions is written in red in Table 7.1. 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 $\mathbf{x}$ 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 $\mathbf{x} = w$. We assume that pairwise different elements in $\mathbf{b}$ address different fields. Finally, data w are either constant values or terms like **now** or **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 $\mathbf{b}$. Formally, the baseline values $\mathbf{b}$ is a *function* from fields’ names to value; hence $\mathbf{b}(\mathbf{x})$ returns the value of $\mathbf{x}$ in $\mathbf{b}$ (in Section 7.4 these functions are called *memories*). If the parties do not specify a value for a field $\mathbf{x}$ whose baseline value is defined in $\mathbf{b}$, then the field is initialized to $\mathbf{b}(\mathbf{x})$. Furthermore, if the import is *mandatory*, and the value of $\mathbf{x}$ agreed upon by the parties does not satisfy the constraints in φ , the agreement is considered invalid if $\mathbf{x} \notin \text{dom}(\mathbf{b})$, and, as before, the field is initialized to $\mathbf{b}(\mathbf{x})$ (and we always assume that $\mathbf{b}(\mathbf{x})$ satisfies φ). No constraint checking is performed in the case of a *default import*.

Regarding functions, let $F_{\mathbf{N}}$ 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 $F_{\mathbf{N}}$. If no function from $F_{\mathbf{N}}$ 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 $F_{\mathbf{N}}$ 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. Consider the contract in Table 7.2

This represents a simple contract for the supply of a VPN digital service. 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

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.*
-

Table 7.2: Digital Service Contract

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.

```

1 stipula VPN {
2   fields: tstart, tend, price, product
3   asset: wallet, good
4
5   agreement (Trader, Consumer)(){
6     Trader, Consumer : tend, price, product
7     // price 10 Euro, tend = 12M, 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 }
```

Listing 7.1: *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.

Default Import 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 7.2. 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 7.2: 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 7.4). 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

$0.5 \times \text{_wallet} \multimap \text{_wallet}, \text{_trader}$

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 [7.1](#) is performed by modifying the contract declaration as follows:

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

As a consequence, 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 [7.2](#) 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
2   <Consumer, Trader>(tstart,tend)[wallet]/Run,End/ {
3
4   fields: tstart, tend, price, twith, product
5   assets: wallet, good
6
7   agreement (Consumer, Trader)(){
```

```

8      Consumer, Trader: tend, price, product
9      // tend= 12M, price= 10Euro, product= VPN
10     } => @Init
11
12     @Init Trader: supply(type)[v] (type == product){
13         v -o good
14     } => @Pay
15
16     @Pay Consumer: pay()[w] (w == price) {
17         now -> tstart
18         w -o wallet
19         tstart + tend >> @Run { wallet -o Trader } => @End
20     } => @Run
21
22     @Run Consumer: withdraw() [] (now <= tstart+7D) {
23         0.1 x wallet -o wallet, Trader
24         wallet -o Consumer
25     } => @End
26
27     @Run Consumer: withdraw() [] (now > tstart+7D) {
28     } => @Run
29 }

```

Listing 7.3: *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 7.2 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 7.1.

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 [14] that cannot be derogated from, except in ways that are more favourable to the Consumer⁹. 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  $\multimap$  _consumer
5      }  $\Rightarrow$  @_Qb
6  }
```

Listing 7.4: Norm `m_52cons`: Article 52 of Italian Consumer Code

In contrast to the norm in Listing 7.2, 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  $\multimap$  _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 7.4.

Assume now that the VPN service agreement includes a different clause regulating unilateral termination, Clause 4(b).

In contrast to Clause 4(a), this provision is only partially inconsistent with the consumer’s right of withdrawal expressed by Article 52. While the right must be exercisable free of charge within the initial 14-day period, according to 4(b) the

⁹Article 25 of Directive 2011/83 and, at the national level, Article 59 of the Italian Consumer Code.

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

withdrawal has a cost of 2 euro in the first thirty days and of 4 euro afterwards. 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):

```
1 stipula VPN3 mandatory import m_52cons
2   <Trader, Consumer>(tstart)[wallet]/ Run,End / {
3
4   fields: tstart, tend, price, product
5   assets: wallet, good
6
7   agreement (Trader, Consumer)(){
8     Trader, Consumer: tend, price, product
9     // tend= 12M, price= 10Euro, product= VPN
10  } => @Init
11
12  ...
13
14  @Run Consumer: withdraw() [] (now <= tstart+30D){
15    2 -> wallet, Trader
16    wallet -> Consumer
17  } => @End
18
19  @Run Consumer: withdraw() [] (now > tstart+30D){
20  } => @Run
21 }
```

Listing 7.5: *Stipula* VPN service Agreement with termination clause 4(b)

This version closely mirrors the contract in Listing 7.3; 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 7.4, provided that the constraint $\text{now} \leq \text{tstart} + 14\text{D}$ is satisfied. After this period, the invocation triggers the `withdraw` function defined in Listing 7.5, 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.

7.4 Semantics of the import extension

We now illustrate the semantic extensions introduced to accommodate the integration of statutory norms. Compared to the semantics presented in Chapter 4, the *Stipula* **import** construct introduces two new auxiliary operations. These operations are required to apply the *baseline* and *constraint* values defined in the imported modules.

In particular, the *mandatory import* operation $\varphi \succ_{\mathbf{M}} \ell, \flat$ returns an updated memory that:

- adopts the baseline values defined in the module if the contract memory is empty;
- keeps the values agreed by the parties whenever these satisfy the module’s constraints;
- replaces any non-compliant value with the corresponding baseline, if such baseline is defined;
- is *undefined*—and therefore causes the agreement to fail—if some constraint is violated and no baseline value is available for the affected field.

The *default import* $\varphi \succ_{\mathbf{D}} \ell, \flat$ returns a memory ℓ that:

- adopts the baseline values defined in the module if the contract memory is empty;
- otherwise preserves the values agreed by the parties, without checking any constraint.

In short, the mandatory import enforces compliance with the module’s constraints, whereas the default import merely fills any missing values without performing constraint checks.

Moreover, the *selection operation* defined in Section 4.2 is extended to account for the priority between the imported module and the main contract. First of all, the declaration part $\mathbb{D}$ of a contract can take two forms:

$$\mathbb{D} = F \quad \text{or} \quad \mathbb{D} = F \blacktriangleright F'$$

where F and F' denote, respectively, the sets of functions belonging to the contract and to the imported module. The operator $\blacktriangleright$ defines a priority: functions in F

take precedence over those in F' (or vice versa, depending on the import type).

Formally, given a contract configuration with imported clauses

$$\mathbb{D} \vdash_{\mathbb{t}} \mathbb{Q}, \ell, \Sigma, \Psi$$

where $\mathbb{D} = F \blacktriangleright F'$, the selection operation is defined as follows:

$$\begin{aligned} \mathbb{D}[\textcircled{\mathbb{Q}} \mathbb{A} : \mathbf{f}]_{\ell, \bar{u}, \bar{v}}^{\mathbb{t}} &= F[\textcircled{\mathbb{Q}} \mathbb{A} : \mathbf{f}]_{\ell, \bar{u}, \bar{v}}^{\mathbb{t}} \quad \text{if } F[\textcircled{\mathbb{Q}} \mathbb{A} : \mathbf{f}]_{\ell, \bar{u}, \bar{v}}^{\mathbb{t}} \neq \emptyset, \\ &= F'[\textcircled{\mathbb{Q}} \mathbb{A} : \mathbf{f}]_{\ell, \bar{u}, \bar{v}}^{\mathbb{t}} \quad \text{otherwise.} \end{aligned}$$

In other words, when a function with the given name and caller is found in the first component (for instance, in the contract or in the imported module, depending on the priority order) and its preconditions are satisfied, that version is executed. If no matching function is found there, the system proceeds to check the second component.

Depending on the import type, the transition relation Rule [AGREEMENT] defines a different declaration at the initialisation of the contract.

- For *mandatory import*, the declaration is $F_N \blacktriangleright F$, meaning that the *norm's* functions override the contract's ones;
- in a *default import*, it is $F \blacktriangleright F_N$, meaning that the *contract's* functions have priority.

This mechanism allows *Stipula* to represent, at execution level, the legal distinction between *mandatory* and *default* statutory rules: the former override the contract's provisions, while the latter apply only in their absence.

7.4.1 Conflicts Among Imports

The `import` extension presented so far, admits imports of a single norm only. 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. However, the formalism introduced in Section [7.4](#) is also adequate for modelling sequences of imports defined by the syntax

$$\text{stipula } C \overline{\text{import_clause}}; \{ \dots \}$$

where $\overline{\text{import_clause}}$ 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

$$\text{stipula } C \text{ import_clause}_1; \dots; \text{import_clause}_n; \{ \dots \}$$

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

$$\text{stipula } C \overline{\text{import_clause}}; \{ \dots \}, \ell \Vdash \mathbb{D}, \ell'$$

that takes a term $\text{stipula } C \overline{\text{import_clause}}; \{ \dots \}$ and a memory ℓ and returns a sequence of function declarations $\mathbb{D}$ and a memory ℓ' . For example, if F are the functions in the contract C and F_1 and F_2 are the functions in the norms N_1 and N_2 , respectively, then

- $\text{stipula } C \text{ mandatory import } N_1 \dots; \text{mandatory import } N_2 \dots; \{ \dots \}, \ell \Vdash F_2 \blacktriangleright F_1 \blacktriangleright F, \ell'$, which highlights the precedence order F_2 then F_1 and finally F ;
- $\text{stipula } C \text{ mandatory import } N_1 \dots; \text{import } N_2 \dots; \{ \dots \}, \ell \Vdash F_1 \blacktriangleright F \blacktriangleright F_2, \ell'$, which specifies that the functions with lower precedence are those in F_2 .

(with suitable ℓ and ℓ').

7.4.2 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 presented in the current section. 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 [58, 141]. The integration of a similar feature, however, falls outside the scope of this work and is left as a direction for future research.

7.5 Key Findings

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.

The use case scenario presented shows the practical utility of this feature for heavily regulated contractual domains such as consumer contract law.

Conclusions

This chapter presented the design and progressive development of *Stipula*, a domain-specific language for the formalisation and execution of computable contracts. The work aimed to demonstrate how computational abstractions can faithfully capture the structure and dynamics of legal agreements, combining formal precision with legal analysis.

The first part of the chapter provided the foundations of the language, describing its main building blocks, syntax, and semantics, together with the legal abstractions they are intended to represent. *Stipula* builds on an imperative and object-oriented programming style, mirroring the paradigms currently used in practical implementations of computable contracts, such as Solidity, Java, or Go. At the same time, its semantics is grounded in the finite-state automaton (FSA) model, which has been shown to effectively represent dynamic objects like contracts [67]. This dual grounding ensures both expressiveness and operational clarity: contracts can be written in an intuitive yet formally analyzable way.

On top of this foundational layer, the chapter introduced three extensions developed specifically to capture key patterns of contract law. Each extension addresses a distinct legal or practical issue that arises in real-world contracting. The first concerns *amendment clauses*: through the higher-order extension *higher-order Stipula*, contracts can be modified at runtime under controlled conditions. This feature reconciles the immutability of digital execution with the flexibility required by legal practice, allowing parties to introduce or adapt contractual provisions without departing from the original protocol.

The second extension introduces mechanisms for *contract analysis*. Through the examination of real legal cases, it was shown that incompatible clauses—common sources of disputes—can be systematically mapped onto specific code patterns in

Stipula. This insight enabled the application of program-analysis techniques to detect incompatibilities during drafting, such as unreachable code, non-deterministic behaviours, or violations of access restrictions. These analyses increase both the technical robustness and the legal reliability of computable contracts.

The third extension concerns *contract integration*, addressing how statutory norms interact with private agreements. By distinguishing between mandatory and default provisions and discussing formal methods for handling conflicts among legal sources, *Stipula* provides a framework for integrating external legal norms into the executable logic of a contract. This development aligns the language with the complexity of real legal systems, in which private contracts operate within layered normative hierarchies that must be respected during execution.

In a nutshell, *Stipula* follows a block-by-block methodology for the development of a domain-specific language: each legal concept is analysed individually, and suitable techniques from the imperative and object-oriented paradigms are then adapted or extended to capture that concept effectively within a computational framework. This modular and incremental approach has allowed *Stipula* to evolve from a basic contract formalisation tool into a comprehensive framework that combines formal rigour, legal awareness, and operational flexibility.

Overall, the language now provides not only a means to express and execute contractual obligations, but also to analyse, verify, and dynamically adapt them. *Stipula* thus represents a step forward in bridging the gap between legal reasoning and computational execution, paving the way for more reliable and legally grounded smart contract systems.

Part III

Survey

Chapter 8

Setting the stage for Comparison

The present chapter develops a comparative framework for evaluating different programming languages in their ability to represent *computable contracts*. Having introduced a new domain-specific language for contracts, *Stipula*, we now compare it with other approaches to legal modelling in order to position it within the current landscape of programming languages for the law, particularly with respect to declarative and imperative paradigms. At the same time, this comparison aims to identify the specific benefits and limitations inherent in each paradigm, as well as distinctive features emerging from individual languages.

To ensure a consistent basis for evaluation, a common use-case scenario is employed. The case study presented in the following section consists of a simplified *Data Use Agreement* drafted in accordance with the EU *Data Act*. The choice of this example is particularly significant because, unlike other comparative studies [78], it grounds the analysis in a contractual model explicitly envisaged by the European legislator. This approach not only anchors the discussion in a legally relevant and contemporary context, but also illustrates how computable contracts can concretely operationalise statutory provisions concerning data sharing and access.

Section 8.1 describes the common criteria developed for evaluating the chosen programming languages; Section 8.2 presents the Data Use Agreement used as common case study, and provides an initial decompositions of its main legal elements establishing a common objective ground for comparison.

8.1 Evaluation Criteria

Routen and Bench-Capon [147] define three key requirements as the essential characteristics of an ideal formal representation: (1) *faithful representation of the law*; (2) *ease of validation and maintenance*; and (3) *computational adequacy*. (1) concerns the ability of the formalisation to represent the legal source as closely as possible, either through structural isomorphism with the legislation or through a correct interpretation of its meaning. (2) refers to the possibility of checking the correctness of a formalisation and updating it efficiently when the legal source changes. (3) indicates that the formalisation should be executable in practice, i.e., it should support the reasoning and operations required without incurring prohibitive computational costs.

These requirements were originally formulated as guiding principles for experts when formalising legal texts. For the purposes of the present work, the focus of the evaluation slightly shifts from the final formalisation to the *languages* that employed for it. In other words, rather than assessing whether a given formalisation meets these requirements, we evaluate the extent to which the programming languages under consideration provide the means to satisfy them.

According to this perspective shift, we define three categories of evaluation for the purposes of Chapter 8 inspired to Routen’s requirements: *legal fidelity*, *accessibility*, *operational expressiveness*. These are addressed separately below.

8.1.1 Legal Fidelity

Legal fidelity concerns the extent to which a language enables a faithful representation of the contractual terms, embracing both structural and semantic aspects.

From a structural standpoint, legal fidelity refers to the ability of a language to preserve the visual organisation of the original legal text in the formalisation. This characteristic, known as *isomorphism* [15], ensures that the formal representation has a one-to-one correspondence with the legal source. For the evaluation of *structural fidelity* (isomorphism), each language is assigned with a qualitative level of isomorphism:

H High isomorphism: the contractual structure and concepts are preserved with

clarity, and individual articles or clauses can be directly mapped into formal rules.

- M* Medium isomorphism: the overall structure remains visible but simplified; some clauses may be merged or restructured, dependencies between provisions are only partially maintained, and legal concepts are recognizable but not always preserved in their original form.
- L* Low isomorphism: the representation loses its direct connection to the contractual text; articles and clauses are no longer traceable, and legal concepts are replaced by computational constructs that obscure the link with the original source.

From a semantic perspective, legal fidelity depends on a system’s capacity to represent the elements of contract law and legal reasoning. Three properties are taken as central indicators [74]:

- *Defeasibility.* Defeasible reasoning requires a system, under certain conditions, to refrain from adopting conclusions despite endorsing the premises [151]. This is an intrinsic feature of normative systems, as legal rules always have exceptions and a balancing between competing principles is always required (see Section 3.1). In this work, defeasibility is assessed not only as an abstract logical feature but as the ability to model contractual exceptions and conflict-resolution patterns in a faithful and executable way. Concretely, the focus is on whether the language allows one to:
 1. represent exceptions and derogations in a way that is transparent at the level of the contract (e.g. “X must pay, unless...”);
 2. encode and apply conflict-handling mechanisms (e.g. rule priorities, specificity, or explicit override patterns) that determine which provision prevails;
- *Normative positions:* contractual provisions define obligations, prohibitions, permissions (i.e., deontic positions), and rights—deontic positions with distinct legal consequences. This indicator focuses on a language’s ability to

1. represent these modalities as well-defined modelling patterns; and
 2. reason about compliance status, including contrary-to-duty structures and exceptions, within the formalisation.
- *Temporal expressiveness.* Time is a crucial dimension of contracts: obligations may carry deadlines, rights can be exercised only within certain periods, and remedies often depend on the expiration of terms. This indicator concerns generally the ability of a language to handle time-bound contractual clauses in a faithful and executable way. Specifically, temporal expressiveness evaluates the extent to which the language allows one to:
 1. represent temporal features of legal clauses (e.g. obligations deadline, duration of a contract). Different languages support different levels of time granularity, which may range from precise dates and timestamps (continuous time) to discrete or event-driven notions of time (where the passage of time is tracked through state changes or events);
 2. keep track of the passage of time to automate legally relevant consequences. This may rely on an *internal* notion of time (e.g. system clocks or built-in mechanisms) or on *external* inputs (e.g. oracles, libraries, or user-provided facts).

The evaluation of *semantic fidelity* will be conducted against these three properties, using the following scale, which reflects the level of specificity and support provided by the language:

- ✗ The support provided by the language to capture the relevant property is lacking or very limited;
- ✓ The property can be captured by the language faithfully;
- ✓✓ The language captures the property through ad hoc features.

8.1.2 Accessibility

Formalising contract clauses requires legal expertise to analyse the legal text, identify its elements, and map them into the constructs of a formal language. As contracts may evolve, such representations must be legally validated to ensure ongoing

correspondence with the source text. Validation concerns the faithfulness of the code to the legal text [37].

Legal validation generally demands collaboration between lawyers and programmers or, ideally, individuals with interdisciplinary skills. However, this increases the time and resources needed to develop and maintain computable contracts. These challenges can be mitigated by designing languages that enable legal experts to understand and use them directly.

Accessibility therefore measures the extent to which a language allows non-technical users, particularly legal experts, to comprehend and work with it without specialised programming knowledge. In this evaluation, accessibility is assessed primarily through *readability*, i.e., the ease with which a legal expert can understand what a code fragment represents and what it does¹

We focus on the readability of languages' syntax, analyzing it both quantitatively and qualitatively. In particular, we evaluate the level of:

- *Syntactic Commitment (SC)*: the amount of predefined syntax elements (e.g., type declaration, keywords) compared to the amount customizable syntax. Low SC = high naming/modeling freedom; High SC = many prescribed constructs; and
- *Domain Alignment (DA)*: the extent to which predefined constructs natively reflect contractual concepts (e.g., obligations, time, states, events, roles, terms) rather than general-purpose or system-level notions.

Following these indicators, the overall accessibility will be calculated according to the levels shown in Table 8.1, and evaluated on a three-tier scale: *Tier 3*, indicating systems not designed for use without IT expertise, requiring general-purpose programming skills; *Tier 2* for systems that still require technical training, but are comparatively more intuitive for legal experts thanks to their high level of isomorphism or domain-specific design; *Tier 1*, for systems designed to be directly accessible to non-IT experts, e.g. languages with highly customizable syntax (such

¹The intelligibility of a language's syntax does not imply that the same language is equally easy to write. However, we believe that the evaluation of a language's *usability* is too closely dependent on the individual user and would therefore require an empirical analysis, which falls outside the scope of the present work.

as CNLs). Where predefined syntax is provided is closely related to contractual elements.

Tier	SC level	DA level
Tier 3	high	low
Tier 2	medium	medium
Tier 1	low	high

Table 8.1: Accessibility Tiers

8.1.3 Operational Expressiveness

The third and final evaluation category concerns the extent to which a language can represent and program the *operational aspects* of legal agreements. These aspects concern the performance of the contract and therefore the concrete actions to be carried out by the parties, or those that we wish to automate through the contract [38]. In other words, operational expressiveness captures how well a language supports the specification and automation of behaviours that give effect to contractual provisions.

For the purpose of evaluation, operational expressiveness will be assessed along two specific indicators:

- *Resource handling*: the ability to represent and manage the resources regulated by the contract (e.g., goods, currency).
- *Automatability*: the capacity of the language to trigger computational effects at the occurrence of parties' actions or when specific conditions are fulfilled. In some systems, execution depends on external queries or manual input, whereas others include event-driven or time-driven mechanisms that allow actions to fire autonomously.

Both characteristics will be assessed using the three-level scale reported in Table 8.2. The symbol $\odot$ indicates that the feature is not supported; $\bullet$ signals that it is only partially supported or requires workarounds; and $\bullet$ denotes full support, meaning that the feature is provided natively and enables the language to represent or execute the relevant aspect comprehensively.

Symbol	Meaning
⊙	No support
◐	Limited support
●	Full support

Table 8.2: Operational Expressiveness Scale

8.2 The Case Study: A Data Use Agreement under the EU Data Act

The use-case scenario for this chapter is represented by a *Data Use Agreement* (DUA), a contractual instrument expressly envisaged by the Data Act to govern the use and exploitation of data generated by interconnected products. As discussed in Section 2.2, one of the main policy goals of the Data Act is to stimulate the contractualisation of data exploitation, thereby fostering the creation of a flourishing data market in which those who provide access to their data can rightfully obtain compensation [89].

The Act covers both personal and non-personal data but, for the purpose of this case study, only non-personal data will be considered. As for the *products* generating such data, the Act defines them as “an item that obtains, generates or collects data concerning its use or environment and that is able to communicate product data via an electronic communications service, physical connection or on-device access [...]”².

For the purposes of the case study, two actors addressed by the regulation are central: the *Data Holder* and the *User*. The Data Holder is the party legally entitled, or obliged, to use and make available the data generated by a connected product or related service³. This role is often, though not always, played by the product’s manufacturer, who typically retains control over the data flows produced by the device. The User, by contrast, is the person or organisation that operates the connected product and, through its use, generates the relevant data⁴. In a typical business context, this might be the customer or licensee to whom the

²Article 2(5) EU Reg. 2023/2854

³Article 2(13) EU Reg. 2023/2854

⁴Article 2(12) EU Reg. 2023/2854

product is supplied under a contractual arrangement.

In regulating the relationship between these two subjects, on one hand the Data Act establishes a non-derogable right for the user, the *right of access*. Manufacturers of connected products must design them so that users can directly and easily access the data they generate⁵. Where such access cannot be provided by design, the data holder is instead required to make the data available upon the user's request⁶. On the other, it provides that any use of non-personal data by the data holder must rely on a contractual arrangement with the user⁷. In practice, whenever the operation of a product or service entails the generation of data accessible to the data holder, a contractual instrument (either a clause within the main contract or a separate agreement) must define the permitted and prohibited uses of such data⁸. These agreements may also include compensatory mechanisms, for instance when the user allows the data holder to reuse or share the data for specific purposes⁹.

The *Data Use Agreement* thus becomes the legal vehicle for allocating rights, obligations, and possible remuneration related to the exploitation of data generated by connected products.

8.2.1 Computable Contracts for Data Sharing

The Data Act explicitly identifies computable contracts as a possible technology for implementing data use and data sharing agreements¹⁰. This is particularly significant, since the scenario features many of the elements required by this technology to provide the intended benefits of self-enforcement as discussed in Section 2.1.4.

First, the agreement concerns a natively digital asset, data, allowing, in principle, direct control of the resource through a digital means. Contractual logic could even be integrated within the product's software, recalling Szabo's idea of "contracts embedded in the world" (Section 2.1). Moreover, several obligations, such as the user's right of access, are formulated in clear and objective terms, requiring

⁵Article 3 EU Reg. 2023/2854

⁶Article 4 EU Reg. 2023/2854

⁷Article 4(13) EU Reg. 2023/2854

⁸See Recital 25

⁹Recital 25

¹⁰Recital 104; Art. 36

minimal interpretation and thus lending themselves to automation. Finally, the inclusion of compensation mechanisms introduces a transactional dimension that can be efficiently managed through standard computable contract features such as escrow or automated payment functions.

Taken together, these features make the Data Use Agreement an especially fitting test case for the exploration of computable contracts.

8.2.2 A Data Use Agreement in the Car Rental sector

This section presents the running example of data use agreement that will be used in for the comparison in Chapter 8.

The scenario is taken directly from the official “Frequently Asked Question on the Data Act” document published by the EU commission on February 2025 [43]. The structure and type of provisions, instead, are drafted on the basis of the model contractual terms (MCT) published in the report of the Expert Group on B2B data sharing and cloud computing contracts [62], set up by the EU commission for the definition of non-binding standardized causes for the data sharing agreements provided for by the data act¹¹

The contract reproduced in Table 8.3 exemplifies a Data Use Agreement in a B2B context between two main parties. *CarMaker Inc.* is the manufacturer of interconnected vehicles and, in this scenario, acts as the *Data Holder* under the Data Act. *CarRental Inc.*, a rental company purchasing vehicles from CarMaker for its fleet, qualifies as the *User* since it owns and operates the connected products. The object of the agreement is the *Car-Product*—a connected vehicle equipped with an embedded telemetry system that collects and transmits data about its usage and operating environment.

The contract defines the conditions governing access to and use of the data generated by the vehicles. In line with the Data Act, Article 3 establishes the Data Holder’s obligation to grant access to the User, while Articles 4 and 5 specify the permitted purposes for data use. For the Data Holder, these purposes form a closed list of authorised uses, whereas the User may generally use the data freely,

¹¹Art. 41 of the Data Act provides that “The Commission, before 12 September 2025, shall develop and recommend non-binding model contractual terms on data access and use, including terms on reasonable compensation [...]”

8.2. THE CASE STUDY: A DATA USE AGREEMENT UNDER THE EU DATA ACT

Data Use Agreement	
Art.1 Parties	The present Agreement is stipulated between: - CarRental Inc. ("User") - CarMaker Inc. ("Data Holder") for the mangement of the data generated by the car vehicles purchased by User from Data Holder (the 'Product').
Art.2 Scope	This Agreement governs the access to and use of data generated by the Product, including all relevant metadata necessary for interpretation and usage ("Data"). Data includes any operational, diagnostic, and usage data generated by the Product
Art.3 Data Access Right	Upon request by the User, the Data Holder shall make the Data and necessary metadata accessible to the User or a third party acting on their behalf within 5 days.
Art.4 Data Holder Permitted Uses	The Data Holder may use the Data solely for the following purposes: a) fulfil contractual obligations towards the User under this Agreement; b) monitoring, maintaining, and improving the functioning, safety, and security of the Product or related services; c) create aggregated or derived data for commercial purposes.
Art.5 User Permitted Uses	The User may use the Data made available by the Data Holder upon their request for any lawful purpose and/or share the Data freely subject to the limitations below The User shall not share the Data with any third party that qualifies as a Gatekeeper under Article 3 of Regulation (EU) 2022/1925 (Digital Markets Act), unless explicitly authorized in writing by the Data Holder.
Art.6 Compensation for Commercial Use	In relation to the use under Article 4(c), the Parties agree that the Data Holder shall pay the User a compensation equal to [0.10] EUR per [MB] of Data generated and used. The total compensation due shall be calculated at the end of each calendar month and paid by the Data Holder to the User within thirty (30) days.
Art.7 Breach and Remedies	Failure to comply with any of the obligations set out in the agreement shall constitute a Breach. In the event of Breach, the non-breaching Party may: a) request the termination of the contract; or b) claim compensation for direct pecuniary damages incurred.
Art.8 Termination	This Agreement shall terminate automatically upon the expiration or termination of the contract governing the use of the Product.

Table 8.3: Rented Cars Data Use Agreement

8.2. THE CASE STUDY: A DATA USE AGREEMENT UNDER THE EU DATA ACT

subject only to a restriction on sharing with entities designated as *gatekeepers*¹². The list of purposes and the exceptions draw upon the *Model Contract Clauses* developed by the EU Expert Group on B2B Data Sharing and Cloud Computing Contracts¹³.

Article 6 introduces a compensation mechanism linked to the commercial exploitation of data as defined in Article 4(c). In particular, the User authorises the Data Holder to reuse the data for commercial purposes beyond the strict fulfilment of contractual obligations, in exchange for fair remuneration, as envisaged by Recital 25 of the Data Act.

Article 7 defines contractual breach and specifies the remedies available to the non-breaching party, while Article 8 outlines the general causes of termination.

8.2.3 From Contract Clauses to Normative Components

In order to formalise the Data Use Agreement within different computational paradigms, it is first necessary to identify the fundamental components of the knowledge we are trying to represent. Such a decomposition not only provides the conceptual bridge between natural language drafting and formal modelling, but also establishes a common basis for comparing how different programming languages capture and operationalise the same normative content.

The provisions of the Data Use Agreement can be decomposed into a set of fundamental normative building blocks. These include both *constitutive rules* (CR), which define key legal concepts, and *regulative rules* (RR), which specify obligations, permissions, and prohibitions. Table 8.4 represent the core elements relevant for formalisation.

It is important to stress, however, that any formalisation of a legal text inevitably represents an interpretation. Different modellers may represent the same contract in different ways, depending on a slightly different reading of the provisions, on the chosen focus, or on the methodological preferences guiding their work. The formal examples presented in the following chapter should therefore

¹²Gatekeepers are large digital platforms meeting the criteria set out in Article 3 of Regulation (EU) 2022/1925, the Digital Markets Act.

¹³[62](#)

8.2. THE CASE STUDY: A DATA USE AGREEMENT UNDER THE EU DATA ACT

not be regarded as the only — or necessarily the best — possible versions, but rather as consistent applications of each language to a shared case study.

ID	Clause	Role	Content
<i>CR1</i>	Art.1 - Agreement	Constitutive Rule	Identifies parties, product and general object of the contract.
<i>CR2</i>	Art 2 - Scope	Constitutive Rule	Identifies the types of data covered.
<i>RR1</i>	Art.3 - Data Access Right	Obligation	Subject: Data Holder. Action: Make data available to the User (or a third party). Deadline: 5 days from the request.
<i>RR2</i>	Art.4 - Data Holder Permitted Uses	Permission	Subject: Data Holder. Action: Use data. Condition: listed purposes hold.
<i>RR3</i>	Art.5 - User Permitted Uses	Permission	Subject: User. Action: Sharing with third parties.
<i>RR4</i>	Art.5 - Prohibition to Share Data	Prohibition	Subject: User. Action: sharing with gatekeepers. Exception: Authorisation.
<i>RR5</i>	Art.6 - Compensation for Commercial Use	Obligation	Subject: Data Holder. Action: Pay User. Amount: €0.10 per MB of data. Deadline: within 30 days.
<i>CR3</i>	Art.7 - Breach	Constitutive Rule	Breach as the failure to comply with obligations or prohibitions.
<i>RR6</i>	Art.7 - Remedies (Termination)	Permission	Subject: Non-breaching party. Action: Terminate the contract. Condition: Breach occurs (CR3).
<i>RR7</i>	Art.7 - Remedies (Claim Damages)	Permission	Subject: Non-breaching party. Action: Claim compensation for direct pecuniary damages. Condition: Breach occurs.
<i>CR4</i>	Art.8 - Termination	Constitutive Rule	Defines when the contract is considered terminated

Table 8.4: Data Use Agreement - normative components

Chapter 9

Comparison

Building upon the framework outlined in the previous section, this chapter presents a comparative analysis of selected programming languages representing two distinct paradigms.

On the one hand, it focuses on *logic-based declarative* languages, which have long been employed to model legal knowledge in a computable form, particularly within the AI and Law field (Section 3.1). On the other hand, *object-oriented imperative* (henceforth, OO-imperative) languages constitute the prevailing paradigm in real-world implementations of computable contracts. Examples include Java, used in the ISDA Common Domain Model for financial products¹, Solidity on the Ethereum platform, and Java or Go for Hyperledger-based distributed ledger systems².

The objective of this analysis is to critically examine how different computational paradigms can represent legal agreements in ways that are both legally faithful and computationally effective, highlighting the differences that arise not only between paradigms but also among the languages within each of them, and assessing the specific tasks for which each is best suited.

¹<https://cdm.finos.org/docs/cdm-overview>

²<https://hyperledger-fabric.readthedocs.io/en/release-2.5/>

9.1 Scope of the Evaluation

The comparative analysis presented in this chapter will be conducted on a selected set of programming languages that have been deemed representative of the two paradigms under consideration, logical and imperative. The selection aims to balance between widely adopted languages and more experimental proposals, so as to provide a representative overview of the design space and the trade-offs involved.

Four languages are considered for the logic-based declarative paradigm:

- *Prolog* [42]: a general-purpose logic programming language widely used in AI and Law for modelling legal knowledge.
- *Logical English* [101]: a Controlled Natural Language built on Prolog, designed to enhance readability and accessibility for non-technical users.
- *Defeasible Deontic Logic on ASP* [76]: a specialised logic integrating deontic modalities and defeasibility to capture normative reasoning.
- *Logic Production System* [107]: an experimental hybrid language combining declarative rules with imperative constructs for reactive systems.

As to the OO-imperative paradigm:

- *Solidity* [25]: the leading language for blockchain-based smart contracts and the de facto standard on the Ethereum platform.
- *Stipula* [47]: is newly developed domain-specific language designed specifically for legal agreements, presented in Chapter 4.

The selection of languages is admittedly unbalanced towards logical approaches, as the imperative paradigm is already extensively examined through the development and analysis of *Stipula* in Part II. Given that the thesis as a whole focuses on this domain-specific imperative language, the comparative analysis in this chapter concentrates instead on logical frameworks, which provide the most established foundations for the formal representation of legal knowledge and normative reasoning.

9.2 Logic-Based Declarative Languages

9.2.1 Prolog and First-Order logic

Prolog (PROgramming in LOGic) is a logic programming language used to make statements about a world, that can be interpreted and evaluated to answer questions [175]. The language grew out of experiments with Q-systems for automated translation of meteorological reports in Canada [42]. Soon after, Robert Kowalski and colleagues connected these ideas with SLD-resolution (Selective Linear Definite clause resolution) [100], an inference mechanism for theorem proving. Prolog's design was guided by the idea of using First-Order Predicate Logic not only as a formal specification tool but also as an executable language, where computation corresponds to logical deduction.

A Prolog program (also referred to as a *knowledge base*) is composed of three main elements: facts, rules, and queries. *Facts* represent atomic statements about the world, expressed as predicates with possible arguments. For example, the statement “Alice is the parent of Bob” can be formalised as:

$$\textit{parent}(\textit{alice}, \textit{bob}).$$

Prolog also supports the use of *variables*, denoted by capitalized identifiers (e.g., X, Y, Z). Variables act as placeholders for arbitrary terms and are instantiated with concrete values during query resolution.

Rules encode implications between predicates and are written as Horn clauses of the form:

$$H \leftarrow B_1, \dots, B_n$$

where H is the *head* of the rule, and $B_1, \dots, B_n$ are the predicates conjoined in the *body*. In Prolog, the head (consequent) is inferred whenever all the predicates in the body (antecedent) are satisfied. For instance:

```
mother(alice, bob) :- parent(alice, bob), female(alice).
```

Finally, *queries* are used to evaluate whether certain statements hold in the knowledge base. Consider the program:

```
mother(alice, bob) :- parent(alice, bob), female(alice).  
parent(alice, bob).  
female(alice).
```

The query:

```
?- mother(alice, bob).
```

returns `true`, since the facts `parent(alice, bob)` and `female(alice)` allow the rule `mother/2` to be satisfied.

When presented with a query, the Prolog interpreter attempts to prove it by recursively searching through the knowledge base. This process relies on two key mechanisms: *unification*, which instantiates variables by matching terms, and *backtracking*, which systematically explores alternative rules or facts whenever a branch of reasoning fails. In this way, the Prolog engine can explore all possible derivations until either a solution is found or no further possibilities remain [39].

Prolog became particularly relevant in the field of AI and Law, since the seminal work by Sergot, Kowalski and colleagues on the British Nationality Act [156], which represented one of the first systematic attempts to encode a statute as a logic program. Following this, a rich line of research explored the use of logic programming for formalising legislation and automating legal reasoning [16, 46, 26].

Over time, researchers also developed extensions of Prolog to better address features specific to legal reasoning. For example, PROLEG [154] introduced explicit mechanisms for representing exceptions; Epilog [71] and dynamic logic programming extended the paradigm to handle temporal and evolving rules; and DR-Prolog [8] provided support for defeasible deontic logic, crucial for representing permissions, obligations, and prohibitions in a non-monotonic setting.

The Contract

In Prolog, the contract is represented as a set of clauses that correspond to the constitutive and regulative rules identified in Section 8.2.3, both structured as Horn Clauses. For each rule, the corresponding facts are provided to ensure the correct instantiation of the agreement code with the specific contractual terms. Listing 9.1 reports the complete Prolog implementation. For the subsequent languages, only

selected code excerpts will be shown, as many structures and concepts are repetitive; this avoids unnecessary redundancy and keeps the discussion concise. All the full code implementations are reported in Appendix [A](#).

```

1 :- set_prolog_flag(unknown, warning).
2 :- use_module(library(date_time)).
3 :- dynamic data_used_mb/2.
4
5 % Art.1 - Agreement
6 data_agreement(Holder, User, Product, Data):-
7     party(Holder, holder),
8     party(User, user),
9     product(Product).
10    covered_data(Data).
11
12 %%% Facts
13 party(carMaker, holder).
14 party(carRental, user).
15 product(car).
16
17
18 % Art.2 - Scope
19 covered_data(Data):-
20     operational_data(Oper),
21     diagnostic_data(Diagn),
22     usage_data(Usq),
23     metadata(Meta),
24     Data = [Oper, Diagn, Usq, Meta].
25
26 %%% Facts
27 operational_data(operational).
28 diagnostic_data(diagnostic).
29 usage_data(usage).
30 metadata(metadata).
31
32
33 % Art.3 - Data Access Right
34 shall(make_data_available(Holder, User), Within):-
35     data_agreement(Holder, User, _, _),
36     request_data(User, T1),
37     date_add(T1, 5 days, Within).
38
39
40 % Art. 4 - Data Holder Permitted Uses
41 may(holder_use_data(Holder, User, Purpose)):-
42     data_agreement(Holder, User, _, _),
43     permitted_purpose(Purpose).
44
45 permitted_purpose(contractual_obligations).
46 permitted_purpose(monitring).
47 permitted_purpose(development).
48 permitted_purpose(commercial_use).
49
50 % Art. 5 "User Permitted Uses"
51 may(user_use_data(User, Holder, Purpose)):-
52     data_agreement(Holder, User, _, _),
53     \+ unlawful_purpose(Purpose, use).
54
55 may(share_data(User, Holder, Third_party)):-
56     data_agreement(Holder, User, _, _),
57     third_party(Third_party),
58     \+ gatekeeper(Third_party).
59
60 may(share_data(User, Holder, Third_party)):-
61     data_agreement(Holder, User, _, _),

```

```

62     gatekeeper(Third_party),
63     authorised_to_share(Holder, User, Third_party)).
64
65
66 % Art. 6 "Compensation for Commercial Use"
67 shall(compensate(Holder, User, Amount), Within):-
68     data_agreement(Holder, User, _, _),
69     data_used_mb(MB,T0),
70     Amount is 0.10 * MB,
71     date_add(T0, 5 days, Within).
72
73 generated_data_mb(MB):-
74     date_get(today, T),
75     retractall(data_used_mb(_, _)),
76     assertz(data_used_mb(MB,T)).
77
78
79 % Art. 7 "Breach and remedies"
80 breach(Breacher, shall(0, Within), NonBreacher):-
81     shall(0, Within),
82     date_get(today, T),
83     date_compare(T, >, Within),
84     \+ action(0),
85     0 =.. [_, Breacher, NonBreacher | _].
86
87 breach(Breacher, may(Permission), NonBreacher):-
88     action(Permission),
89     \+ may(Permission),
90     Permission =.. [_, Breacher, NonBreacher | _].
91
92 %%% Remedies
93 remedy(Claimant, Breacher, claim_damages):-
94     breach(Breacher, _, Claimant).
95
96 remedy(Claimant, Breacher, request_termination):-
97     breach(Breacher, _, Claimant).
98
99
100 % Art. 8 "Termination"
101 termination(data_agreement(Holder, User, Data)):-
102     remedy(_, _, request_termination)
103     ; terminates_product_contract.

```

Listing 9.1: Prolog Formalisation of DUA

CR1 The constitutive rule for Article 1 is expressed as a Horn clause that collects the essential elements of the agreement: the parties, the product, and the data. In the code, this is captured by the predicate `data_agreement/4` (line 6) This clause functions as a “header rule” for the entire contract, establishing its validity whenever its basic conditions are satisfied. All other contractual provisions rely on this predicate, ensuring that their applicability presupposes the existence of a valid agreement.

CR2 The types of data covered by the contract defined in Article 2 are grouped together using a list structure (line 24), which allows the aggregation of different categories into a single term. The predicate `covered_data/1` collects operational,

diagnostic, usage data, and metadata into a list. Lists in Prolog are ordered collections written in square brackets (e.g., [a, b, c]). Here, the list represents the set of data types to which the contractual provisions apply. As with Article 1, this clause is also constitutive, since the definition of the data object is a necessary precondition for the contract’s validity [39].

RR1 Article 3 introduces the obligation for the Data Holder to make data available to the User in relation with the exercise of the data access right. Deontic positions are represented using meta-predicates as this would prove particularly useful for the definition of contract breaches. In particular, obligations are identified with the `shall` predicate which features two arguments. The first describes the action to which the deontic position refers to, while the second specifies the temporal constraint (`Within`). Here, the relevant action is (`make_data_available`) and the conditions for the obligation to arise form the rule body: the validity of the agreement and a data request by the User made at a specific time $T1$. The deadline (`Within`) is calculated adding five days to the time of the request, as provided by Article 3.

RR2 Article 4 is modelled as a prescriptive rule expressed by the meta-predicate `may/1`, denoting a permission. The predicate takes as argument the action of using the data for a certain purpose. Again, the agreement rule works as a general precondition. The set of admissible purposes is captured through the predicate `permitted_purpose/1`, which enumerates the authorised categories (lines 45-48) (e.g., contractual obligations, monitoring, development, commercial use). In this way, the permission to use the data is conditional on the purpose being one of those explicitly listed in the contract.

RR3– RR4 Article 5 is formalised through multiple Prolog rules, each corresponding to the different provisions it contains. The first captures the User’s general permission to make use of the data, similarly to *RR2*. However, while Article 4 specifies a closed list of authorised purposes, Article 5 refers to “any lawful purpose”, which is inherently open-ended. Therefore, rather than enumerating all lawful uses, the formalisation interprets this clause as a general permission with

an exception: the action is permitted unless the purpose is unlawful. In Prolog, such exceptions can be represented through *negation as failure* [39]. This means that if the interpreter cannot match the predicate `unlawful_purpose/1` with any fact in the knowledge base, the condition is considered satisfied³.

A second rule specifies the permission to *share* the data, which Article 5 regulates explicitly among the permitted uses. Here again, negation as failure is employed to capture the exception: the permission does not hold if the third party is a gatekeeper (line 58).

Finally, a further rule defines the exception to this exception, namely the case where the Data Holder has explicitly authorised sharing with a gatekeeper (lines 60-63). Taken together, these rules reproduce the layered logic of Article 5. For example, the query:

```
?- may(share_data(carRental, carMaker, Third_party)).
```

will succeed when the third party is not a gatekeeper, fail if the third party is a gatekeeper without authorisation, and succeed again when explicit authorisation is present.

RR5 The obligation to pay compensation under Article 6 is represented through the meta-predicate `shall/2`, which associates the prescribed action (`compensate/3`) with a corresponding deadline (line 67). Two elements must be specified in order to define this obligation: the amount due and the deadline for payment. Both are derived from the predicate `data_used_mb/2` (line 69), which records the amount of data used (in megabytes) within a certain date (T0). Based on this reference point, the program computes the amount by multiplying the volume of data by the contractual rate (0.10 EUR per MB) and determines the deadline by calculating the thirty days-deadline using the predicates from the `date_time` library. The fact `data_used_mb/2` itself is produced through the rule `generated_data_mb/1` (line 73), which simulates the logging of data production by the connected product and the monthly calculation required by Art 6. When called, this rule updates the knowledge base by removing any previous record of

³Negation as failure operates under the “closed world” assumption, i.e., all relevant facts about the domain are assumed to be included in the knowledge base.

data used and adds a new one with the amount of data (MB) and the current timestamp ($T0$). This mechanism illustrates a potential imperative use of the program: beyond representing static contractual rules, Prolog can also dynamically update the knowledge base, automatically recording events that trigger obligations.

CR3– RR6– RR7 Whenever a party performs an action, the system records it in the knowledge base as `action(A)`; this record becomes the basis for reasoning about whether obligations or permissions have been respected.

For obligations, a breach occurs when a prescribed action is not performed within the deadline. The corresponding rule (line 81) checks the presence of a `shall/2` predicate, verifies that the current date has passed the deadline, and confirms that no matching action has been recorded. For permissions, a breach arises when an action is performed that is not actually allowed by the contract. In this case, the rule (line 87) refers to the meta-predicate `may/1`: if an action is recorded but no permission can be established, the system classifies it as a violation. This design shows the advantage of using meta-predicates: by uniformly encoding obligations and permissions, breaches can be defined in a general way, covering all types of contractual provisions with a single, reusable structure.

Finally, remedies are formalised as prescriptive rules (permissions). Two distinct permissions are granted to the non-breaching party: the right to claim damages and the right to request termination of the contract. Both are automatically triggered when a breach is detected, thereby linking violations of deontic positions to the contractual consequences specified in Article 7.

CR4 Termination is defined by the predicate `termination/1`, which is triggered in two cases: either when a party exercises the remedy of termination following a breach, or when the underlying product contract comes to an end. The latter condition is left as an atomic predicate (`terminates_product_contract/0`) for simplicity, while the former directly links Article 8 to the remedies established in Article 7.

Evaluation

Isomorphism	Defeasibility		Normative positions	Temporal express.	
	Exc	Confl		Rep	Track
H	✓	✓	✓	✓	✗

Table 9.1: Legal fidelity in Prolog (updated criteria).

Legal Fidelity From a structural perspective, the formalisation in Prolog exhibits a high level of isomorphism. Each article of the contract can be mapped to a corresponding rule or small set of rules, preserving the organisation of the original text. For instance, Article 5 requires multiple rules because of its broader normative scope, but there are no provisions that are scattered across unrelated parts of the program. This modularity ensures that modifying a contractual clause only requires updating the corresponding rule, without affecting the overall structure of the formalisation.

From a semantic perspective, Prolog offers considerable expressive power, as it is Turing complete. Although it does not provide domain-specific constructs for legal concepts, the language is sufficiently flexible to encode the key features of contractual reasoning. The following table summarises the evaluation across the three dimensions considered:

In the contract example, defeasibility comes into play in the representation of Articles 4 and 5, which regulate permitted uses of data together with their exceptions. These provisions were modelled by combining *negation as failure* with conditional rules, allowing the system to capture exceptions without producing an explicit conflict. For instance, the coexistence of permissions and prohibitions to share data (depending on whether the recipient is a gatekeeper or whether authorisation is given) can be represented in Prolog so that, given the facts at hand, only one rule results true.

This approach is often sufficient, but in other contexts it may be necessary to handle conflicts of norms in a more structured manner. Negation as failure provides a way to encode exceptions that resembles defeasible reasoning, yet its effectiveness depends on the assumption that the set of exceptions is fixed and

known. Since Prolog remains based on monotonic logic, the system cannot retract conclusions when new information is added, unless the rules themselves are revised. A possible workaround is to anticipate future exceptions by leaving an exception in every rule through NAF, for example:

```
may(p) :- a, b, \+ exception(may(p), _, _).
```

This allows the solution of potential future rule conflicts without the need to rewrite existing rules, but simply adding specific exceptions to the KB of the form `exception/3, .` However, this solution is verbose and does not provide a direct ordering of conflicting rules. For contractual modelling, where exceptions are typically finite and explicitly stated, these limitations are less problematic. In contrast, for open-ended or evolving legal domains, dedicated non-monotonic extensions of Prolog—such as Defeasible Prolog (d-Prolog) [130]—offer more adequate support.

As to normative positions, these can be represented smoothly through meta-predicates such as `shall/2` and `may/1`. This provides a common framework that enables general reasoning patterns, for instance to detect breaches. However, since these positions are not natively embedded in Prolog’s semantics, the handling of violations must be explicitly defined through additional rules.

Time is not a primitive feature of Prolog, but it can be represented by manipulating dates and timestamps as arguments of predicates. In the implementation, temporal constraints were encoded with the help of external libraries such as `library(date.time)` (see Articles 3 and 6), which allow operations like adding days or months (`date_add/3`) and comparing deadlines (`date_compare/3`). This makes it possible to represent contractual deadlines and compute due dates. However, while the representation of temporal features is supported, the tracking of time is not: Prolog does not have an internal clock or event-driven mechanism, so temporal effects only materialise when queries are made or facts are explicitly asserted. This means that consequences depending on the passage of time cannot be automatically triggered, but require an external process to check the conditions and update the knowledge base.

Accessibility From the standpoint of *syntactic commitment* (SC), the language offers a high degree of customizability: users are free to define their own predicates

and structures without being bound to a fixed set of constructs. The Horn clause structure together with the use of parenthesis and punctuation (commas and periods) represent the only prefixed pattern. We therefore evaluate Prolog's SC as medium-low.

As to *domain alignment* (DA), Prolog does not natively provide constructs corresponding to contractual concepts such as obligations, rights, or temporal clauses. These must instead be introduced manually through meta-predicates (e.g. `shall/2`, `may/1`) and auxiliary rules. For these reason we consider DA as low.

In conclusion, according to Table 8.1 Prolog's accessibility can be categorized as a Tier 2, with the very high level of customizability able to balance the absence of contractual-specific constructs.

Operational Expressiveness Operational expressiveness in Prolog is constrained by the fact that every element in a program must be represented as either a fact or a Horn clause.

In general, Effects such as “if a deadline expires then trigger a breach” cannot be triggered automatically but require the execution of specific queries (e.g., `?-breach(X)`) to fire the computation. Nevertheless Prolog provide a mechanism to dynamically update the KB through the `assertz/1` and `retract/1` predicates. In the contract formalisation, this technique was used to record the generation of data and to log the fulfilment of obligations.

As a further example, the performance of the obligation to compensate could be represented as follows:

```
compensate(Holder, User, Amount) :-
    shall(compensate(Holder, User, Amount), Within),
    date_get(today, T),
    assertz(action(compensate(Holder, User, Amount), T)),
    assertz(user_balance(Amount)).
```

Here, the system checks whether the obligation is in force, and if so, records its performance in the knowledge base by asserting new facts.

This feature can be used to simulate imperative behavior, and attach automatic to effects parties actions. However the possible effects stick with adding or deleting

facts from the KB, automatability in Prolog is *limited*.

From the standpoint of *resource handling*, Prolog resources can be modelled and manipulated as anything else simply as predicates and arguments, in a purely symbolic fashion, without guaranteeing, uniqueness, or conservation properties. As a result, while it is possible to simulate transfers or updates (for instance by asserting or retracting facts), the representation remains indirect and lacks dedicated constructs that can guarantee resource-specific properties such as linearity or uniqueness. For this reason, resource handling is evaluated as *limited*.

9.2.2 Logical English

Logical English⁴ (LE) is a Controlled Natural Language (CNL) designed for programming and knowledge representation [101, 103].

Technically, LE is syntactic sugar for Prolog: its statements are mapped to Prolog syntax and then interpreted. This mapping occurs through the declaration of *templates*, english sentences representing predicates where asterisks indicate the positions of the arguments (e.g, variables). Table 9.2 shows a simple example, comparing an ordinary English sentence with its formulation in Logical English and its corresponding Prolog clause.

Ordinary English	Alice likes anyone who likes logic.
LE Template	*a person* likes *a thing*
LE Syntax	Alice likes a person if the person likes logic.
Prolog	likes(alice, A) :- likes(A, logic).

Table 9.2: Example sentence in Logical English.

Logical English does not rely on knowledge of natural language semantics, except for a limited set of logical connectives (“if”, “and”, “or”, “it is not the case that”) and determiners (“a”, “an”, “the”), as well as the interrogative “which” and the relative pronoun “that” for building meta predicates. Variables are expressed as noun phrases that begin with a determiner and end with a common noun (e.g. “a person”, “the thing”). The indefinite article (“a” or “an”) introduces a variable on first occurrence, while the definite article (“the”) refers back to the same variable in subsequent occurrences.

⁴<https://github.com/LogicalContracts/LogicalEnglish>

Apart from these syntactic considerations, LE retains the essential features of Logic Programming. A program consists of facts and rules of the form “conclusion if conditions” (i.e., Horn clause). All variables are implicitly universally quantified within the scope of the rule in which they occur, meaning that variables with the same name in different rules remain independent. Finally, queries are expressed through the interrogative determiner “which”, which identifies the unknown variable to be retrieved. For instance, the Prolog query `?- likes(alice, A)` corresponds in LE to:

```
?- Alice likes which person.
```

LE has been proposed for several applications in the legal domain, such as financial contracts [102], tax law [45] or traffic rules for automated vehicles [149].

The Contract

The Logical English formalisation largely mirrors the Prolog implementation in terms of rule structure, but with some additional syntactic constructs that are necessary to allow the handling of natural language-like sentences. The full LE formalisation is reported in Appendix A.1. Here we will mention the most relevant elements to our discussion.

LE requires the declaration of *templates* for all predicates that will appear in the knowledge base so that the parser is able to translate LE sentences into Prolog predicates. For example:

```
1 the templates are:
2 *a person* concludes an agreement with *an other person* for *a dataset*.
3 *a person* has the obligation that *an eventuality*.
```

CR1 – CR2 The structure of the rules then mirrors almost identically the corresponding Prolog code. Articles 1 and 2 are defined as constitutive rules with their respective conditions, where the rule for Article 1 is reused as a general precondition throughout the code. Like in Prolog, it is also possible to define lists, which is useful to describe the set of data covered by Article 2 (line 58).

RR1– RR2– RR3– RR4 With respect to the regulative rules of Articles 3, 4, and 5, permissions, prohibitions and obligations can be built using meta predicates, expressed with the connector “that”, similarly to what happens in Prolog. Below, an excerpt of the full code representing Article 3.

```
1 % Art.3 - Data Access Right
2 a holder has the obligation that
3   the holder makes the data available to a user within a D
4   if the holder concludes an agreement with the user for a dataset
5   and the user requests the data on a D1
6     and the D is 5 days after the D1.
```

In this context, a key difference emerges with Prolog: in LE a meta-predicate can only take a single argument after “that”. This restriction prevents the encoding of deadlines as a specific argument of the obligation predicate, separated by the action `shall(Action, Within)`. For this reason, the temporal condition is here embedded in the action predicate itself (e.g. `a person makes the data available ... within a D`, in line 3 of the excerpt). For permissions (artt 4 and 5) this limitation is less problematic, since they normally do not involve deadlines and thus require no additional argument beyond the action itself. This adaptation, however, has consequences for the subsequent definition of breaches.

RR5 This follows the same general structure as in Prolog, but here too the temporal constraint must be integrated into the action predicate, rather than represented as a separate argument of the obligation. Another important difference is that LE does not support Prolog’s dynamic predicates such as `assert` and `retract`. As a result, facts like the amount of data generated or the date of calculation cannot be updated internally but have to be supplied as external input to the knowledge base, as a further predicate.

CR3– RR6– RR7– CR4 The definition of breaches (Art. 7) highlights the limitations on meta-predicates. Because each meta-predicate can only have a single argument, it is impossible to define a general breach rule that systematically handles all obligations and permissions. Instead, a separate breach clause has to be written for each prescriptive rule in the knowledge base. This makes it necessary to explicitly repeat the structure of each action in the breach conditions, for

instance distinguishing between failure to provide access to data, unlawful sharing, or compensation not made within the deadline. Below, the breach of the compensation obligation and of the sharing restriction are defined.

```

1  % Art. 7 "Breach and remedies"
2  %% Breach of Compensation obligation
3  a holder is in breach of the agreement with a user
4      if the holder concludes an agreement with the user for a dataset
5      and the holder has the obligation that
6          the holder compensates the user with an amount eur within a D
7      and it is not the case that
8          the holder compensates the user with the amount eur within the D.
9
10 a holder compensates a user with an amount eur within a D
11     if the holder compensates the user with the amount eur on a D1
12     and the D1 is before the D.
13
14 %% Breach of user sharing with gatekeeper
15 a user is in breach of the agreement with a holder
16     if the holder concludes an agreement with the user for a dataset
17     and the user shares the data given by the holder with a thirdParty
18     and it is not the case that
19         the user has the permission that
20             the user shares the data given by the holder with the thirdParty.

```

Finally, remedies and termination are formalised in the same way as in Prolog. Remedies (a party has the remedy to) are linked to the occurrence of a breach, while termination is expressed through the meta-predicate `it terminates that`, again reproducing the logical structure already discussed in the Prolog version.

Evaluation

Legal Fidelity The evaluation of Logical English (LE) under this criterion broadly mirrors that of Prolog, since the underlying semantics is the same. Table 9.3 therefore reflects the same values.

Isomorphism	Defeasibility		Normative positions	Temporal express.	
	Exc	Confl		Rep	Track
H	✓	✓*	✓*	✓	✗

Table 9.3: Legal fidelity in Logical English

From a structural perspective, the level of isomorphism is comparable to Prolog. On the one hand, the restrictions on meta-predicates in LE require duplication of rules to capture the same normative provision, as illustrated by the modelling

of breaches in Article 7. This makes the code more verbose and somewhat less elegant. On the other hand, the natural-language style of LE makes the correspondence between clauses and legal text more transparent: a careful reader can usually match a contractual provision to its formal representation with little effort. Overall, these tendencies balance out, and the degree of isomorphism remains equivalent to Prolog.

From a semantic standpoint, LE mirrors Prolog’s expressiveness. Negation as failure and conditional rules can be used to represent exceptions, as in Article 5, and deontic modalities are expressed through meta predicates such as “a person has the obligation that”. However, the single argument restriction (after `that`) significantly reduces expressiveness and increase verbosity. For instance, in Prolog one can introduce general patterns such as `exception/3` to handle conflicts without rewriting rules. In LE this strategy is not feasible due to the argument limitation, which makes it harder to capture potentially relevant information in a reusable way. Moreover as shown in the modelling of Article 7, reasoning about normative positions also becomes more cumbersome, requiring the explicit definition of lengthy rules instead of compact meta-level reasoning. These constraints do not prevent the representation of defeasible or normative reasoning altogether, but they limit their practical expressiveness. For this reason, in Table 9.3 these properties are marked with an asterisk (*) to indicate the partial limitation.

Finally, expressing temporal elements in LE follows the same declarative style as in Prolog: deadlines and dates are expressed as arguments but no internal temporal reasoning semantics is present.

Accessibility Accessibility is the main strength of Logical English, as its syntax closely resembles natural English.

The language imposes very few syntactic patterns (SC): users can freely define templates in controlled natural language, with only minimal requirements regarding punctuation and sentence structure. This makes the syntax highly customisable and easy to adapt to the contractual domain. Although the resulting code can at times be verbose, the overall meaning of each clause remains clear to a careful legal reader even without any IT expertise.

As to DA, the language does not offer predefined constructs corresponding to legal concepts. Instead, such notions must be introduced through user-defined templates. This makes a strict evaluation of DA less meaningful, since alignment with the legal domain depends entirely on the modeller's choices.

Taken together, these considerations place LE within Tier 1. Its extremely low syntactic commitment and natural-language readability make it directly accessible to non-technical users, even though domain alignment relies on customisation rather than native features. It is important to note, however, that readability does not imply ease of writing. Drafting valid LE program, for example, requires understanding the mechanism of templates and the correct use of the `the` and `a` keywords.

Operational Expressiveness Again, LE behaves Prolog under this evaluative category. On the one hand, resources can be represented and manipulated only in a symbolic way, lacking *ad hoc* constructs (limited: $\bullet$). As to automatability, all provisions must be expressed as Horn clauses, limiting the possibility to trigger automatic effects, without the launch of a query. There is no internal clock or event-handling mechanism, meaning that deadlines or temporal events must be introduced into the knowledge base from outside.

In addition, LE also lacks built-in dynamic updates such `assertz/1` and `retract/1` in Prolog. As a result, actions' effects over the contract cannot be dynamically triggered but can only be observed as the final result of an inference process. For this reason, we assess automatability in LE as not supported ($\odot$).

9.2.3 Defeasible Deontic Logic on ASP

Defeasible Deontic Logic (DDL) is a non-monotonic formalism specifically designed to capture the normative aspects of reasoning, including obligations, permissions, prohibitions, and their possible exceptions [82]. It originates from the broader family of Defeasible Logic [130], a lightweight yet expressive system for rule-based reasoning that has been extensively employed in Artificial Intelligence and Law to represent legal norms and to model contrary-to-duty reasoning [151, 84]. DDL extends this framework with deontic operators, thus enabling the formalisation of

normative positions in a way that is both computationally efficient and faithful to legal practice.

A DDL theory is composed of three main components [76]: *facts*, representing indisputable pieces of information about a case; *rules*, which encode constitutive, prescriptive, or permissive norms; and *superiority relations* that resolve conflicts by determining which rule prevails when two conclusions are incompatible. In particular, prescriptive rules may contain *compensation chains*, allowing one to model contrary-to-duty obligations: if an initial obligation is violated, a secondary obligation comes into force to compensate, and so forth.

While DDL provides the formal semantics, Answer Set Programming (ASP) offers a powerful computational environment to implement them. ASP is a logic programming paradigm based on stable model semantics, in which a program is evaluated in terms of “stable models” or “answer sets” that satisfy all implications contained in it⁵. Reasoning is supported by efficient solvers such as **Clingo**⁶, which have been successfully applied to large-scale reasoning problems.

The ASP implementation of DDL⁷ realizes the semantics of the logic as a *meta-program*, while individual legal theories are represented as ASP programs [76].

A DDL program begins by declaring the atoms that may be subject to deontic qualifications, using the predicate `atom/1`. For example, `atom(access).` explicitly states that the atom `access` can appear as the content of obligations, permissions, or prohibitions. This declaration fixes the normative universe of discourse: only the atoms introduced in this way can be associated with deontic predicates.

Rules are expressed by associating a label to their head and by specifying the conditions under which they are applicable. Depending on the type of norm, different predicates are used, such as `constitutiveRule`, `prescriptiveRule`, or `permissiveRule`. For example, a prescriptive rule might be declared as

```
prescriptiveRule(r1, obligation(access)).
applicable(r1, obligation(access)) :-
```

⁵ASP programs consist of rules syntactically similar to those in Prolog, but they are evaluated in a “bottom-up” fashion to compute all answer sets, rather than being queried imperatively. In this sense, ASP can be thought of as a purely declarative form of logic programming. Moreover, ASP supports both classical negation and negation-as-failure, allowing the representation of statements that are provably false, or merely not provable as true [87].

⁶<https://potassco.org/>

⁷<https://github.com/gvdgdo/Defeasible-Deontic-Logic/tree/main>

```
defeasible(agreement(holder,user)).
```

Here the label `r1` identifies the rule, while the predicate `applicable/2` links the head of the rule to its preconditions. Prohibitions are modelled as obligations to perform a negative action.

The notion of *defeasibility* is central to the framework. The predicate `defeasible(X)` states that X can be derived defeasibly, meaning that it is either given as a fact or supported by an applicable rule which is not overruled by a stronger one. This mechanism allows rules to be overridden by exceptions and priorities, a feature that is indispensable in the legal domain. Superiority relations are defined explicitly using the predicate `superior(r1,r2)`, which states that rule `r1` prevails over rule `r2` whenever both are applicable.

The evaluation of normative positions with the available set of facts is captured by dedicated predicates for obligations, permissions, and violations. An obligation arises whenever there exists an applicable prescriptive rule whose conclusion has not been defeated. Permissions are derived in two ways: first, through the axiom that “obligation implies permission”, and second, through weak permission, which holds when it is impossible to prove the opposite as obligatory [76]. Violations occur when an obligation is in force and the opposite of its content also holds defeasibly. For instance, if both `obligation(p)` and `defeasible(non(p))` are present, the system infers a violation.

DDL also supports the representation of contrary-to-duty obligations through the predicate `compensate/4`. This allows one to specify that, whenever an obligation is violated, another obligation becomes applicable as a form of compensation, giving rise to chains of reparatory duties. In this way, the framework can faithfully capture the layered structure of many legal norms, where violations trigger new obligations rather than leaving a normative gap.

The Contract

The contract is represented in DDL on ASP through a set of rules that roughly correspond to the constitutive and regulative rules norms identified in the legal analysis (section 8.2.3). Each article is mapped into either a constitutive rule, which defines the fundamental elements of the contract, or a prescriptive or per-

missive rule introducing obligations, prohibitions, or permissions. The rules are labelled systematically: **cr** for constitutive rules, **ob** for obligations, **pro** for prohibitions (i.e. obligations to a negative action), **per** for permissions, and **br** for breach definitions. The full DDL code is available in Appendix [A.2](#).

CR1– CR2 Article 1 and 2 are expressed as constitutive rules. The conditions for applicability of these rules are specified separately under respective `applicable/2` rule, which refer to the same rule label.

```
1 % Art.1 - Agreement
2 constitutiveRule(cr1, agreement).
3 applicable(cr1, agreement) :-
4     party(holder),
5     party(user),
6     covered_data(data).
```

RR1 The obligation to make data available is modelled as a prescriptive rule, and here a variable is introduced in order to capture the time constraint. Specifically, the deadline (**X**) is computed relative to the moment of the request (**Y**), and set to five days afterwards. ASP requires all variables to be grounded during computation, and since in DDL rules are always split between their declaration and their applicability, the grounding conditions have to be repeated in both parts. The same applies when a rule head must also be declared as an `atom`. In practice, the declaration of a rule using variables requires three layers of formalisation. Below the formalisation of Article 3 is shown:

```
1 atom(make_data_available(X)) :- request_data(Y), X = Y + 5.
2
3 prescriptiveRule(ob1, make_data_available(X)) :-
4     request_data(Y),
5     X = Y + 5.
6 applicable(ob1, make_data_available(X)) :-
7     request_data(Y),
8     X = Y + 5.
```

RR2– RR3– RR4 The permission for the User to share data, together with its exceptions, is modelled directly through the defeasible logic of DDL rather than by relying on negation-as-failure, as in Prolog. This is shown in the following code snippet:

```
1 % Art. 5 "User Permitted Uses"
2 permissiveRule(per1, user_share).
3 applicable(per1, user_share) :-
4     defeasible(agreement).
5
6 %% --- gatekeeper prohibition
7 prescriptiveRule(pro1, non(user_share)).
8 applicable(pro1, non(user_share)) :-
9     defeasible(agreement),
10    gatekeeper.
11
12 %% --- gatekeeper authorisation permission
13 permissiveRule(per2, user_share).
14 applicable(per2, user_share) :-
15     defeasible(agreement),
16     defeasible(authorisation).
17
18 %% --- User Sharing priorities
19 superior(pro1,per1).
20 superior(per2,pro1).
```

The code first establishes a general permissive rule granting the User the ability to share the data (line 2). Subsequently, the exception for gatekeepers is expressed as a prohibition, namely a prescriptive rule for the negative action `non(user_share)` (line 7-10). A superiority relation then specifies that this prohibition overrides the general permission, ensuring that the exception has priority in case of conflict (line 19). Finally, a further permissive rule is introduced for the case in which the Data Holder explicitly authorises sharing with a gatekeeper (line 13-16) and an additional superiority relation establishes that this specific permission prevails over the prohibition (line 20), reproducing the layered structure of Article 5.

CR3 In DDL the logic of violation is already built into the system: an obligation to `make_data_available`, for example, is violated when its opposite (`non(make_data_available)`) holds. The system detects this through the built-in predicate `violation/3`. However, the internal definition of violation does not take deadlines into account, which are essential for obligations in this contract. For this reason, specific constitutive rules must be defined to capture the breach of each obligation. In practice, the breach provision in Article 7 is split into multiple rules, each specific to one of the contractual obligations. For example, for the obligation in article 3:

```
1 constitutiveRule(br1, non(make_data_available(X))) :-
2     request_data(Y),
3     X = Y + 5.
4 applicable(br1, non(make_data_available(X))) :-
```

```

5  request_data(Y),
6  X = Y + 5,
7  five_days_expired,
8  not make_data_available(X).

```

This is only required for positive obligations, since permissions and prohibitions do not involve deadlines and can therefore rely solely on the internal semantics. For the breach of the obligation in Article 3, the rule uses negation-as-failure to check that the prescribed action did not occur within the deadline (line 102).

RR6– RR7– CR4 Finally, remedies under Article 7 are modelled as permissive rules, granting the non-breaching party the right to claim damages or to request termination of the contract. Termination itself, as established in Article 8, is captured as a constitutive rule, triggered either by the exercise of the remedy of termination or by the end of the underlying product contract.

Evaluation

Legal Fidelity From a structural perspective, the DDL formalisation exhibits a level of isomorphism almost comparable to that of Prolog and Logical English. Each article of the contract can be mapped to a corresponding rule or small set of rules, preserving the modularity of the original text. However the need to split the representation of a rule between declaration, applicability and variable grounding makes the encoding more verbose and less mappable to the original contract text. Modifying a clause requires updating different parts of the formalization, although often recognizable with the same label. For this reason we deem the level of isomorphism provided by DDL as medium (M).

Semantic richness is the clear strength of DDL as legal patterns such as defeasible and deontic reasoning are integrated in the underlying semantics. Table 9.4 summarises the evaluation of the legal fidelity category:

Isomorphism	Defeasibility		Normative positions	Temporal express.	
	Exc	Confl		Rep	Track
M	✓✓	✓✓	✓✓	✓	✗

Table 9.4: Legal fidelity in DDL

Defeasibility is directly supported by the possibility to define conflicting rules and explicit superiority relations to solve such conflicts. This mechanism makes it possible to introduce new exceptions without having to rewrite the existing rules, which is necessary when defeasibility relies on negation-as-failure. In the contract, this can be seen in the interaction between rules `per1`, `pro1`, and `per2`, which all conflict with one another as they concern either the action `share_data` or its opposite, `non(share_data)`. The superiority relations determine which rule prevails in each case, thereby resolving the conflict smoothly.

DDL also provides specific rule constructs to capture normative positions. Obligations and prohibition are captured by the `prescriptiveRule` construct and permissions through `permissiveRule`. Their use entails a specific meaning in the language semantics and specific legal effects in terms of reasoning. For example, violations are automatically calculated when an atom object of an obligation and its opposite is found in the knowledge base.

Moreover, DDL supports the representation of contrary-to-duty obligations through the predicate `compensate/4`, which links a primary obligation to a secondary one activated upon violation. For instance, in addition to the compensation obligation of Article 6, a CTD clause could impose the obligation to pay interests if the data holder fails to compensate on time:

```
compensate(ob2, compensate(X,T), pay_interests, 1) :-  
    rate(R),  
    generated_data(D, T1),  
    X = D / R,  
    T = T1 + 30.
```

With such a rule, in case a violation of the obligation `ob2`, the system will automatically raise the CTD obligation `pay_interests`.

Finally, the time expressiveness of DDL shares the same limitations as Prolog and LE. Temporal constraints can be represented and manipulated declaratively, for example by calculating deadlines as offsets from a reference date, but the system lacks an internal notion of time or event-handling capabilities. As a result, deadlines and temporal conditions must be checked explicitly, rather than monitored automatically. This approach is sufficient for representing contractual deadlines such as those in Arts. 3 and 6, but becomes less smooth in scenarios involving continuous monitoring or multiple overlapping temporal constraints.

Accessibility From the standpoint of *syntactic commitment* (SC), the language is more rigid than Prolog, in the sense that the formalization must employ a set of predefined constructs such as `constitutiveRule`, `prescriptiveRule`, `permissiveRule`, and `superior`, as well as explicitly distinguish between `atom`, `applicable`, and `fact`. This requirements not only make the formalization more verbose, but also presuppose prior knowledge of these constructs in order to be read and understood.

On the other hand, the predefined syntax directly reflects legally derived notions such as obligations, prohibitions, permissions, constitutive rules, and priorities among them. This results in a comparatively high level of *domain alignment*, as the constructs map closely onto familiar legal categories.

Combining these indicators, DDL is placed in Tier 2. SC is comparatively high, but the strong alignment with contractual concepts makes the system more accessible to domain experts than general-purpose programming languages.

Operational Expressiveness A DDL program consists of constitutive, permissive, and prescriptive rules. Its execution (described by the answer set) produces the *normative state* of the contractual relationship given a set of facts, which includes the obligations, permissions, prohibitions, and potential violations that follow from them. Such production, however, entails the same structural limitations as Prolog and LE in terms of *automatability*. Its execution in ASP does not involve the enactment of behaviours or the management of events over time, but only the generation of stable models consistent with the underlying rules, i.e., a description of the normative status. Moreover, ASP lacks features allowing dynamic updates of the knowledge base such as Prolog’s `assert` and `retract`. Automatability in DDL is thus evaluated as not supported (©).

From the standpoint of *resource handling*, DDL behaves similarly to Prolog. Contractual resources can only be represented symbolically as arguments within predicates, without any built-in mechanism to ensure uniqueness, linearity, or conservation. Transfers or updates can be simulated declaratively by changing the relevant facts, but there are no native constructs to guarantee correct resource management. For this reason, resource handling is assessed as limited (●) .

9.2.4 Logic Production Systems (LPS)

Logic Production System (LPS) [107] was introduced by Kowalski and Sadri [104] to bridge the gap between declarative logic programming and imperative, action-oriented computation. While languages such as Prolog, grounded in first-order logic, support reasoning by deduction, they are less suited for modelling dynamic systems in which actions, events, and temporal evolution play a central role. According to the authors, this limitation has historically hindered the adoption of pure logic programming in areas such as agent-based systems, where imperative languages remain dominant.

To overcome this issue, LPS integrates Logic Programming (LP) with the main features of Production Systems (PS). A PS is a computational model based on forward reasoning, where rules of the form “*if conditions, then action*” are evaluated against a database of facts, and applicable rules trigger changes in the system state [148, 54]. This architecture has been widely used in AI to simulate cognitive tasks and intelligent behaviour. Whereas LP focuses on static inference, deriving conclusions from given premises, PS centres on generating and executing actions. LPS combines both approaches, retaining the logical foundation of LP while adopting the reactive, action-producing nature of PS.

The language incorporates explicit representations of external events, internal actions, and the passage of time, together with reactive rules for state-changing behaviour [105, 106].

LPS is implemented as an extension of SWI-Prolog and is available on the SWISH platform⁸. The web interface allows users to write and execute LPS programs and to visualise the evolution of the system’s state along a timeline.

An LPS program is thus composed of the following building blocks:

- **Fluents.** Fluents represent facts that may hold at a particular point in time and whose truth value can change as the system evolves. Formally, they encode time-dependent properties of the world. For example:

`initially lightOff`

⁸demo.logicalcontracts.com.

declares that the fluent `lightOn` is true at the start of execution. Fluents are modified over time through events.

- **Events.** Events represent external occurrences that are time-stamped. For example:

```
observe switch at 1
observe switch at 2
```

This records the occurrence of a `switch` event at time 1, Events affect the system state according to the defined causal laws.

- **Causal Laws.** These are expressed as normal LP rules of the form “*conclusion if conditions*”. For example, the rules

```
switch initiates lightOn if lightOff
switch terminates lightOff if lightOff
```

describe the effects of the `switch` event on the system’s state, namely turning the light on if the light is off.

- **Reactive Rules and Actions.** LPS extends LP with rules of the form *conditions then action*, which specify behaviours the agent should perform once conditions are satisfied. For example:

```
if lightOn at T then ring from T.
```

This instructs the agent to perform a `ring` action whenever the fluent `lightOn` holds at time *T*. Actions, like events, need to be timestamped.

The combination of causal laws and reactive rules represents the combination between LP and PS. The first are represented by normal traditional horn clauses and capture the agent’s knowledge base. The second, instead, represent the system’s goals and when their conditions are satisfied, they generate actions

To represent time and change of state, LPS builds on Event Calculus (EC), a formalism for reasoning about time and events using logic [126]. In EC, frame axioms are employed to determine whether a fluent holds, i.e., at each step the system evaluates through a logical inference whether the fluent was initiated, and whether it was terminated by an event. Although declaratively elegant, this approach is

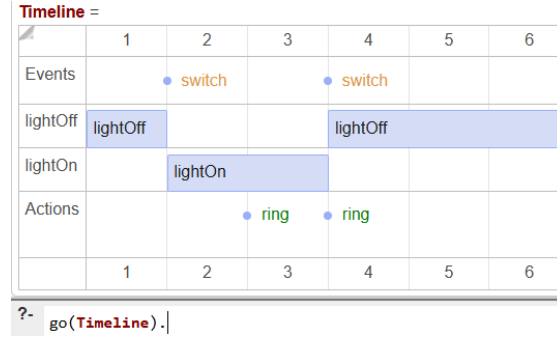


Figure 9.1: The Outcome of an LPS program

computationally expensive. LPS addresses the problem by adopting *destructive updates* [105]: instead of reasoning explicitly with frame axioms, the system keeps an internal state and modifies it when events occur. Fluents are added or removed from the current database according to causal laws, producing a single evolving state rather than a full history of timestamped states.

The resulting overall program, when executed, produces a history of the events, fluents, and actions produced by the system. For example, the rules and fluents defined above would produce the timeline in Figure 9.1.

The Contract

In contrast to other languages seen so far, the contract in LPS is not described as a static set of rules, but as a sequence of evolving states in which events occur and actions are triggered. The full LPS code is available in Appendix A.3. Fluents are used to represent the normative state of the agreement, i.e., constitutive rules, obligations, and permissions. Each of these is, in fact, characterised by a temporal dimension: for example, an obligation is in force only within a specified time interval, the contract is valid within a specific time frame, and so on. *Events* denote external occurrences in the real world, such as acts performed by the contracting parties. *Actions* correspond to consequences automatically generated by the system in response to certain events (e.g. discharging an obligation, recording a breach, or enforcing a remedy).

CR1 In LPS, the constitutive rule establishing the agreement is represented by the fluent `active_contract/3`, indicated in the code snippet below. This is initiated by the event `agreement/3` that represents the meeting of the minds among the parties. This fluent marks the validity of the contractual relationship.

```
1 % Art.1 - Agreement
2 observe agreement(carMaker, carRental, covered_data) at 2.
3 agreement(Holder, User, Data) initiates active_contract(Holder, User, Data).
```

RR1 The duty of the Data Holder to grant access to the data is represented by the fluent `obligation_to_make_available/3`, initiated when the User issues a data request (Lines 2-3 of the code snippet below).

```
1 % Art.3 - Data Access Right
2 request_data(User,Holder,Data,T) initiates obligation_to_make_available(Holder,
   User, D)
3   if access_deadline(Deadline1), D is T + Deadline1.
4
5 if   obligation_to_make_available(Holder, User, DueDate) at T,
6   makes_available(Holder, User, Date) from T to T1,
7   Date <= DueDate
8 then obligation_fulfilled(Holder, User, make_available) from T1.
9
10 obligation_fulfilled(Holder, User, make_available) terminates
   obligation_to_make_available(Holder, User, D).
```

The LPS formalisation also requires the definition of the termination of fluents. In this case, this is implemented through a reactive rule. Once the Holder makes the data available before the due date, represented by the event `makes_available/3` (line 6), a reactive rule discharges the obligation by triggering the action `obligation_fulfilled/3`. This action then terminates the corresponding fluent, ensuring that the obligation is no longer in force once it has been properly executed (line 10).

RR2– RR3– RR4 Permissions, like obligations, are represented by fluents (e.g. `permission_to_share_data/1`). The permissions defined by Articles 4 and 5 concerning data use and sharing are initiated by the `agreement/3` event. However, Articles 4 and 5 introduce specific exceptions to these permissions.

Because of the features of LPS, modelling such exceptions is not as straightforward as in previous languages. Since fluents in LPS describe the general state of the

system, in fact, they cannot capture exceptions that apply only in specific cases. These are therefore represented through reactive rules, which specify whether the system allows or prevents certain actions from being executed, thus capturing the operational effect of exceptions. For example, the exception to the user's permission to share data with gatekeepers is expressed through negation-as-failure in a reactive rule, as shown below:

```

1 agreement(Holder, User, Data) initiates permission_to_share_data(User).
2
3 if    permission_to_share_data(User) at T,
4      asks_to_share(User, ThirdParty) from T to T1,
5      not gatekeeper(ThirdParty)
6 then happens(User, ThirdParty, share_data) from T1.

```

Thus, the fluent captures the standing permission, while the reactive rules ensure that concrete actions are performed only under the conditions specified by the contract.

RR5 As with the obligation to make data available, of the compensation duty of Article 6 is represented by a fluent. Here, however, the reactive features of LPS prove particularly useful.

A reactive rule is used to implement an automated calculation of the data generated by the product. Specifically, every month (represented by the event (month_tick/1)) the action calculate_mb/2 is triggered. This action, in turn, automatically initiates the fluent obligation_to_compensate/4, which encodes both the amount due and the corresponding deadline for payment (lines 5-9 of the excerpt below). Finally, as for article 3, the fulfilment of the obligation terminates the relevant obligation fluent.

```

1 % Art. 6 "Compensation for Commercial Use"
2 if    month_tick(D), generated_data(Number) % - D is the date of the tick; N is
3      the size of data.
4 then calculate_mb(Number, D).
5
6 calculate_mb(Number, D) initiates obligation_to_compensate(Holder, User, Amount,
7      DueDate)
8 if    active_contract(Holder, User, _Data),
9      mb_rate(Rate),
10     Amount is Number * Rate,
11     DueDate is D + 30.

```

CR3 Obligation breaches are defined in a straightforward way, thanks to the temporal structure of fluents. In fact, each obligation fluent explicitly includes its deadline as one of its arguments: if the fluent is still active after the deadline has expired, a breach is detected. This is implemented through a reactive rule, for instance:

```
1 if    obligation_to_make_available(Holder, User, D1) at T,
2      tick(D) from T,
3      D > D1
4 then  breach(Holder).
```

RR7– RR6– CR4 Remedies are modelled as permissions: When the breach fluent holds true, the system enables parties to perform the actions related to the remedy. For example:

```
1 if    obligation_to_make_available(Holder, User, D1) at T,
2      tick(D) from T,
3      D > D1
4 then  breach(Holder).
```

For the purposes of this example, these actions are left as abstract names corresponding to the remedies they represent. In a real computable contract, however, such actions would automate concrete effects: for instance, contract termination could refund each party with its own money or goods, while a damages claim could release an escrowed amount to the entitled party.

Finally, the termination of the contract is formalised by simply terminating the `active_contract` fluent as a consequence of either remedy or the natural end of the underlying product contract. When the action `terminates_contract/3` occurs, it terminates all fluents associated with the active relationship. This ensures that once the contract ends, no further obligations or permissions remain in force.

By running the LPS contract code presented Listing [A.3](#) on the SWISH interpreter, one can obtain the visual representation of evolving state of the contract (in Figure [9.2](#)), with the indication of observed events, in the top row, and the triggered actions, in the bottom row.

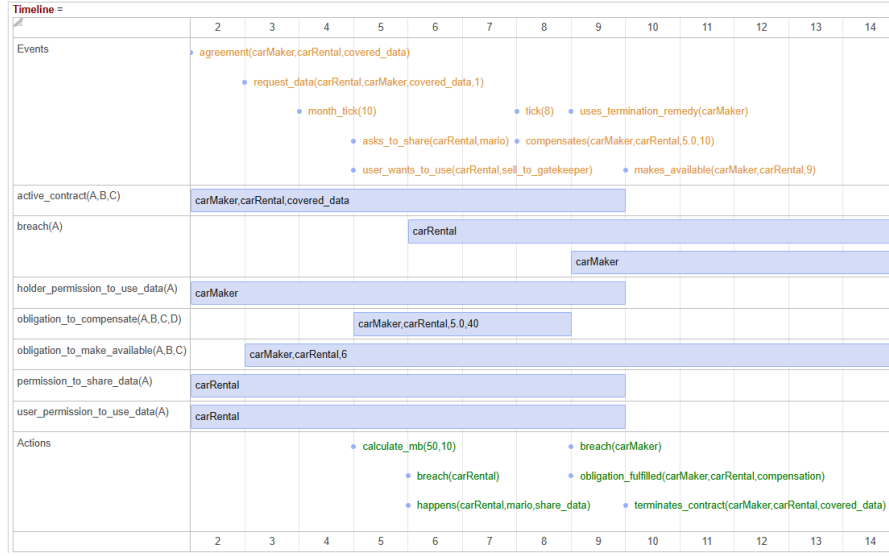


Figure 9.2: Execution of the LPS D.U.A

Evaluation

Legal Fidelity From a structural standpoint, the representation of the contract in LPS exhibits a rather low level of isomorphism (L). This is because, as mentioned above, the program does not formalise simply contract rules, but rather the life cycle of the system, thereby requiring some additional logic that is not present in legal text. Provisions are not encoded as single, identifiable rules but as combinations of different constructs that together reproduce the intended logic. For example, Article 3 (*RR1*) is distributed across: (i) the rule that initiates the obligation fluent, (ii) the reactive rule that specifies when the obligation is fulfilled, and (iii) the rule terminating the obligation fluent. The need to split a single contractual clause into multiple semantically different elements makes the link with the legal source less direct and reduces structural fidelity.

From a semantic standpoint, LPS can represent the key dimensions of legal reasoning, although with different degrees of effectiveness. Defeasibility can be modelled in a LP fashion using NAF, without further built-in conflict-handling mechanisms. However, the fact that specific exceptions cannot be specified on fluents, but only on specific actions through reactive rules, results in more verbose formulations, as seen in the permissions of Articles 4 and 5 (*RR2*, *RR3*).

As for normative positions, LPS does not natively embed a deontic logic seman-

Isomorphism	Defeasibility		Normative positions	Temporal express.	
	Exc	Confl		Rep	Track
L	✓	✓	✓	✓✓	✓✓

Table 9.5: Legal fidelity in LPS

tics. Obligations, permissions, and prohibitions are represented through dedicated fluents and reasoning over them is enabled by specifying the necessary rules. For example, the breach of an obligation can be identified simply by checking whether the related fluent is still active after the deadline is expired.

Finally, LPS natively incorporates temporal reasoning within its semantics, which constitutes one of its main strengths. Fluents represent states that persist over time, and all core constructs—fluents, actions, and events—are temporally indexed through timestamps. In the current implementation, these timestamps refer to an abstract timeline represented by natural numbers, while concrete temporal references (such as specific dates or calendar days) must still be expressed declaratively, in a traditional logic programming fashion. Nonetheless, the framework is, in principle, capable of supporting real-time processing, which would significantly enhance its practical applicability.

Table 9.5 reports the overall evaluation of LPS for the legal fidelity category.

Accessibility LPS can be evaluated as a Tier 3 language. From the standpoint of *syntactic commitment* (SC), the language is more prescriptive than Prolog, as it requires the use of predefined constructs such as `fluent`, `action`, `event`, and different rules structures such as `initiates` and `terminates` (causal rules) and if-then (reactive). This results in a medium - high level of SC: the modeller enjoys some flexibility in defining predicates, but the overall structure of the program must conform to a fixed set of categories.

By contrast, *domain alignment* (DA) is low. Although the LPS framework is expressive, its constructs do not directly mirror legal concepts such as obligations or rights, but rather computational notions related to state transitions and event processing. As a result, legal experts would find it challenging to map the code to contractual provisions, especially because single clauses are often scattered across

multiple constructs.

Combining these indicators, LPS falls within Tier 3. Its prescriptive syntax and low domain alignment make it difficult for non-technical experts to grasp the relationship between the code and the legal text.

Operational Expressiveness Operational expressiveness is the dimension where LPS performs best, thanks to its PS semantics. The framework maintains an internal state described by *fluents*, which evolve dynamically through destructive updates [105]. This makes it possible to capture both the actions performed by the parties (as observable events) and the automated actions that the system itself triggers in order to give effect to contractual clauses.

Therefore, LPS provides strong support with respect to *automatability*. The interaction between events, states, and actions is modelled directly through causal laws and reactive rules: for example, an external event may activate an obligation, which in turn triggers an action that updates the contractual state. Through these explicit causal chains, LPS can naturally represent the contract’s dynamic evolution in response to external stimuli—an essential feature for modelling the full life cycle of computable contracts.

More generally, the framework lends itself well to practical automation. External events can be fed into the system from sources such as IoT devices or databases, and the actions generated by the system can correspond to concrete real-world effects, such as executing a payment or blocking access. In conclusion, automatability in LPS is evaluated as full (●).

From the standpoint of *resource handling*, LPS behaves similarly to other logic-based languages. Contractual resources can only be represented symbolically as predicates or arguments, without any built-in resource-handling mechanism. Transfers or updates must therefore be modelled declaratively, and the system does not natively enforce the integrity of contractual goods. Resource handling is thus assessed as limited (●).

9.3 Object-Oriented Imperative Languages

9.3.1 Solidity

Solidity⁹ is a high-level programming language specifically developed for the Ethereum Virtual Machine for developing smart contracts. The language was introduced in 2014 [25] and remains the dominant language for programming on Ethereum and other compatible blockchains.

Solidity adopts an imperative, object-oriented paradigm, syntactically close to languages such as Java or C++, and it is statically typed [125]. The language supports both global and local variables. Global variables, i.e., state variables, are stored permanently on the blockchain and define the persistent state of the contract, whereas local variables are created temporarily during function execution and discarded once the function terminates.

Contracts in Solidity are defined in a way that resembles object-oriented classes: they contain state variables, functions, events, and may inherit elements from other contracts¹⁰. A Solidity program is mainly composed by a set of functions that express the operational logic of the contract. They may be internal, i.e., only callable within the contract itself or its descendants, or external, publicly exposed and callable by any address.

Beside functions, the language presents several features. We mention some of them that are important in the formalisation of the case study.

- *Enumerations (enums)* allow developers to define custom finite types, improving the readability of the code and the clarity of state transitions;
- *modifiers* are special function decorators that specify conditions under which a function can be executed, for example to restrict access to certain addresses or to enforce preconditions before executing the function's body;
- *events* are constructs that allow contracts to log information during execu-

⁹soliditylang.org

¹⁰However, the analogy is not perfect: while classes in traditional programming can be instantiated multiple times within a program, a Solidity contract is instantiated only when it is deployed on the blockchain, at which point it is assigned a unique address and becomes a persistent account.

tion, which is then stored in the transaction logs of the blockchain. These are commonly used to record significant occurrences, such as payments, transfers of tokens, or changes of state, providing a communication channel between the on-chain execution of a contract and off-chain components;

- solidity allows the definition of user-defined data structures through **structs**, as well as the use of *special global variables* provided by the Ethereum environment, such as **msg.sender** (the address of the caller) or **block.timestamp** (the timestamp of the current block);
- the **require** statement is a built-in mechanism used to check conditions at runtime: if the condition evaluates to false, the execution halts and all changes to the state are reverted, with the unused gas refunded [69].

Together, these constructs contribute to making Solidity expressive for implementing both the operational logic and the safeguards of contractual interactions.

A crucial aspect of Solidity is its integration with Ethereum’s execution model, which is governed by the so-called “gas economy”. Every operation and transaction consumes a certain amount of gas, which must be provided by the account submitting the transaction. This mechanism ensures that computational and storage resources are economically accounted for, while also preventing abuse: if the gas allocated to a transaction is exhausted, execution halts and all changes are reverted.^[11]

The Contract

The full contract in Solidity is shown in Appendix A.4^[12]. Unlike the declarative formalisations seen with logic-based approaches, the Solidity implementation does not describe the contract as a set of rules, but as a sequence of executable instructions that are triggered either by party interactions or by the satisfaction of certain conditions.

¹¹This mechanism prevents infinite loops and other forms of non-termination, thereby addressing the halting problem in practice by bounding execution with economic costs.

¹²Although this is a fully working implementation that can be tested in environments such as Remix^[13], a real deployment on a public blockchain would require additional security measures to protect against malicious behaviour. For the sake of clarity and brevity, such aspects are omitted here.

Operational Setup Parties’ addresses as well as key contract parameters (type of data covered, price per MB, access deadline, and contract end date) are declared as state variables and persist throughout the contract’s duration.

A specific design choice was to represent all normative positions (obligations and permissions) as data structures. A unified record, `NormPosition` (lines 5-13 of the code snippet below), encodes both obligations (e.g. data delivery, compensation) and permissions.

```
1  // NORMATIVE POSITIONS
2  enum NormPositionType { DataRequest, Compensation, Permitted_uses }
3  uint256 constant NO_DEADLINE = type(uint256).max;
4  enum NormPositionStatus { active, fulfilled, breached }
5
6  struct NormPosition {
7      uint256 id;                // unique incremental identifier
8      NormPositionType oType;    // type of normative position (data request,
9                                // compensation, ect)
10     string content;            // description
11     address obligor;           // party bound to perform
12     uint256 deadline;          // timestamp of expiry
13     NormPositionStatus oStatus;
14     uint amountDue;            // optional amount to be paid
15     string data;               // optional data delivered
16 }
```

Each `NormPosition` struct includes a type, the party it refers to, its current status (`active`, `fulfilled`, or `breached`), and optional fields such as deadline, amount due, and content. While permissions cannot be acted upon through the contract, they act as anchors for off-chain evaluation and breach signalling. An internal function allows new normative positions to be created (lines 90-108). This approach shows the operational nature of Solidity: obligations are not abstractly reasoned upon, but instantiated and tracked as concrete data records.

The life cycle of the agreement is organised through a state-machine pattern defined through the `enum State`. It defines the contract phases `Inactive`, `Active`, `Breach`, `Dispute`, and `Ended`. This allows functions’ invocation to be restricted only to the appropriate contract phase, through ad hoc modifiers.

In our set up, the Data Use smart contract interacts with an external contract which controls the digital representation of the rented vehicles as a non-fungible token (NFT). The NFT contract, compliant with the ERC-721 standard, associates each vehicle with a unique token ID and provides functions that can be called by the agreement. In particular, the `transferCar` function is used to transfer

ownership of the vehicle to or from the agreement, thereby ensuring that the asset is under the control of the contract while the agreement is in force. The NFT contract also exposes the function `getDataAmount`, which returns the quantity of data generated by the vehicle during a given period. An interface is defined to link the two contracts:

```
1 interface ICarNFT {
2     function transferCar(address to, uint256 tokenId) external;
3     function getDataAmount(uint256 tokenId) external view returns (uint256);
4     function approve(address to, uint256 tokenId) external;
5 }
```

Through this integration, the Data Use Agreement is not an isolated piece of code, but part of a broader infrastructure in which the asset itself (the connected car) is represented digitally and can be controlled by the smart contract.

To begin the contract life cycle, the `start()` function is defined. Unlike other functions, it does not correspond to any provisions of the contract text but is required for the operational deployment of the contract.

```
1 function start(uint256 _carTokenId) external onlyCarMaker inState(State.
2   Inactive) {
3     carTokenId = _carTokenId;
4     carNFT.transferCar(address(this), carTokenId);
5
6     _createNormPosition( // Art. 4 "Holder Permitted Uses"
7       NormPositionType.Permitted_uses, carMaker,
8       "only use for art4 purposes", NO_DEADLINE,
9       0, ""
10    );
11    _createNormPosition( // Art. 5 "User Permitted Uses"
12      NormPositionType.Permitted_uses, carRental,
13      "Do NOT share with gatekeepers", NO_DEADLINE,
14      0, ""
15    );
16    currentState = State.Active;
17 }
```

When invoked, the Data Holder transfers the NFT representing the vehicle to the contract, ensuring that the underlying asset is placed under the contract's control. At the same time, the function instantiates the normative positions defined in Articles 4 and 5, which represent the standing permissions and prohibitions on data use and sharing throughout the contract's validity. Finally, it changes the state from `Inactive` to `Active`, marking the agreement as operational.

RR1 Article 3 is implemented through the functions `requestData` and `deliverData` (lines 141 and 152) The invocation of the first triggers the creation of the struct representing the obligation to make data available within five days. Once created, the contract state changes from `Active` to `RequestPending`.

```
1 // Art.3 "Data Access Right"
2 function requestData(string calldata content) external onlyCarRental inState(
  State.Active) {
3   _createNormPosition(
4     NormPositionType.DataRequest, carMaker,
5     "data request ex. art 3", block.timestamp + accessDeadline,
6     0, content
7   );
8 }
```

The second function allows the Data Holder to fulfill the request, by delivering the requested data. Accordingly the `NormPosition` just created is marked as fulfilled.

RR2– RR3– RR4– CR3 In declarative languages such as Prolog or LPS, permissions and prohibitions can be explicitly represented as rules and reasoned upon. In Solidity, by contrast, contractual provisions regulating party behaviour can only be expressed in terms of observable on-chain actions. Since acts of using or sharing data occur off-chain (e.g. the Holder or User could freely copy the data to another medium), such behaviour cannot be directly monitored by the contract. For this reason, encoding permissions and their exceptions as enforceable rules on-chain would be ineffective. These aspects are only captured indirectly with functions allowing a party to signal a breach of the relevant permissions (line 215) and change the corresponding `NormPosition` status to `breached`. The evaluation of exceptions, and thus the determination of whether a breach has occurred, however, is deferred to the off-chain domain, possibly involving human actors or external authorities, as the reasoning is not carried out by the contract.

RR5 Similarly to Article 3, the representation of the obligation to compensate in Article 6 is split across two functions. One creates the `NormPosition` related to the compensation obligation. To fulfill this obligation the holder can invoke the `compensate` function through which they can transfer the required amount in Ether. If the payment matches the amount due, the struct is updated and the

obligation marked as fulfilled. The `compensate` function is shown below.

```
1 function compensate(uint256 norm_positionId) external onlyCarMaker payable {
2   uint256 i = _idx[norm_positionId];
3   require(i != 0, "NormPosition not found");
4   NormPosition storage ob = norm_positions[i - 1];
5
6   require(ob.oType == NormPositionType.Compensation, "Not a compensation
7     normative position");
8   require(ob.oStatus == NormPositionStatus.active, "Already fulfilled");
9   require(msg.value == ob.amountDue, "Incorrect ETH sent");
10
11   // transfer ETH to beneficiary
12   (bool ok, ) = carRental.call{value: msg.value}("");
13   require(ok, "Transfer failed");
14
15   ob.oStatus = NormPositionStatus.fulfilled;
16   emit NormPositionFulfilled(ob.id, ob.obligor, ob.content);
17 }
```

CR3 Article 7, which defines breach of contract, is implemented through two sets of functions. The first concerns breach detection. Specifically `markBreach` functions records violations of the permissions established in Articles 4 and 5 which are invoked by parties (as explained in the above paragraph *RR2–RR3*). By contrast, `checkObligations` monitors time-bound obligations such as data delivery and compensation and detects breaches when their status is still `active` and the deadline is expired.

```
1 function checkObligations() external { // Breach of arts. 3 and 6
2   uint256 n = norm_positions.length;
3   for (uint256 i = 0; i < n; i++) {
4     NormPosition storage ob = norm_positions[i];
5     if (ob.oStatus == NormPositionStatus.active && block.timestamp > ob.
6       deadline) {
7       ob.oStatus = NormPositionStatus.breached;
8       currentState = State.Breach;
9     }
10  }
```

Ideally, such a check should occur automatically as the obligation deadline expires without it being fulfilled. However, smart contracts on the EVM are not continuously running: they remain inactive until a transaction triggers one of their functions[52]. As a result, time-dependent checks cannot occur automatically in the background, but must be initiated explicitly by a transaction, either by one of the contracting parties or by an external service (i.e., oracle) that calls the relevant function at fixed intervals.

Evaluation

Legal fidelity. The level of legal fidelity achieved in Solidity is generally limited. The overview is shown in Table 9.6.

Isomorphism	Defeasibility		Normative positions	Temporal express.	
	Exc	Confl		Rep	Track
L	✓	✗	✓	✓✓	✗

Table 9.6: Legal fidelity in Solidity

From the standpoint of structural fidelity, Solidity performs poorly. The code is considerably more verbose than in the other languages, often twice as long, since it requires the declaration of numerous structures, variables, enumerations, and types that have no direct counterpart in the contract text but are necessary for the program to run correctly. Moreover, parts of the logic implemented in Solidity do not correspond to any specific provision of the legal text. Examples include the `start()` function, or the internal functions used to create the `NormPosition` structures. For these reasons, the level of structural isomorphism is low (L).

From a semantic perspective, Solidity shows significant limitations. With regard to *defeasibility*, the language does not provide any native mechanism to represent exceptions or to resolve conflicts among rules. In practice, the execution of a function can be constrained through `require` statements or `if-else` blocks. This allows simple exceptions to be represented (✓), but offers limited support for more structured conflict resolution (✗).

In the case study, however, the exceptions of Articles 4 and 5 cannot be encoded directly on-chain, as they refer to actions that take place off-chain and remain invisible to the contract. The code can only provide logic to record or signal breaches once such determinations have been made externally. This is not a limitation of the language itself, but of computable contracts in general: Solidity has no access to facts that take place outside the execution environment. However, if the relevant actions were performed on-chain through dedicated functions, conflict resolution could be only dealt with by a combination of constraints to function calls (via `require` statements or `if-else` blocks).

Regarding normative positions, these were represented through **structs**. This approach allows their status to be tracked within the contract; however, these records do not possess any intrinsic legal semantics, as the language itself lacks concepts such as obligations, permissions, or rights. Reasoning over them is therefore possible only through explicitly defined functions. For instance, in the Data Use Agreement, the **check_obligations** function serves this purpose by iterating over the stored records to verify and update their status.

Finally, with regard to temporal reasoning, Solidity provides convenient access to temporal information through global variables such as **block.timestamp**, and manipulating deadlines or calculating future dates is straightforward in an imperative style. However, Solidity is constrained by the transactional nature of the EVM: time does not progress autonomously, and no operation can be triggered merely by the passage of time. Every temporal check must be invoked through a transaction, whether by one of the parties or by an external oracle. To conclude, while Solidity can represent temporal information effectively, it remains limited in attaching legal or operational effects directly to the progress of time.

Accessibility As a general-purpose programming language, Solidity is not designed with legal experts in mind. Understanding or modifying the code requires technical knowledge of blockchain environments, the Ethereum execution model, and the Solidity syntax. The language’s *syntactic commitment* (SC) is very high: the modeler must rely on a rigid grammar, data types, modifiers, and control structures, as well as technical constructs such as events, mappings and structs. This prescriptiveness leaves little room for tailoring the syntax to legal notions.

By contrast, *domain alignment* (DA), seems very low. The constructs it provides (functions, state variables, structs, enums etc.) are system-level abstractions that do not directly correspond to contractual concepts. Even when the contract stores obligations as data structures (e.g. the **Obligation** struct), these remain artificial encodings rather than native representations of deontic modalities. As a result, the connection between legal clauses and code is weak: while one can broadly match articles with functions, the operational details (**require** checks, state transitions, or oracle calls) remain difficult to interpret for non-programmers.

Combining these indicators, Solidity clearly falls within Tier 3. Its high syn-

tactic commitment and very low domain alignment make it inaccessible to legal experts without significant technical expertise.

Operational Expressiveness Operational expressiveness is the dimension where Solidity performs best. Unlike declarative approaches, a Solidity contract is not a collection of abstract rules but an imperative program that specifies concrete actions to be executed and their consequences. This allows the agreement to capture directly the transactional and operational aspects of the legal text.

From the standpoint of *resource handling*, Solidity provides full support. The language enables the creation and management of digital assets through smart contracts that implement specific token standards—such as ERC-20 for fungible tokens and ERC-721 for non-fungible tokens. These contracts govern the issuance, transfer, and ownership of tokens, ensuring essential properties like uniqueness, scarcity, and non-duplication. Resource handling is therefore assessed as fully supported (●).

As to *automatability*, the imperative nature of the language makes it possible to attach effects directly to actions: updating a field, transferring ownership of a token, or moving Ether between accounts immediately changes the internal state of the contract. Automation is therefore natively supported: for example, invoking the `remedy` function with the termination option immediately causes the NFT to be returned to the Data Holder, while the execution of `calculateGeneratedData` leads directly to the creation of a compensation obligation. The only limitation arises with time-bound clauses, since the EVM cannot act spontaneously. In these cases, functions such as `checkObligations` must be invoked externally by a party or by an oracle to update the status of expired positions. Despite this restriction, automatability in Solidity is evaluated as *full* (●), as the language intrinsically supports the automation of operational clauses and their consequences.

9.3.2 Stipula

Stipula [47] is a domain-specific language for legal agreements. Its main features have been extensively presented in Part II of this dissertation. In this comparative context, the use of *Stipula* is particularly interesting as it represents an imperative

framework that aims at capturing the semantics of contract law, which is instead often the focus of declarative logic-based approaches.

The Contract

The full formalization of the Data Use Agreement in *Stipula* is shown in Appendix [A.5](#).

The contract begins by declaring the necessary assets and the fields . The asset `car` will store the digital representation of the vehicle object of the contract while the other fields will store the main contractual parameters relevant for the implementation.

```
1 stipula DataUseAgreement {  
2   asset car  
3   field covered_data, mb_price, data_req, access_deadline, data_volume_mb,  
    comp_deadline, end_date, breacher
```

CR1–CR2 Execution begins with the *agreement* constructor. Here, the parties agree on the terms of the agreement, namely the data covered by the contract, the price per MB, and the relevant deadlines. This part maps the constitutive rules grounding the whole contract, namely articles 1 and 2.

```
1 // Agreement Setup: Art. 1,  
2   agreement(carMaker, carRental) {  
3     carMaker, carRental : covered_data, mb_price, access_deadline,  
        comp_deadline, end_date  
4   } ==> @Inactive
```

Similarly to the Solidity implementation, the execution then proceeds to the `start()` function, to be invoked by the `carRental` which places the contract in control of the car token.

RR1–CR3 The obligation to provide data of Article 3 is implemented through two functions. First, the `requestData(x)` function (callable by the User) schedules an event at `now + access_deadline`. This is used to automatically detect the breach of this obligation. If, when the deadline expires, the contract is still in the `@Req` state, this event records the Data Holder as the breacher in the `breacher` field and moves the contract to `@Breach`.

```
1 // Art.3 "Data Access Right"
2 @Run carRental: requestData (x)[] {
3   x -> data_req;
4   now + access_deadline >> @Req{           // Obligation to make data available
5     "dataHolder" -> breacher
6   } ==> @Breach
7 } ==> @Req
8
9 @Req carMaker: deliverData(d, req)[] (req == data_req) {
10   d -> carRental
11   0 -> data_req;
12
13 } ==> @Run
```

To prevent this outcome, the Data Holder may invoke `deliverData(d, req)`. Here, `d` represents the data to be delivered, and the field `req` must match the pending request. If the condition is satisfied, the request is fulfilled, the field is cleared, and the contract returns to the `@Run` state.

RR2–RR3–RR4–CR3 Permissions and prohibitions under Articles 4 and 5 cannot be effectively monitored by the contract itself, since the relevant actions (e.g. using or sharing data) occur outside the contract execution environment. Here the same considerations made for Solidity apply. Accordingly, they are modelled from the perspective of breach reporting. Two functions allow the parties to report unauthorised sharing by the User, or non-permitted use by the data holder. In both cases, the contract records the breacher and moves to the `@Breach` state. For instance,

```
1 // -- Breach of Permitted Uses arts 4 and 5
2 @Run carMaker: breach(x)[] (x == 'gatekeeper_share') {
3   x -> carRental
4   "user" -> breacher;
5
6 } ==> @Breach
```

RR5 The obligation to compensate is implemented following the same structure of Article 3 (*RR1*). One function is used to schedule an event that checks the fulfillment of the obligation at the expiration of the deadline.

Another function allows the Data Holder to fulfil the obligation by transferring the required amount.

RR6– RR7 Once the contract enters the `@Breach` state, depending on the identity of the breacher, parties are entitled to remedies, by invoking the relevant function, as shown below.

```
1  @Breach carMaker: terminate() [] (breacher == "user"){
2      "termination" -> carRental
3      car -o carMaker;
4
5  } ==> @End
6
7  @Breach carRental: claim_damages(x) [] (breacher == "dataHolder"){
8      "damages" -> carMaker;
9
10 } ==> @Dispute
```

Where termination is invoked, the digital representation of the car is returned to the carRental and the contract terminated. Alternatively, the claim of damages transfers the contract to the `@Dispute` state, where an adjudication mechanism can be modelled.

Evaluation

Legal Fidelity As in Solidity, the Stipula implementation does not represent the contract as rules, but as executable actions. This inevitably reduces the degree of isomorphism when compared to logic-based approaches. For example, the function `start_product()` has no direct correspondence in the legal text, and some contractual duties (such as Article 3) are split into two separate functions (`requestData` and `deliverData`). However, the amount of code not directly mappable to the contract text is much smaller than in Solidity. Stipula provides native constructs for states and events, so there is no need for helper functions to create obligations or for imperative declarations of the state machine. As a result, the life cycle of the contract is visually and structurally easier to follow, since functions are explicitly tied to states that clearly reflect the contractual phases. Although Stipula does not reach the levels of structural isomorphism of purely logical languages, it achieves a higher level than Solidity, which we assess as medium (M).

As in Solidity, *Stipula* does not provide a general mechanism to represent defeasibility. Exceptions can only be encoded through conditional patterns tied to specific functions, which makes their representation case-by-case rather than a property of the language itself. Nevertheless, *Stipula* introduces two specific features

Isomor.	Defeasibility		Normative pos.	Temporal express.	
	Exc	Confl		Rep	Track
M	✓	✗ Import priorities Non-determ detection	Perm/Pro✗ Obl✓✓	✓✓	✓✓

Table 9.7: Legal fidelity in Stipula

that can partially assist in resolving conflicts among competing clauses, though not emerging in the present scenario. The first is static analysis: the Stipula Analyser is capable of detecting instances of *non-determinism* (Chapter 6), namely situations where two functions satisfy the same invocation requirements in terms of state, party, and constraints. While this does not amount to a runtime resolution mechanism, it allows the drafter to identify inconsistencies at design time and adjust the code to avoid conflicts. The second is the `import` construct, used for contract integration (Chapter 7). This feature allows one to define priorities between contractual clauses and external statutory rules, which can be regarded as a partial form of defeasibility, albeit not within the same contract. Overall, support for defeasibility in Stipula is limited, but these features provide some tools to address targeted types of conflicts.

The treatment of normative positions reveals notable differences between *Stipula* and Solidity. In terms of *permissions* and *prohibitions*, the two languages are operationally similar: in both, the execution of a function can be conditioned on the contract being in a specific state or on the satisfaction of a given condition, thereby enforcing prohibitions in practice. From a representational standpoint, however, Solidity is more expressive, as it allows the definition of structured data objects (`structs`) to model and trace normative positions, even though these lack intrinsic reasoning capabilities. *Stipula*, by contrast, currently supports only simple data fields, which limits its overall expressive power. As a result, permissions and prohibitions concerning actions within the contract’s execution environment can be enforced only implicitly, through state constraints, rather than via explicit representational constructs.

However, the situation is different for *obligations*. In Solidity, obligations are

stored as records whose fulfilment or breach must be checked manually through dedicated functions. In *Stipula*, obligations are represented more directly through *events*: when an obligation arises, an event is scheduled to verify its fulfilment at the relevant deadline and, depending on the outcome, either enforce the obligation or trigger the corresponding contractual consequences (e.g. a transition to the `@Breach` state). This mechanism provides a more natural and legally faithful representation of obligations, enabling reasoning about compliance in a way that goes beyond Solidity's purely data-oriented approach.

Stipula also features a more straightforward handling of time. The primitive `now` can be combined with time expressions, and events can be scheduled to fire automatically upon expiration of deadlines. This means that contractual consequences (such as breach or termination) can be triggered without the need for external function calls. The presence of an internal clock and automatic event handling in *Stipula* therefore makes temporal reasoning more natural and closer to the way deadlines operate in legal contracts.

Accessibility With respect to accessibility, *Stipula* can be placed in Tier 2. From the standpoint of *syntactic commitment* (SC), the language is more prescriptive than Prolog, for example, but less rigid than Solidity. The modeller must use predefined constructs such as `asset`, `field`, explicit `states`, and transactional symbols for obligations and transfers. Functions and events also follow specific syntactic patterns. At the same time, however, many elements of the contract are customizable: the names of states, functions, fields, and assets are freely defined by the user, without the need to declare them imperatively in advance. For instance, unlike Solidity, there is no need to declare the state machine as an `enum` or to enforce party roles through modifiers or `require` statements. This combination places *Stipula*'s SC at a medium level.

As to *domain alignment* (DA), *Stipula* provides several constructs that directly reflect contractual concepts. The `agreement` clause mirrors the contract formation, states represent the phases of the contractual life cycle, and events encode the occurrence of obligations and remedies. These features increase the familiarity of the code for legal experts. Nevertheless, other constructs such as `field` and `asset`, as well as the imperative body of functions, are system-oriented and do not

correspond directly to legal categories. For this reason, domain alignment is also evaluated as medium.

Overall, *Stipula* falls within Tier 2. Its syntactic rules are more prescriptive than those of Prolog, but its strong contractual alignment makes it significantly more intelligible than general-purpose languages like Solidity.

Operational Expressiveness *Stipula* features all the advantages of the imperative paradigm with respect to operational expressiveness. Contracts are represented as executable actions with concrete effects rather than as abstract rules. This makes it straightforward to model contractual transactions, such as the transfer of assets, the performance of obligations, and the handling of information.

From the standpoint of *resource handling*, *Stipula* provides full support (●). The *asset* primitive allows the faithful modeling of linear resources such as currency or goods managed by the contract. The formal semantics ensures that these resources are never duplicated or lost, and static analyses can be performed to guarantee liquidity and avoid freezing assets.

As to *automatability*, *Stipula* also provides full support. The integration of *events* makes it possible to attach effects—such as enforcement measures or penalties—directly to specific phases of the contract life cycle. For example, if a data request is not satisfied within the agreed deadline, an event automatically marks the Data Holder as the breacher and moves the contract into the `@Breach` state. Similarly, payment deadlines are enforced natively without the need for external calls. This eliminates the reliance on oracles for time-dependent obligations, a limitation of Solidity, and makes *Stipula* particularly effective for automation. Automatability in *Stipula* is thus evaluated as *full* (●).

9.4 Discussion

The comparative evaluation of the six languages, summarised in Table 9.8, highlights a series of transversal patterns and trade-offs across the three main criteria: legal fidelity, accessibility, and operational expressiveness. Rather than reiterating the individual assessments, this section highlights the general tendencies that emerge from the comparison.

One clarification is necessary at the outset. The comparative table must be read as a reference tool that captures broad tendencies rather than fine-grained distinctions. By necessity, it abstracts away from the many nuances that differentiate one language from another, and cannot reflect the full complexity of their syntax, semantics, or tooling. Whenever relevant, these subtleties will be discussed in the following sections. The purpose of the table is to provide an intuitive overview of whether and to what extent certain macro-properties are supported, thereby offering a basis for comparison. Importantly, the evaluation is not meant to suggest that one language is “better” than another. Each language reflects a different balance of design choices, and the goal of this discussion is to identify their respective characteristics and the trade-offs they entail, not to rank them in absolute terms.

9.4.1 Legal Fidelity

Isomorphism. Looking at isomorphism, it is clear that logic-based languages generally allow a high level of correspondence between the contract and its formalization. This is because declarative languages are designed to model relations between entities, rather than executable instructions. As a result, individual clauses and their dependencies can be preserved more directly, whereas imperative languages tend to flatten contractual provisions into sequences of actions and effects.

An interesting pattern emerges, however, when considering the presence of domain-specific constructs. The level of isomorphism tends to decrease as languages provide more specialised features. This is evident in DDL and LPS. While both remain faithful to the semantics of contracts, their reliance on predefined patterns such as defeasible deontics in DDL, or fluents and actions in LPS, requires the modeler to extract the relevant concepts, and encapsulate them into the prescribed patterns. In DDL, this means distinguishing between the declaration of a rule, its applicability, and the grounding of atoms. In LPS, the gap is even wider, since the modeler must identify the relevant states described by the contract and formalise them explicitly as fluents. Thus, although these languages enhance semantic fidelity (see below), they achieve this at the cost of a lower structural correspondence to the text, in the sense defined by Bench-Capon [15].

Semantic fidelity. From the standpoint of semantic fidelity, logic-based languages generally provide the richest expressive power. To different extent, they generally support defeasible and normative reasoning, whether through dedicated constructs or suitable workarounds.

With regard to defeasibility, while providing for specific exceptions to rules is quite straightforward in every language, the main difference lies in how conflicts between rules are managed. Except for DDL, conflict handling relies on *negation-as-failure* and must be manually encoded within the knowledge base. This means that resolving contradictions or introducing priority between competing rules requires explicit modifications of the program logic, typically through additional conditions or control structures. As a result, conflict management is ad hoc and structurally embedded in the rule definitions.

DDL, by contrast, offers a native and systematic approach to defeasibility. Its underlying logic allows conflicts to be handled through explicit *superiority relations* between rules. This makes it possible to introduce new exceptions simply by defining a new rule and assigning it a priority, without altering existing ones.

Among the other approaches, Prolog remains the most flexible. Meta-rules can be defined to generalise conflict resolution patterns, although such mechanisms still operate within the limits of NAF reasoning (as shown in Section [9.2.1](#)).

In this context, imperative approaches are generally more limited. General mechanisms for solving conflict of rules are not available, these can only be handled case-by-case modifying the existing code. *Stipula*, however, introduces some features that can partially address conflicts, namely *non-determinism* detection and the `import` feature, which enable the ad hoc representation of very targeted patterns of defeasible reasoning.

A similar set of considerations applies to *normative positions*. DDL is the only language that provides dedicated constructs for obligations, permissions, and prohibitions, embedding their semantics directly into the system. Other logic-based languages can represent these positions as well, but only through more general mechanisms: Prolog and LE rely on meta-predicates, whereas LPS uses fluents to capture changes in the contractual state.

The key distinction lies in the ability to reason about such positions in a general and systematic way—for instance, to determine what should happen while an obli-

gation is active or when a violation occurs. DDL incorporates deontic reasoning into its semantics, allowing the automatic computation of permissions, obligations, violations, and contrary-to-duty clauses. In Prolog, similar reasoning can be implemented relatively easily through user-defined meta-predicates and general rules for detecting violations. In contrast, LE's restrictions on meta-predicate syntax make this process more cumbersome and verbose, as each case requires explicit, ad hoc encoding.

Imperative approaches follow a different logic. In general, normative positions can be represented operationally, by regulating parties behavior through the contract. (e.g, allowing or preventing the execution of specific functions). However, such an approach is only effective for actions occurring within the contract's own environment. Alternatively, these positions can be represented artificially as data structures without any specific semantic value. In this, *Stipula* represents an exception by providing a dedicated construct for obligation, namely *events* allowing the automatic detection of fulfilment or breach.

With regard to *temporal expressiveness*, the picture is more fragmented. Logic-based languages typically represent time in the same way as any other element of the knowledge base, i.e. through predicates and arguments. This permits the encoding of deadlines and durations, but does not provide a mechanism to track the passage of time. The exception is LPS, which features explicit support for fluents and reactive rules, and thus enables continuous monitoring and event-driven behaviours. It is therefore the only logic-based language to offer full support for temporal reasoning.

Imperative approaches take a different path. In Solidity, time is represented thanks to the direct connection with the EVM. However, the EVM does not include an internal scheduler: the contract cannot act spontaneously when a deadline expires. As a result, the verification of temporal conditions depends on external calls or oracles, which must invoke the relevant functions at the appropriate time.

Stipula, by contrast, integrates temporal constructs directly into its semantics. Events can be defined with deadlines that are automatically checked by the runtime environment: if an obligation is not fulfilled by the specified date, the contract transitions to the `@Breach` state or triggers the corresponding remedies. This eliminates the need for external oracles and allows the contract to enforce time-

bound provisions natively. As such, *Stipula* provides full support for temporal reasoning, both in terms of representation and in terms of automatic monitoring.

9.4.2 Accessibility.

The evaluation of accessibility reveals a clear gradient across the languages. Logical English is the only Tier 1 system, as its syntax closely resembles natural language and can be read directly by legal experts without IT expertise. At the opposite end, the more a language prioritises operational expressiveness, the less accessible it becomes in terms of readability. LPS, Solidity, and *Stipula* are the most operationally oriented languages in this study, but they also fall behind in accessibility, as their syntax requires specialised knowledge and relies heavily on computational constructs.

This highlights a broader trend: representing operational logic often requires introducing primitives that shift from strictly legal notions to more technical ones—such as contract states, actions, or party interactions. Within this landscape, *Stipula* occupies an intermediate position. Although it belongs to the same tier as Prolog and DDL (Tier 2), its accessibility profile differs slightly. It is less accessible than Prolog, whose syntax is closer to Tier 1 languages, yet more approachable than Solidity or LPS thanks to the inclusion of domain-specific constructs that make the code more intuitive for legal experts. Features such as the **agreement** clause, explicit **states**, and the built-in state machine facilitate the direct expression of contractual elements, reducing the need for low-level or purely technical encodings.

9.4.3 Operational Expressiveness.

As expected, operational expressiveness is higher in languages that adopt an imperative paradigm, showing an inverse relation with respect to legal fidelity. While logic-based languages excel in representing abstract legal concepts, imperative languages are far more effective in capturing the execution of concrete contractual actions and their consequences.

Resource handling is a first area where this contrast is evident. Logic-based languages do not provide any dedicated construct for resources. Currency or goods

are simply represented declaratively as arguments within predicates. This symbolic treatment suffices for reasoning, but it does not ensure that contractual resources are faithfully managed during contract execution.

By contrast, imperative smart-contract languages embed resource management directly into their execution model. Solidity, for example, does not include a primitive for resources in its semantics, but relies on widely adopted standards such as ERC-20 and ERC-721. These standards allow developers to create and manage fungible (currency-like) and non-fungible (unique) resources through a consistent interface.

Stipula goes even further by introducing a dedicated `asset` type for representing linear resources. Unlike in Solidity, where the preservation of uniqueness depends on compliance with external standards, in Stipula the semantics of the language itself guarantees that assets cannot be duplicated or destroyed improperly. This design enables formal analyses, such as liquidity checks, which verify that all resources are correctly managed throughout the contract’s lifecycle.

As to *automatability*, logic-based languages are generally not designed with practical automation in mind. Their default inference mechanism is backward reasoning: the program evaluates whether certain conclusions hold in light of a query and the current knowledge base. This means that nothing happens “automatically” in response to an external occurrence. When the knowledge base is updated, the consequences of that change are only derived once a query is posed. A small exception is Prolog, where the predicates `assert` and `retract` can be used to simulate dynamic updates of the knowledge base. LPS, by contrast, is explicitly designed to combine backward and forward reasoning. Through its semantics of causal laws and reactive rules, LPS allows one to define automatic behaviours that the contract should perform in response to external events, for example, generating an obligation when a triggering event occurs, or terminating a fluent when its conditions are no longer satisfied. This makes LPS the only logic-based language in the comparison to offer partial support for automatability.

Imperative languages take a very different approach. In Solidity and Stipula, function calls are not queries but actions and their execution directly changes the state of the contract. This imperative design makes it straightforward to encode self-enforcing mechanisms, where the invocation of a function automatically up-

dates obligations, transfers resources, or triggers remedies. Nevertheless, there is an important difference between the two languages. In Solidity, the Ethereum Virtual Machine cannot act spontaneously when it comes to time-dependent effects. Instead, an external party or an oracle must invoke a function such as `checkObligations` to update the state. Stipula, by contrast, relies on an internal notion of time defined in its semantics. It allows events with deadlines to be scheduled, and when these expire the system automatically executes the relevant operations and transitions the contract to the appropriate state (e.g. `@Breach`) without requiring external intervention. In sum, while imperative languages make automation natural and effective, Solidity remains constrained by the architecture of the EVM, whereas Stipula achieves full native support.

9.5 Trade-offs and Conclusions.

This comparative analysis highlights a fundamental trade-off between legal fidelity and operational expressiveness. Logic-based languages excel in modelling the normative dimension of contracts, but lack the means to automate their execution. Imperative languages, by contrast, capture operational aspects with precision but at the cost of semantic richness.

Within this landscape, Stipula and LPS stand out as partial exceptions. LPS, on the one hand, integrates reactivity and forward reasoning into a logic-based framework, thereby extending its automation capabilities beyond those of traditional declarative systems. Stipula, on the other hand, adopts an imperative and object-oriented paradigm that provides full support for operational expressiveness. At the same time, it introduces specific primitives and domain-oriented mechanisms that make it possible to capture certain legal reasoning patterns more faithfully than other imperative languages, thereby raising its overall level of legal fidelity.

No single language emerges as optimal in all respects. Rather, the comparison clarifies the relative strengths and weaknesses of each paradigm, guiding their appropriate selection for different use cases. In perspective, a combined use of both paradigms—or hybrid approaches that borrow features from each—appears a promising direction to reconcile legal fidelity with operational expressiveness.

Language	Legal Fidelity				Accessibility		Operational Express.	
	Isom.	Defeasibility		Norm. positions	Temporal expr.			
		Exc	Confl		Rep	Track		
Prolog	H	✓	✓	✓	✓	✗	●	●
LE	H	✓	✓	✓	✓	✗	●	⊙
DDL	H	✓✓	✓✓	✓✓	✓	✗	●	⊙
LPS	M	✓	✓	✓	✓✓	✓✓	●	●
Solidity	L	✓	✗	✓	✓✓	✗	●	●
Stipula	M	✓	Import Non-determ	Perm/Pro Obl✓✓	✓✓	✓✓	●	●

Table 9.8: Overall evaluation of the languages across all criteria.

Conclusions

Chapter 10

Final Reflections and Future Directions

10.1 Summary of Key Findings

This thesis has explored the conceptual foundations, technological frameworks, and methodological challenges of representing legal agreements in computable form. Its central objective was to investigate how contractual terms can be expressed through programming languages in a way that preserves their legal meaning while enabling automated execution and verification. The research combined theoretical analysis with practical design, resulting in a dual contribution: a conceptual framework for comparing legal modelling paradigms and a concrete implementation through the *Stipula* language.

The first part of dissertation has established a clear conceptual and terminological framework. By retracing the historical evolution of digital contracting, from early Electronic Data Interchange systems to blockchain-based smart contracts, we delimited the scope of the term *computable contract* and described its technical functions. The review of legal and institutional developments in Europe and beyond has shown that, while legal systems can generally accommodate computable contracts within existing doctrines, significant interpretive and practical challenges remain, particularly concerning consent, amendment, and termination under immutable infrastructures.

The second part directly addressed the first research sub-question RQ1 presented in Section 1.2. It showcased the design and implementation of *Stipula* domain-specific language, which provides a computable, legally oriented environment for drafting, executing, and analysing computable contracts. In addition to presenting its formal syntax, semantics, and runtime environment, this work introduces three major extensions to the language. The first concerns the inclusion of *higher-order clauses*, which allow the parties to modify contract terms dynamically, either by adding new provisions or by overriding existing ones. The second involves the identification of patterns of *legally incompatible clauses* and the implementation of tools for their automatic detection. Finally, a *contract integration* mechanism has been developed, enabling the importation of external legal norms and the management of potential normative conflicts. Together, these features demonstrate that complex legal constructs can be effectively represented and operationalised within a computational framework without compromising their legal nature.

The third part of the thesis addressed the second sub-question RQ2 and presented a comparative analysis of logic-based and imperative (object-oriented) approaches. An evaluation framework was developed around three key dimensions—*legal fidelity*, *accessibility*, and *operational expressiveness*—and applied to a common case study of particular relevance for current and future applications of computable contracts: a Data Use Agreement regulated under the EU Data Act. Within this framework, six languages—Prolog, Logical English, Defeasible Deontic Logic, LPS, Solidity, and *Stipula*—were systematically compared.

The analysis revealed that logic-based languages generally achieve higher *legal fidelity*, providing more accurate representations of normative content. Imperative languages, by contrast, excel in modelling the operational aspects of contracts, though they tend to show limitations in terms of accessibility and legal expressiveness. *Stipula*, in particular, emerges as a promising middle ground, combining the formal rigour typical of declarative systems with the executable semantics of imperative programming. In general, however, neither approach can be considered self-sufficient; a comprehensive framework for computable contracts should therefore aim to bridge the two paradigms rather than to choose between them.

Taken together, these findings indicate that while a domain-specific language

can significantly enhance the alignment between legal semantics and computational execution (RQ1), no single programming paradigm fully satisfies all the requirements of computable contracting (RQ2), thus supporting the need for hybrid or layered approaches.

10.2 Limitations of the Present Work

While this thesis has contributed to advancing the theoretical and practical understanding of computable contracts, several limitations should be acknowledged to contextualize its findings and to guide future developments.

First, the comparative survey presented in Part III relied primarily on the author’s individual assessment of the selected languages. Although the evaluation criteria—*legal fidelity*, *accessibility*, and *operational expressiveness*—were explicitly defined, the scoring and qualitative judgments naturally reflect a single researcher’s interpretation. This limitation is particularly significant with respect to the *accessibility* dimension, which concerns the ease with which different users, including legal professionals and technologists, can understand and use the examined formalisms. A more robust and empirically grounded evaluation would require the involvement of multiple assessors, ideally drawn from diverse disciplinary backgrounds. Such a multi-participant study could validate the current results and provide a more reliable, statistically supported basis for comparing usability and readability across paradigms.

Second, the comparison focused mainly on the representational dimension of computable contracts—that is, how contractual rules and concepts are expressed in code. Although the thesis discusses the semantics of each paradigm and occasionally references the outcomes of specific queries or function calls, it does not include systematic *execution traces* or runtime analyses. A more comprehensive study would involve implementing equivalent contractual scenarios across multiple platforms and observing their actual execution, including performance, state evolution, and error handling. This would strengthen the conclusions regarding operational expressiveness and verify whether the formal semantics align with real-world computational behaviour.

A further limitation concerns the scope of the comparative evaluation presented

in Part III. While the design and development of *Stipula* in Part II were illustrated through multiple examples and feature-specific use cases, the systematic comparative analysis of programming paradigms was conducted on a single benchmark scenario, namely a Data Use Agreement under the EU Data Act. Although this case was selected for its contemporary relevance and structural richness, relying on one contractual model limits the generalisability of the comparative findings. Different categories of contracts—such as consumer agreements, employment contracts, or financial instruments—may present distinct modelling challenges and could affect the relative assessment of legal fidelity, accessibility, or operational expressiveness. Future research should therefore replicate the comparative framework across a broader spectrum of contractual types to test the robustness and transferability of the conclusions.

Finally, while the thesis examines *logic-based*, *imperative*, and *domain-specific* approaches in depth, it does not propose an integrative architecture capable of combining these paradigms into a single coherent framework. Each approach is treated as distinct and complementary, yet no hybrid model is provided to bridge declarative reasoning and procedural execution. This represents both a limitation and an opportunity: future research could focus on designing multi-layered frameworks that interconnect normative reasoning, executable procedures, and interoperability mechanisms, thereby advancing the development of genuinely integrated computable contract systems.

10.3 Concluding Remarks

Computable contracts lie at the intersection of law, computer science, logic, and linguistics. Their interdisciplinary nature makes them both promising and challenging: they require not only technical precision, but also a deep understanding of legal reasoning and linguistic nuance. This thesis has argued that the central challenge in this field is the reconciliation of *legal semantics* with *computational logic*—that is, ensuring that what is legally expressed can also be computationally executed without distortion of meaning. In this respect, the work contributes both a conceptual vocabulary and a methodological framework for designing and analysing computable representations of legal agreements.

Despite the confusion that has historically surrounded the terminology of “computable” contracts, a shared understanding of their potential is gradually emerging. The recent intervention of the European legislator confirms this trend: by recognising and regulating a specific application of computable contract technology, the EU has given institutional legitimacy to an idea that had long remained largely theoretical. This marks an important step forward, as it signals a shift from speculative enthusiasm to regulatory awareness—an evolution that will inevitably stimulate new research and technological innovation.

Among the principles introduced by the European framework, the requirement of *consistency between the code and the legal agreement* stands out as particularly significant. This thesis has shown that such consistency, or *faithfulness*, can only be achieved through a design process that begins with the legal analysis of each contractual construct, as exemplified in the development of *Stipula*. By embedding legal meaning within computational syntax and semantics, it becomes possible to produce code that is not merely executable, but legally intelligible.

Yet, the thesis also demonstrates that no single paradigm is sufficient to capture the full complexity of computable contracting. Logical frameworks excel at representing normative reasoning, while imperative models offer the operational mechanisms needed for execution. A comprehensive system will therefore require the integration of both, each contributing where the other reaches its limits. Future research should thus aim to bridge these paradigms, exploring hybrid or layered architectures capable of reconciling legal expressiveness with computational efficiency.

Ultimately, the path toward truly computable contracts is not a purely technical one—it is a process of translation between disciplines, where code becomes a new language for law. In this dialogue, the task is not merely to automate legal processes, but to design systems that respect and preserve the normative structures upon which law itself is built.

Appendix A

Full Code Implementations

A.1 Logical English

```
1 the target language is: prolog.
2
3 the templates are:
4 *a person* concludes an agreement with *an other person* for *a dataset*.
5 *a person* has the obligation that *an eventuality*.
6 *a person* is the holder.
7 *a person* is the user.
8 *a dataset* is covered by the contract.
9 *a person* makes the data available to *an other person* within *a date*.
10 *a person* requests the data on *a date*.
11 *a data* is operational.
12 *a data* is usage.
13 *a data* is diagnostic.
14 *a data* is metadata.
15 *a person* has the permission that *an eventuality*.
16 *a person* uses the data of *a person* for *a purpose*.
17 *a person* uses the data given by *an other person* for *a purpose*.
18 *a purpose* is a permitted purpose.
19 *a person* shares the data given by *an other person* with *a thirdParty*.
20 *a party* is a third party.
21 *a third party* is a gatekeeper.
22 *a purpose* is an unlawful purpose.
23 *a person* compensates *an other person* with *an amount* eur within *a date*.
24 *a number* mb of data is calculated on *a date*.
25 *a number* mb of data are used.
26 *a person* is in breach of the agreement with *an other person*.
27 *a person* compensates *an other person* with *an amount* eur on *a date*.
28 *an amount* is the cost.
29 *a person* makes the data available to *an other person* on *a date*.
30 *a party* has the remedy to terminate the contract.
31 *a party* has the remedy to claim damages.
32 it terminates that *an eventuality*.
33 *a party* exercises the remedy to terminate the contract.
34 *a party* uses the remedy to terminate the contract.
35 *a party* uses the remedy to claim damages.
36 *a party* exercises the remedy to claim damages.
37 the product contract terminates.
38 *a person* authorises *an other person* to share with *a third party*.
39
```

```

40 the knowledge base data_use_agreement includes:
41
42 % Art.1 - Agreement
43 a holder concludes an agreement with a user for a dataset
44     if the holder is the holder
45     and the user is the user
46     and the dataset is covered by the contract.
47
48 %%% Facts
49 carMaker is the holder.
50 carRental is the user.
51
52 % Art.2 - Scope
53 a dataset is covered by the contract
54     if a oData is operational
55     and a uData is usage
56     and a dData is diagnostic
57     and a mData is metadata
58     and the dataset is [the oData, the uData, the dData, the mData].
59
60 %%% Facts
61 op is operational.
62 us is usage.
63 diagn is diagnostic.
64 meta is metadata.
65
66 % Art.3 - Data Access Right
67 a holder has the obligation that
68     the holder makes the data available to a user within a D
69     if the holder concludes an agreement with the user for a dataset
70     and the user requests the data on a D1
71     and the D is 5 days after the D1.
72
73
74 % Art. 4 - Data Holder Permitted Uses
75 a holder has the permission that
76     the holder uses the data of a user for a purpose
77     if the holder concludes an agreement with the user for a dataset
78     and the purpose is a permitted purpose.
79
80 %%% Facts
81 contractual obligations is a permitted purpose.
82 monitoring is a permitted purpose.
83 development is a permitted purpose.
84 commercial use is a permitted purpose.
85
86
87 % Art. 5 "User Permitted Uses"
88 a user has the permission that
89     the user uses the data given by a holder for a purpose
90     if the holder concludes an agreement with the user for a dataset
91     and it is not the case that
92         the purpose is an unlawful purpose.
93
94 a user has the permission that
95     the user shares the data given by a holder with a thirdParty
96     if the holder concludes an agreement with the user for a dataset
97     and the thirdParty is a third party
98     and it is not the case that
99         the thirdParty is a gatekeeper.
100
101 a user has the permission that
102     the user shares the data given by a holder with a thirdParty
103     if the holder concludes an agreement with the user for a dataset
104     and the thirdParty is a third party
105     and the thirdParty is a gatekeeper
106     and the holder authorises the user to share with the gatekeeper.
107

```

```

108
109 % Art. 6 "Compensation for Commercial Use"
110 a holder has the obligation that
111     the holder compensates a user with an amount eur within a D
112     if the holder concludes an agreement with the user for a dataset
113     and an number mb of data is calculated on a D1
114     and the D is 30 days after the D1
115     and the amount is the number*0.1.
116
117
118 % Art. 7 "Breach and remedies"
119 a holder is in breach of the agreement with a user
120     if the holder concludes an agreement with the user for a dataset
121     and the holder has the obligation that
122         the holder makes the data available to the user within a D
123     and it is not the case that
124         the holder makes the data available to the user within the D.
125
126 %% Breach of Data Access obligation
127 a holder makes the data available to a user within a D
128     if the holder makes the data available to the user on a D1
129     and the D1 is before the D.
130
131 %% Breach of Compensation obligation
132 a holder is in breach of the agreement with a user
133     if the holder concludes an agreement with the user for a dataset
134     and the holder has the obligation that
135         the holder compensates the user with an amount eur within a D
136     and it is not the case that
137         the holder compensates the user with the amount eur within the D.
138
139 a holder compensates a user with an amount eur within a D
140     if the holder compensates the user with the amount eur on a D1
141     and the D1 is before the D.
142
143 %% Breach of user sharing with gatekeeper
144 a user is in breach of the agreement with a holder
145     if the holder concludes an agreement with the user for a dataset
146     and the user shares the data given by the holder with a thirdParty
147     and it is not the case that
148         the user has the permission that
149             the user shares the data given by the holder with the thirdParty.
150
151 %% Breach of holder using data for non-permitted purposes
152 a holder is in breach of the agreement with a user
153     if the holder concludes an agreement with the user for a dataset
154     and the holder uses the data of the user for a purpose
155     and it is not the case that
156         the holder has the permission that
157             the holder uses the data of the user for the purpose.
158
159 %% Breach of user using data for unlawful purposes
160 a user is in breach of the agreement with a holder
161     if the holder concludes an agreement with the user for a dataset
162     and the user uses the data given by the holder for a purpose
163     and it is not the case that
164         the user has the permission that
165             the user uses the data given by the holder for the purpose.
166
167 %% Remedies
168 a party has the remedy to terminate the contract
169     if an other party is in breach of the agreement with the party.
170
171 a party has the remedy to claim damages
172     if an other party is in breach of the agreement with the party.
173
174
175 % Art. 8 "Termination"

```

```

176 it terminates that
177     a holder concludes an agreement with a user for a dataset
178     if a party exercises the remedy to terminate the contract.
179
180 a party exercises the remedy to terminate the contract
181     if the party uses the remedy to terminate the contract
182     and the party has the remedy to terminate the contract.
183
184 it terminates that
185     a holder concludes an agreement with a user for a dataset
186     if a party exercises the remedy to terminate the contract.
187     if the product contract terminates.

```

Listing A.1: Logical English Formalisation of DUA

A.2 Defeasible Deontic Logic on ASP

```

1 #include "../Deontic/defeasible-ap.asp".
2 #include "../Deontic/deontic-comp.asp".
3
4 atom(user_share).
5 atom(make_data_available(X)) :- request_data(Y), X = Y + 5.
6 atom(compensate(X,T)) :- rate(R), generated_data(D, T1), X = D / R, T = T1 + 30.
7
8 % Art.1 - Agreement
9 constitutiveRule(cr1, agreement).
10 applicable(cr1, agreement) :-
11     party(holder),
12     party(user),
13     covered_data(data).
14
15 %%% Facts
16 party(holder).
17 party(user).
18 covered_data(data).
19
20
21 % Art.2 - Scope
22 constitutiveRule(cr2, covered_data(data)).
23 applicable(cr2, covered_data(data)) :-
24     metadata(data).
25 applicable(cr2, covered_data(data)) :-
26     diagnostic(data).
27 applicable(cr2, covered_data(data)) :-
28     operational(data).
29 applicable(cr2, covered_data(data)) :-
30     usage(data).
31
32
33 % Art.3 - Data Access Right
34 prescriptiveRule(ob1, make_data_available(X)) :-
35     request_data(Y),
36     X = Y + 5.
37 applicable(ob1, make_data_available(X)) :-
38     request_data(Y),
39     X = Y + 5.
40
41
42 % Art. 5 "User Permitted Uses"
43 permissiveRule(per1, user_share).
44 applicable(per1, user_share) :-

```

```
45     defeasible(agreement).
46
47     %%% --- gatekeeper prohibition
48     prescriptiveRule(pro1, non(user_share)).
49     applicable(pro1, non(user_share)) :-
50         defeasible(agreement),
51         gatekeeper.
52
53     %%% --- gatekeeper authorisation permission
54     permissiveRule(per2, user_share).
55     applicable(per2, user_share) :-
56         defeasible(agreement),
57         defeasible(authorisation).
58
59     %%% --- User Sharing priorities
60     superior(pro1,per1).
61     superior(per2,pro1).
62
63
64     % Art. 6 "Compensation for Commercial Use"
65     prescriptiveRule(ob2, compensate(X,T)) :-
66         rate(R),
67         generated_data(D, T1),
68         X = D / R,
69         T = T1 + 30.
70     applicable(ob2, compensate(X,T)) :-
71         rate(R),
72         generated_data(D, T1),
73         X = D / R,
74         T = T1 + 30.
75
76     %%% Facts
77     rate(10).
78
79     % Art. 7 "Breach and remedies"
80     %%% --- Data Access Breach
81     constitutiveRule(br1, non(make_data_available(X))) :-
82         request_data(Y),
83         X = Y + 5.
84     applicable(br1, non(make_data_available(X))) :-
85         request_data(Y),
86         X = Y + 5,
87         five_days_expired,
88         not make_data_available(X).
89
90     %%% --- Compensation Breach
91     constitutiveRule(br2, non(compensate(X,T))) :-
92         rate(R),
93         generated_data(D, T1),
94         X = D / R,
95         T = T1 + 30.
96     applicable(br2, non(compensate(X,T))) :-
97         rate(R),
98         generated_data(D, T1),
99         X = D / R,
100         T = T1 + 30,
101         expired(T),
102         not occurred(compensate(X,T)).
103
104     % - REMEDIES
105     permissiveRule(per4, remedy_termination).
106     applicable(per4, remedy_termination) :-
107         violation(_,_,1).
108
109     permissiveRule(per5, remedy_claim_damages).
110     applicable(per5, remedy_claim_damages) :-
111         violation(_,_,1).
112
```

```

113
114 % Art. 8 "Termination"
115 constitutiveRule(ter, termination).
116 applicable(ter, termination) :-
117     remedy_termination.
118 applicable(ter, termination) :-
119     product_contract_terminated.

```

Listing A.2: DDL Formalisation of DUA

A.3 Logic Production System

```

1  events agreement/3, request_data/4, asks_to_share/2, holder_wants_to_use/2,
   user_wants_to_use/2, month_tick/1, tick/1, compensates/4, makes_available/3,
   uses_termination_remedy/1, uses_damages_remedy/2.
2
3  actions obligation_data_access/3, happens/3, calculate_mb/2, breach/1,
   obligation_fulfilled/3, terminates_contract/3, claim_damages/2.
4
5  fluents active_contract/3, obligation_to_make_available/3,
   user_permission_to_use_data/1, permission_to_share_data/1,
   holder_permission_to_use_data/1, obligation_to_compensate/4, breach/1.
6
7
8  % Art.1 - Agreement
9  agreement(Holder, User, Data) initiates active_contract(Holder, User, Data).
10
11
12 % Art.3 - Data Access Right
13 request_data(User, Holder, Data, T) initiates obligation_to_make_available(Holder,
   User, D)
14     if access_deadline(Deadline1), D is T + Deadline1.
15
16 if obligation_to_make_available(Holder, User, DueDate) at T,
17     makes_available(Holder, User, Date) from T to T1,
18     Date <= DueDate
19 then obligation_fulfilled(Holder, User, make_available) from T1.
20
21 obligation_fulfilled(Holder, User, make_available) terminates
   obligation_to_make_available(Holder, User, D).
22
23 %%% --- Static Facts
24 access_deadline(5).
25
26
27 % Art. 4 - Data Holder Permitted Uses
28 agreement(Holder, User, _Data) initiates holder_permission_to_use_data(Holder).
29
30 if holder_permission_to_use_data(Holder) at T,
31     holder_wants_to_use(Holder, Purpose) from T to T1,
32     permitted_purpose(Purpose)
33 then happens(Holder, Purpose, use_data) from T1.
34
35 %%% --- Static Facts
36 permitted_purpose(contractual_obligations).
37 permitted_purpose(monitored).
38 permitted_purpose(development).
39 permitted_purpose(commercial_use).
40
41
42 % Art. 5 "User Permitted Uses"

```

```

43 agreement(Holder, User, Data) initiates user_permission_to_use_data(User).
44
45 if      user_permission_to_use_data(User) at T,
46      user_wants_to_use(User, Purpose) from T to T1,
47      not unlawful(Purpose)
48 then happens(User, Purpose, use_data) from T1.
49
50 agreement(Holder, User, Data) initiates permission_to_share_data(User).
51
52 if      permission_to_share_data(User) at T,
53      asks_to_share(User, ThirdParty) from T to T1,
54      not gatekeeper(ThirdParty)
55 then happens(User, ThirdParty, share_data) from T1.
56
57 if      permission_to_share_data(User) at T,
58      asks_to_share(User, ThirdParty) from T to T1,
59      gatekeeper(ThirdParty),
60      authorized(ThirdParty)
61 then happens(User, ThirdParty, share_data) from T1.
62
63
64 % Art. 6 "Compensation for Commercial Use"
65 if      month_tick(D), generated_data(Number) % - D is the date of the tick; N is
66      the size of data.
67 then calculate_mb(Number, D).
68
69 calculate_mb(Number, D) initiates obligation_to_compensate(Holder, User, Amount,
70      DueDate)
71 if      active_contract(Holder, User, _Data),
72      mb_rate(Rate),
73      Amount is Number * Rate,
74      DueDate is D + 30.
75
76 if      obligation_to_compensate(Holder, User, Amount, DueDate) at T,
77      compensates(Holder, User, Amount, Date) from T to T1,
78      Date <= DueDate
79 then obligation_fulfilled(Holder, User, compensation) from T1.
80
81 obligation_fulfilled(Holder, User, compensation) terminates
82      obligation_to_compensate(Holder, User, Amount, DueDate).
83
84 %% --- Static Facts
85 mb_rate(0.10).
86
87
88 % Art. 7 "Breach and remedies"
89 breach(Person) initiates breach(Person).
90
91
92 %% -- Obligations Breach
93 if      obligation_to_compensate(Holder, User, Amount, D1) at T,
94      tick(D) from T,
95      D > D1
96 then breach(Holder).
97
98 if      obligation_to_make_available(Holder, User, D1) at T,
99      tick(D) from T,
100      D > D1
101 then breach(Holder).
102
103 %% -- Permissions Breach
104 if      active_contract(Holder, User, Data) at T,
105      asks_to_share(User, ThirdParty) from T to T1,
106      gatekeeper(ThirdParty),
107      not authorized(ThirdParty)
108 then breach(User) from T1.
109
110 if      active_contract(Holder, User, Data) at T,
111      holder_wants_to_use(Holder, Purpose) from T to T1,

```

```

108     not permitted_purpose(Purpose)
109 then    breach(Holder) from T1.
110
111 if      active_contract(Holder, User, Data) at T,
112         user_wants_to_use(User, Purpose) from T to T1,
113         unlawful(Purpose)
114 then    breach(User) from T1.
115
116 % --- REMEDIES
117 if      breach(Holder) at T,
118         active_contract(Holder, User, Data) at T,
119         uses_termination_remedy(User) from T to T1
120 then    terminates_contract(Holder, User, Data) from T1.
121
122 if      breach(User) at T,
123         active_contract(Holder, User, Data) at T,
124         uses_termination_remedy(Holder) from T to T1
125 then    terminates_contract(Holder, User, Data) from T1.
126
127 if      breach(Holder) at T,
128         active_contract(Holder, User, Data) at T,
129         uses_damages_remedy(User, Holder) from T to T1
130 then    claim_damages(User, Holder) from T1.
131
132 if      breach(User) at T,
133         active_contract(Holder, User, Data) at T,
134         uses_damages_remedy(Holder, User) from T to T1
135 then    claim_damages(Holder, User) from T1.
136
137
138 % Art. 8 "Termination"
139 terminates_contract(Holder, User, Data) terminates active_contract(Holder, User,
140 Data).
141 terminates_contract(Holder, User, Data) terminates holder_permission_to_use_data(
142 Holder).
143 terminates_contract(Holder, User, Data) terminates permission_to_share_data(User).
144 terminates_contract(Holder, User, Data) terminates user_permission_to_use_data(
145 User).
146
147 %% -- Observable Events
148 observe agreement(carMaker, carRental, covered_data) at 2.
149 observe request_data(carRental, carMaker, covered_data, 1) at 3.
150 observe asks_to_share(carRental, mario) from 4 to 5.
151 observe user_wants_to_use(carRental, sell_to_gatekeeper) from 4 to 5.
152 observe month_tick(10) at 4.
153 observe compensates(carMaker, carRental, 5.0, 10) from 7 to 8.
154 observe tick(8) at 8.
155 observe makes_available(carMaker, carRental, 9) from 9 to 10.
156 observe uses_termination_remedy(carMaker) from 8 to 9.

```

Listing A.3: LPS Formalisation of DUA

A.4 Solidity

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.7;
3
4 interface ICarNFT {
5     function transferCar(address to, uint256 tokenId) external;
6     function getDataAmount(uint256 tokenId) external view returns (uint256);

```

```

7   function approve(address to, uint256 tokenId) external;
8 }
9
10 contract DataUseAgreement {
11
12     // PARTIES
13     address public carRental;    // User
14     address public carMaker;    // Data Holder
15
16     // PARAMETERS - Contract Terms
17     string public dataCovered;
18     uint256 public mbPrice;
19     uint256 public accessDeadline;
20     uint256 public endDate;
21
22     // NORMATIVE POSITIONS
23     enum NormPositionType { DataRequest, Compensation, Permitted_uses }
24     uint256 constant NO_DEADLINE = type(uint256).max;
25     enum NormPositionStatus { active, fulfilled, breached }
26
27     struct NormPosition {
28         uint256 id;                // unique incremental identifier
29         NormPositionType oType;    // type of normative position (data request,
30                                   // compensation, ect)
31         string content;            // description
32         address obligor;           // party bound to perform
33         uint256 deadline;          // timestamp of expiry
34         NormPositionStatus oStatus;
35         uint amountDue;            // optional amount to be paid
36         string data;              // optional data delivered
37     }
38
39     // Storage for Normative positions
40     NormPosition[] public norm_positions;
41     uint256 private nextNormPositionId = 1;
42     mapping(uint256 => uint256) private _idx;
43
44     event NormPositionCreated(uint256 id, NormPositionType oType, string content,
45                               address obligor, uint256 deadline);
46     event NormPositionFulfilled(uint256 id, address obligor, string content);
47
48     // STATES
49     enum State { Inactive, Active, Breach, Dispute, Ended }
50     State public currentState;
51
52     // Car NFT
53     ICarNFT public carNFT;
54     uint256 public carTokenId;
55
56     // CONSTRUCTOR
57     constructor(
58         address _carRental, address _carMaker,
59         address _carNFT, string memory _dataCovered,
60         uint256 _mbPrice, uint256 _accessDeadline,
61         uint256 _compensationDeadline, uint256 _endDate
62     ) {
63         carRental = _carRental;
64         carMaker = _carMaker;
65         carNFT = ICarNFT(_carNFT);
66         dataCovered = _dataCovered;
67         mbPrice = _mbPrice;
68         accessDeadline = _accessDeadline;
69         compensationDeadline = _compensationDeadline;
70         endDate = _endDate;
71         currentState = State.Inactive;
72     }
73
74     // MODIFIERS

```

```

73     modifier onlyCarMaker() {
74         require(msg.sender == carMaker, "Only car maker can call this");
75         -;
76     }
77
78     modifier inState(State _state) {
79         require(currentState == _state, "Invalid state");
80         -;
81     }
82
83     modifier onlyCarRental() {
84         require(msg.sender == carRental, "Only car rental can call this");
85         -;
86     }
87
88     // FUNCTIONS
89     // Internal helper function
90     function _createNormPosition(
91         NormPositionType oType,
92         address obligor,
93         string memory content,
94         uint256 deadline,
95         uint256 amount,
96         string memory data
97     ) internal {
98         uint256 id = nextNormPositionId++;
99         norm_positions.push(NormPosition({
100             id: id,
101             oType: oType,
102             content: content,
103             obligor: obligor,
104             deadline: deadline,
105             amountDue: amount,
106             data: data,
107             oStatus: NormPositionStatus.active
108         }));
109
110         _idx[id] = norm_positions.length; // store index+1 for O(1) lookup
111         emit NormPositionCreated(id, oType, content, obligor, deadline);
112     }
113
114
115     // "Operational setup" - Beginning of the contractual relationship
116     function start(uint256 _carTokenId) external onlyCarMaker inState(State.
117         Inactive) {
118         carTokenId = _carTokenId;
119         carNFT.transferCar(address(this), carTokenId);
120
121         _createNormPosition( // Art. 4 "Holder Permitted Uses"
122             NormPositionType.Permitted_uses,
123             carMaker,
124             "only use for art4 purposes",
125             NO_DEADLINE,
126             0,
127             ""
128         );
129
130         _createNormPosition( // Art. 5 "User Permitted Uses"
131             NormPositionType.Permitted_uses,
132             carRental,
133             "Do NOT share with gatekeepers",
134             NO_DEADLINE,
135             0,
136             ""
137         );
138
139         currentState = State.Active;
140     }

```

```

140 // Art.3 "Data Access Right"
141 function requestData(string calldata content) external onlyCarRental inState(
142     State.Active) {
143     _createNormPosition(
144         NormPositionType.DataRequest,
145         carMaker,
146         "data request ex. art 3",
147         block.timestamp + accessDeadline,
148         0,
149         content
150     );
151 }
152 function deliverData(uint256 norm_positionId, string calldata _data)
153     external
154     onlyCarMaker
155 {
156     uint256 i = _idx[norm_positionId];
157     require(i != 0, "NormPosition not found");
158     NormPosition storage ob = norm_positions[i - 1];
159
160     require(ob.oType == NormPositionType.DataRequest, "Not a data request
161         normative position");
162     require(ob.oStatus == NormPositionStatus.active, "Already fulfilled");
163
164     // -- Store the data in the request struct
165     ob.data = _data;
166     ob.oStatus = NormPositionStatus.fulfilled;
167     emit NormPositionFulfilled(ob.id, ob.obligor, ob.content);
168 }
169
170 // Art. 6 "Compensation for Commercial Use"
171 function calculateGeneratedData() external inState(State.Active) {
172     uint256 amountDue = carNFT.getDataAmount(carTokenId) * mbPrice; // pull
173     // from NFT
174
175     _createNormPosition(
176         NormPositionType.Compensation,
177         carMaker,
178         "Compensation ex art.6",
179         block.timestamp + compensationDeadline,
180         amountDue,
181         ""
182     );
183 }
184 function compensate(uint256 norm_positionId) external onlyCarMaker payable {
185     uint256 i = _idx[norm_positionId];
186     require(i != 0, "NormPosition not found");
187     NormPosition storage ob = norm_positions[i - 1];
188
189     require(ob.oType == NormPositionType.Compensation, "Not a compensation
190         normative position");
191     require(ob.oStatus == NormPositionStatus.active, "Already fulfilled");
192     require(msg.value == ob.amountDue, "Incorrect ETH sent");
193
194     // transfer ETH to beneficiary
195     (bool ok, ) = carRental.call{value: msg.value}("");
196     require(ok, "Transfer failed");
197
198     ob.oStatus = NormPositionStatus.fulfilled;
199     emit NormPositionFulfilled(ob.id, ob.obligor, ob.content);
200 }
201
202 // Art. 7 "Breach and Remedies"
203 // -- Breach

```

```

204     function checkObligations() external { // Breach of arts. 3 and 6
205         uint256 n = norm_positions.length;
206         for (uint256 i = 0; i < n; i++) {
207             NormPosition storage ob = norm_positions[i];
208             if (ob.oStatus == NormPositionStatus.active && block.timestamp > ob.
                deadline) {
209                 ob.oStatus = NormPositionStatus.breached;
210                 currentState = State.Breach;
211             }
212         }
213     }
214 }
215
216 function markBreach(uint256 norm_positionId, string memory reason) external {
217     // Breach of arts. 4 and 5
218     uint256 i = _idx[norm_positionId];
219     require(i != 0, "NormPosition not found");
220     NormPosition storage ob = norm_positions[i - 1];
221
222     require(ob.oType == NormPositionType.Permitted_uses, "Invalid normative
        position");
223
224     // Check caller matches the counterpart obligor
225     if (msg.sender == carRental) {
226         require(ob.obligor == carMaker, "Invalid obligor");
227     } else if (msg.sender == carMaker) {
228         require(ob.obligor == carRental, "Invalid obligor");
229     } else {
230         revert("Not a party");
231     }
232
233     ob.oStatus = NormPositionStatus.breached;
234     ob.data = reason;
235     currentState = State.Breach;
236 }
237
238 // -- Remedies
239 enum RemedyType { Termination, Damages }
240
241 function remedy(uint256 norm_positionId, RemedyType rType) external inState(
    State.Breach) {
242     uint256 i = _idx[norm_positionId];
243     require(i != 0, "NormPosition not found");
244     NormPosition storage ob = norm_positions[i - 1];
245
246     require(ob.oStatus == NormPositionStatus.breached, "NormPosition not
        breached");
247
248     if (msg.sender == carRental) {
249         require(ob.obligor == carMaker, "Invalid normative position for user");
250     } else if (msg.sender == carMaker) {
251         require(ob.obligor == carRental, "Invalid normative position for
            holder");
252     } else {
253         revert("Not a party");
254     }
255
256     if (rType == RemedyType.Termination) {
257         // Return NFT to CarMaker
258         carNFT.transferCar(carMaker, carTokenId);
259         currentState = State.Ended;
260     } else if (rType == RemedyType.Damages) {
261         currentState = State.Dispute;
262     }
263 }

```

Listing A.4: Solidity Formalisation of DUA

A.5 Stipula

```

1 stipula DataUseAgreement {
2   asset    car
3   field    covered_data, mb_price, data_req, access_deadline, data_volume_mb,
4           comp_deadline, end_date, breacher
5   init     Inactive
6
7   // Agreement Setup: Art. 1,
8   agreement(carMaker, carRental) {
9     carMaker, carRental : covered_data, mb_price, access_deadline,
10    comp_deadline, end_date
11  } ==> @Inactive
12
13  // "Operational Setup" - Starting the contract lifecycle
14  @Inactive carMaker: start_product()[p]{
15    p -o car;
16    end_date >> @Run{           // Art.8 "Termination"
17      car -o carMaker
18      "termination" -> carRental, carMaker
19    } ==> @End
20  } ==> @Run
21
22  // Art.3 "Data Access Right"
23  @Run carRental: requestData (x)[] {
24    x -> data_req;
25    now + access_deadline >> @Req{           // Obligation to make data available
26      "dataHolder" -> breacher
27    } ==> @Breach
28  } ==> @Req
29
30  @Req carMaker: deliverData(d, req)[] (req == data_req) {
31    d -> carRental
32    0 -> data_req;
33  } ==> @Run
34
35  // Art. 6 "Compensation for Commercial Use"
36  @Run ~: get_generated_data(x)[] {
37    x -> data_volume_mb;
38    comp_deadline >> @Comp{           // Obligation to compensate
39      "dataHolder" -> breacher
40    } ==> @Breach
41  } ==> @Comp
42
43  @Comp carMaker: compensate()[m] (m == data_volume_mb * mb_price) {
44    m -o carRental
45    null -> data_volume_mb;
46  } ==> @Run
47
48  // Art. 7 "Breach and remedies"
49
50
51
52
53

```

```
54 // -- Breach of Permitted Uses arts 4 and 5
55 @Run carMaker: breach(x)[] (x == 'gatekeeper_share') {
56   x -> carRental
57   "user" -> breacher;
58 } ==> @Breach
59
60 @Run carRental: breach(x)[] (x == 'forbidden_use') {
61   x -> carMaker
62   "dataHolder" -> breacher;
63 } ==> @Breach
64
65 // -- Remedies
66 @Breach carRental: terminate()[] (breacher == "dataHolder"){
67   "termination" -> carMaker
68   car -o carMaker;
69 } ==> @End
70
71 @Breach carMaker: terminate()[] (breacher == "user"){
72   "termination" -> carRental
73   car -o carMaker;
74 } ==> @End
75
76 @Breach carRental: claim_damages(x)[] (breacher == "dataHolder"){
77   "damages" -> carMaker;
78 } ==> @Dispute
79
80 @Breach carMaker: claim_damages(x)[] (breacher == "user"){
81   "damages" -> carMaker;
82 } ==> @Dispute
83
84 }
```

Listing A.5: Stipula Formalisation of DUA

Bibliography

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2nd edition, 2006.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Boston, MA, USA, 2006.
- [3] Jason G Allen. Wrapped and stacked: ‘smart contracts’ and the interaction of natural and formal language. *European Review of Contract Law*, 14(4):307–343, 2018.
- [4] Layman E Allen. Symbolic logic: A razor-edged tool for drafting and interpreting legal documents. *The Yale Law Journal*, 66(6):833–880, 1957.
- [5] Layman E Allen. Language, law and logic: plain legal drafting for the electronic age. *Computer science and law*, pages 75–100, 1980.
- [6] Layman E. Allen. Some examples of using legal relations language in the legal domain: Applied deontic logic. *Notre Dame Law Review*, 73(3):535–574, 1998.
- [7] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP and the minizinc challenge. *Theory Pract. Log. Program.*, 18(1):81–96, 2018.
- [8] Grigoris Antoniou and Antonis Bikakis. Dr-prolog: A system for defeasible reasoning with rules and ontologies on the semantic web. *IEEE transactions on knowledge and data engineering*, 19(2):233–245, 2006.

- [9] Joaquín Arias, Mar Moreno-Rebato, Jose A Rodriguez-García, and Sascha Ossowski. Automated legal reasoning with discretion to act using s(LAW). *Artificial Intelligence and Law*, 32(4):1141–1164, 2024.
- [10] Matthias Armgardt and Giovanni Sartor. Leibniz, gottfried wilhelm von: legal philosophy. In *Encyclopedia of the Philosophy of Law and Social Philosophy*, pages 2100–2104. Springer, 2023.
- [11] Kevin D Ashley. *Artificial intelligence and legal analytics: new tools for law practice in the digital age*. Cambridge University Press, 2017.
- [12] Tara Athan, Harold Boley, Guido Governatori, Monica Palmirani, Adrian Paschke, and Adam Wyner. Oasis legalruleml. In *proceedings of the fourteenth international conference on artificial intelligence and law*, pages 3–12, 2013.
- [13] Pablo Marcello Baquero, David Restrepo Amariles, Daniel Amyot, Amal Ahmed Anda, Mustafa Bayirli, Luigi Logrippo, Alberto Lopes, John Mylopoulos, Alireza Parvizimosaed, Aidin Rasti, et al. The compliance gap in data supply chains: contract specification languages and smart contracts as compliance technologies: P. baquero et al. *Artificial Intelligence and Law*, pages 1–50, 2025.
- [14] Omri Ben-Shahar and Oren Bar-Gill. Regulatory techniques in consumer protection: a critique of european consumer contract law. *Common Market Law Review*, 50(Special), 2013.
- [15] Trevor JM Bench-Capon and Frans P Coenen. Isomorphism and legal knowledge based systems. *Artificial Intelligence and Law*, 1:65–86, 1992.
- [16] Trevor JM Bench-Capon, Gwen O Robinson, Tom W Routen, and Marek J Sergot. Logic programming for large scale applications in law: A formalisation of supplementary benefit legislation. In *Proceedings of the 1st international conference on Artificial intelligence and law*, pages 190–198, 1987.

- [17] Francoois Bergeron and Louis Raymond. The advantages of electronic data interchange. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems*, 23(4):19–31, 1992.
- [18] B Douglas Bernheim and Michael D Whinston. Incomplete contracts and strategic ambiguity. *American Economic Review*, pages 902–932, 1998.
- [19] Guido Boella, Gabriella Pigozzi, and Leon van der Torre. Agm contraction and revision of rules. *J Log Lang Inf*, 25:273–297, 2016.
- [20] Chantal Bompreszi. Commento in materia di blockchain e smart contract alla luce del nuovo decreto semplificazioni. *Diritto, Mercato e Tecnologia*, 2019.
- [21] Chantal Bompreszi. *Implications of Blockchain-Based Smart Contracts on Contract Law*, volume 23 of *Luxemburger Juristische Studien - Luxembourg Legal Studies*. Nomos, Baden-Baden, 1 edition, 2021.
- [22] Veronica Bongiovanni et al. *Integrazione del contratto e clausole implicite*. Giuffrè, 2018.
- [23] Timothy Budd. *Introduction to object-oriented programming*. Pearson Education India, 2008.
- [24] Leslie Burkholder. The halting problem. *ACM SIGACT News*, 18(3):48–60, 1987.
- [25] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. ethereum.org/en/whitepaper/.
- [26] Michele Caianiello, Giuseppe Contissa, and Giulia Lasagni. Facilitating judicial cooperation in the eu: A computable approach to mutual recognition in criminal matters, 2025.
- [27] Gioia Caldarelli. Unilateral modification of long term contracts: American change of terms clauses and italian *Ius Variandi* from a ‘relational’ point of view. *European Review of Contract Law*, 17(1):37–53, 2021.

- [28] Giulio Caldarelli. Real-world blockchain applications under the lens of the oracle problem. a systematic literature review. In *2020 IEEE International Conference on Technology Management, Operations and Decisions (ICT-MOD)*, pages 1–6. IEEE, 2020.
- [29] Giulio Caldarelli. Understanding the blockchain oracle problem: A call for action. *Information*, 11(11), 2020.
- [30] Roberta Calegari, Giuseppe Contissa, Francesca Lagioia, Andrea Omicini, and Giovanni Sartor. Defeasible systems in legal reasoning: A comparative assessment. In *Legal Knowledge and Information Systems*, pages 169–174. IOS Press, 2019.
- [31] Federico Casolari, Mariarosaria Taddeo, Aina Turillazzi, and Luciano Floridi. How to improve smart contracts in the european union data act. *Digital Society*, 2(1):9, 2023.
- [32] Paul Catchlove. Smart contracts: a new era of contract use. *Available at SSRN 3090226*, 2017.
- [33] Luca Cervone, Monica Palmirani, and Fabio Vitali. The intelligible contract. In *53rd Hawaii International Conference on System Sciences, HICSS 2020*, pages 1–10, 2020.
- [34] Ilias Chalkidis and Dimitrios Kampas. Deep learning in law: early adaptation and legal word embeddings trained on large corpora. *Artificial Intelligence and Law*, 27(2):171–198, 2019.
- [35] Albert Choi and George Triantis. Strategic vagueness in contract design: The case of corporate acquisitions. *Yale LJ*, 119:848, 2009.
- [36] Christopher D Clack. Smart contract templates: legal semantics and code validation. *Journal of Digital Banking*, 2(4):338–352, 2018.
- [37] Christopher D Clack. Languages for Smart and Computable Contracts. In *Smart Legal Contracts: Computable Law in Theory and Practice*. Oxford University Press, 04 2022.

- [38] Christopher D Clack, Vikram A Bakshi, and Lee Braine. Smart contract templates: essential requirements and design options. *arXiv preprint arXiv:1612.04496*, 2016.
- [39] William F Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.
- [40] George M Cohen. Implied terms and interpretation in contract law. *Encyclopedia of law and economics*, 3:78–99, 2000.
- [41] Hugh Collins. Implied terms: The foundation in good faith and fair dealing. *Current Legal Problems*, 67(1):297–331, 2014.
- [42] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. 1996.
- [43] European Commission. Frequently asked questions about the data act, June 2024. Accessed August 2025.
- [44] UK Law Commission. Smart legal contracts: Advice to government. UK Government Web Archive, 2021. Law Com No. 401, UK Law Commission, November 2021.
- [45] Giuseppe Contissa, Alessia Fidelangeli, Galileo Sartor, et al. Multilingual logical english for double taxation conventions. In *Proceedings of the Joint Ontology Workshops (JOWO)-Episode X: The Tukker Zomer of Ontology, and satellite events*, pages 1–12, 2024.
- [46] Giuseppe Contissa, Giulia Lasagni, Michele Caianiello, and Giovanni Sartor. *Effective Protection of the Rights of the Accused in the EU Directives: A Computable Approach to Criminal Procedure Law*. Brill — Nijhoff, Leiden, The Netherlands, 2022.
- [47] Silvia Crafa, Cosimo Laneve, Giovanni Sartor, and Adele Veschetti. Pacta sunt servanda: legal contracts in stipula. *Science of Computer Programming*, 225:102911, 2023.

- [48] Silvia Crafa, Cosimo Laneve, and Adele Veschetti. The Stipula Project. On github: <https://github.com/stipula-language>.
- [49] Michael R Crum and Benjamin J Allen. Shipper edi, carrier reduction, and contracting strategies: Impacts on the motor carrier industry. *Transportation Journal*, pages 18–32, 1990.
- [50] Lawrence A Cunningham. Language, deals, and standards: The future of xml contracts. *Wash. UL Rev.*, 84:313, 2006.
- [51] Brian Curran. What are oracles? smart contracts, chainlink and “the oracle problem”. Blockonomi, September 19 2018. Accessed: 2025-08-01.
- [52] Chris Dannen et al. *Introducing Ethereum and solidity*, volume 1. Springer, 2017.
- [53] Aspasia Daskalopulu. Modelling legal contracts as processes. In *Proceedings 11th International Workshop on Database and Expert Systems Applications*, pages 1074–1079. IEEE, 2000.
- [54] Randall Davis and Jonathan King. *An overview of production systems*. Stanford University, 1975.
- [55] Riccardo de Caria. Blockchain and smart contracts: legal issues and regulatory responses between public and private economic law. *Italian LJ*, 6:363, 2020.
- [56] Gregory M Duhl. Conscious ambiguity: Slaying cerberus in the interpretation of contractual inconsistencies. *U. Pitt. L. Rev.*, 71:71, 2009.
- [57] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming, and n -person games. *Artificial Intelligence*, 77:321–57, 1995.
- [58] Phan Minh Dung and Giovanni Sartor. The modular logic of private international law. *Artificial Intelligence and Law*, 19(2):233, 2011.

- [59] Mateja Durovic and André Janssen. The formation of blockchain-based smart contracts in the light of contract law. *European Review of Private Law*, 26:753–771, 2019.
- [60] Mateja Durovic and Chris Willett. A legal framework for using smart contracts in consumer contracts: Machines as servants, not masters. *The Modern Law Review*, 86(6):1390–1421, 2023.
- [61] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2001.
- [62] European Commission Expert Group. Final report of the expert group on b2b data sharing and cloud computing contracts, April 2025. Accessed August 2025.
- [63] E Allan Farnsworth. “meaning” in the law of contracts. *The Yale Law Journal*, 76(5):939–965, 1967.
- [64] Stephen Fenech, Gordon J Pace, and Gerardo Schneider. Automatic conflict detection on contracts. In *International colloquium on theoretical aspects of computing*, pages 200–214. Springer, 2009.
- [65] Giusella Finocchiaro. Articolo 25. In Francesco Delfini and Giusella Finocchiaro, editors, *Identificazione elettronica e servizi fiduciari per le transazioni elettroniche nel mercato interno: commento al regolamento UE 910/2014*. G. Giappichelli Editore, 2017.
- [66] Giusella Finocchiaro and Chantal Bomprezzi. A legal analysis of the use of blockchain technology for the formation of smart legal contracts. *MediaLaws*, July, 2020.
- [67] Mark D Flood and Oliver R Goodenough. Contract as automaton: representing a simple financial agreement in computational form. *Artificial Intelligence and Law*, 30(3):391–416, 2022.
- [68] Marcel Fontaine and Filip De Ly. *Drafting international contracts*. BRILL, 2006.

- [69] Ethereum Foundation. Solidity documentation, version 0.8.30, 2025. Version 0.8.30, accessed 12 September 2025.
- [70] Pedro Franco. *Understanding Bitcoin: Cryptography, engineering and economics*. John Wiley and Sons, 2014.
- [71] Michael Genesereth. Dynamic logic programming. In *Prolog: The Next 50 Years*, pages 197–209. Springer, 2023.
- [72] Georgios K Giannikis and Aspasia Daskalopulu. Defeasible reasoning with e-contracts. In *2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 690–694. IEEE, 2006.
- [73] Michael Gisler, Katarina Stanoevska-Slabeva, and Markus Greunz. Legal aspects of electronic contracts. In *ISDO*. Citeseer, 2000.
- [74] Thomas F Gordon, Guido Governatori, and Antonino Rotolo. Rules and norms: Requirements for rule interchange languages in the legal domain. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 282–296. Springer, 2009.
- [75] Guido Governatori. Representing business contracts in ruleml. *International Journal of Cooperative Information Systems*, 14(02n03):181–216, 2005.
- [76] Guido Governatori. An asp implementation of defeasible deontic logic. *KI-Künstliche Intelligenz*, 38(1):79–88, 2024.
- [77] Guido Governatori, Trevor Bench-Capon, Bart Verheij, Michał Araszkiewicz, Enrico Francesconi, and Matthias Grabmair. Thirty years of artificial intelligence and law: the first decade. *Artificial Intelligence and Law*, 30(4):481–519, 2022.
- [78] Guido Governatori, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor, and Xiwei Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law*, 26:377–409, 2018.

- [79] Guido Governatori, Michael J Maher, Grigoris Antoniou, and David Billington. Argumentation semantics for defeasible logic. *Journal of Logic and Computation*, 14(5):675–702, 2004.
- [80] Guido Governatori and Zoran Milosevic. Dealing with contract violations: formalism and domain specific language. In *Ninth IEEE International EDOC Enterprise Computing Conference (EDOC’05)*, pages 46–57. IEEE, 2005.
- [81] Guido Governatori, Zoran Milosevic, and Shazia Sadiq. Compliance checking between business processes and business contracts. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC’06)*, pages 221–232. IEEE, 2006.
- [82] Guido Governatori, Francesco Olivieri, Antonino Rotolo, and Simone Scanapieco. Computing strong and weak permissions in defeasible logic. *Journal of Philosophical Logic*, 42(6):799–829, 2013.
- [83] Guido Governatori, Antonino Rotolo, and Giovanni Sartor. Logic and the law: Philosophical foundations, deontics, and defeasible reasoning. In *Handbook of Deontic Logic and Normative Systems, Volume 2*, pages 657–764. College Publications, 2022.
- [84] Guido Governatori, Antonino Rotolo, Giovanni Sartor, D Gabbay, J Horty, X Parent, R van der Meyden, and L van der Torre. Logic and the law: philosophical foundations, deontics, and defeasible reasoning. *Handbook of Deontic Logic and Normative Reasoning*, 2:655–760, 2021.
- [85] Ian Grigg. Digital trading. *Virtual Finance Report*, 1997.
- [86] Ian Grigg. The ricardian contract. In *Proceedings. First IEEE International Workshop on Electronic Contracting, 2004.*, pages 25–31. IEEE, 2004.
- [87] Gopal Gupta, Elmer Salazar, Sarat Chandra Varanasi, Kinjal Basu, Joaquín Arias, Farhad Shakerin, Richard Min, Fang Li, and Huaduo Wang. Automating common sense reasoning with asp and s (casp). In *Proceedings of 2nd Workshop on Goal-directed Execution of Answer Set Programs (GDE 2022)*, 2022.

- [88] James Hazard and Helena Haapio. Wise contracts: smart contracts that work for people and machines. In *Trends and communities of legal informatics. Proceedings of the 20th international legal informatics symposium IRIS*, pages 425–432, 2017.
- [89] Moritz Hennemann. *Data Act: An Introduction*. First edition, 2024.
- [90] John F. Horty. Deontic logic as founded on nonmonotonic logic. *Annals of Mathematics and Artificial Intelligence*, 9(1–2):69–91, March 1993.
- [91] Paola Iamiceli 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. *Giustizia Civile*, 2020(4):713–753, 2020.
- [92] Florian Idelberger. *The uncanny valley of computable contracts: analysis of computable contract formalisms with a focus towards controlled natural languages*. PhD thesis, European University Institute, 2022.
- [93] European Law Institute. Guiding principles and model rules on digital assistants for consumer contracts. Technical report, 2025.
- [94] Fulvio Sarzana di S Ippolito and Massimiliano Nicotra. *Diritto della blockchain, intelligenza artificiale e IoT*. Wolters Kluwer, 2018.
- [95] Nils Jansen and Reinhard Zimmermann. “A European Civil Code in all but name”: Discussing the nature and purposes of the draft common frame of reference. *The Cambridge Law Journal*, 69(1):98–112, 2010.
- [96] A.U. Janssen. Demystifying smart contracts. In I.V. Aronstein, C.J.H. Jansen, and B.A. Schuijling, editors, *Onderneming en digitalisering*, pages 15–29. Wolters Kluwer, 2019.
- [97] Andrew JI Jones and Marek Sergot. Deontic logic in the representation of law: Towards a methodology. *Artificial Intelligence and Law*, 1:45–64, 1992.

- [98] Adrian Kashivskyy. Imperative vs. declarative programming — pros and cons, October 2024. www.netguru.com/blog/imperative-vs-declarative. Accessed: 2025-08-25.
- [99] Alan Khoja, Martin Kölbl, Stefan Leue, and Rüdiger Wilhelm. Automated consistency analysis for legal contracts. In *International Symposium on Model Checking Software*, pages 1–23. Springer, 2022.
- [100] Robert Kowalski. Predicate logic as programming language. In *IFIP congress*, volume 74, pages 569–544, 1974.
- [101] Robert Kowalski. Logical english. *Proceedings of Logic and Practice of Programming (LPOP)*, 2020.
- [102] Robert Kowalski and Akber Datoo. Logical english meets legal english for swaps and derivatives. *Artificial Intelligence and Law*, 30(2):163–197, 2022.
- [103] Robert Kowalski, Jacinto Dávila, Galileo Sartor, and Miguel Calejo. Logical english for law and education. In *Prolog: The Next 50 Years*, pages 287–299. Springer, 2023.
- [104] Robert Kowalski and Fariba Sadri. Integrating logic programming and production systems in abductive logic programming agents. In *Web Reasoning and Rule Systems: Third International Conference, RR 2009, Chantilly, VA, USA, October 25-26, 2009, Proceedings 3*, pages 1–23. Springer, 2009.
- [105] Robert Kowalski and Fariba Sadri. An agent language with destructive assignment and model-theoretic semantics. In *Computational Logic in Multi-Agent Systems: 11th International Workshop, CLIMA XI, Lisbon, Portugal, August 16-17, 2010. Proceedings 11*, pages 200–218. Springer, 2010.
- [106] Robert Kowalski and Fariba Sadri. A logic-based framework for reactive systems. In *Rules on the Web: Research and Applications: 6th International Symposium, RuleML 2012, Montpellier, France, August 27-29, 2012. Proceedings 6*, pages 1–15. Springer, 2012.

- [107] Robert Kowalski, Fariba Sadri, Miguel Calejo, and Jacinto Dávila. Combining logic programming and imperative programming in lps. In *Prolog: The Next 50 Years*, pages 210–223. Springer, 2023.
- [108] Robert A. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.
- [109] P Radha Krishna, Kamalakar Karlapalem, and Ajay R Dani. From contracts to e-contracts: Modeling and enactment. *Information Technology and Management*, 6:363–387, 2005.
- [110] Ho-Pun Lam and Guido Governatori. The making of spindle. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 315–322. Springer, 2009.
- [111] Kwok-Yan Lam, Victor CW Cheng, and Zee Kin Yeong. Applying large language models for enhancing contract drafting. In *Proceedings of the Third International Workshop on Artificial Intelligence and Intelligent Assistance for Legal Professionals in the Digital Workspace (LegalAIIA 2023)*, 2023.
- [112] Cosimo Laneve. Reachability analysis in micro-stipula. In *PPDP '24: Proc. of the 26th Int. Symposium on Principles and Practice of Declarative Programming*, New York, NY, USA, 2024. Association for Computing Machinery.
- [113] Cosimo Laneve, Alessandro Parenti, and Giovanni Sartor. Legal contracts amending with *Stipula*. In *25th International Conference on Coordination Languages and Models (COORDINATION'23)*, volume 13908 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2023.
- [114] John W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition, 1993.
- [115] Olivera Marjanovic and Zoran Milosevic. Towards formal modeling of e-contracts. In *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, pages 59–68. IEEE, 2001.

- [116] Claudia Masciopinto. *L'impatto degli smart contracts sulla disciplina del contratto. Una nuova frontiera dell'autonomia negoziale?* Phd thesis, Luiss Guido Carli, Rome, Italy, 2023.
- [117] Marisaria Maugeri. Smart contracts e disciplina dei contratti. *Osservatorio del diritto civile e commerciale*, 9(2):1375–408, 2020.
- [118] Marisaria Maugeri. Smart contracts, consumer protection, and competing european narratives of private law. *German Law Journal*, 23(6):900–909, 2022.
- [119] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: a programming language for the law. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, 2021.
- [120] Eliza Mik. Smart contracts: terminology, technical limitations and real world complexity. *Law, innovation and technology*, 9(2):269–300, 2017.
- [121] Eliza Mik. Deconstructing smart contracts. *Forthcoming, Emerging Issues at the Intersection of Commercial Law and Technology*, ed by Nancy Kim and Stacy-Ann Elvy, Tilburg Law School Research Paper, 2022.
- [122] Eliza Mik. Smart contracts and the “oracle problem” in the context of insurtech. *European Insurance Contract Law: The Promises and Perils of Digitalization (Springer, Forthcoming)*, 2023.
- [123] Zoran Milosevic and Andy Bond. Electronic commerce on the internet: what is still missing? In *Proceedings of 5th Conference of the Internet Society*, pages 245–254, 1995.
- [124] Zoran Milosevic and R Geoff Dromey. On expressing and monitoring behaviour in contracts. In *Proceedings. Sixth International Enterprise Distributed Object Computing*, pages 3–14. IEEE, 2002.
- [125] Ritesh Modi. *Solidity programming essentials : a guide to build smart contracts and using the widely used Solidity language*. Second edition, 2022.

- [126] Erik T Mueller. *Commonsense reasoning: an event calculus based approach*. Morgan Kaufmann, 2014.
- [127] Satoshi Nakamoto. Bitcoin whitepaper, 2008. bitcoin.org/bitcoin.
- [128] Pablo E. Navarro and Jorge L. Rodríguez. *Deontic Logic and Legal Systems*. Cambridge University Press, 2014.
- [129] Stephen Neal, James Cole, Peter F Linington, Zoran Milosevic, Simon Gibson, and Sachin Kulkarni. Identifying requirements for business contract language: a monitoring perspective. In *Seventh IEEE International Enterprise Distributed Object Computing Conference, 2003. Proceedings.*, pages 50–61. IEEE, 2003.
- [130] Donald Nute. d-prolog: an implementation of defeasible logic in prolog. *Non-monotonic extensions of logic programming: theory, implementation, and applications*, pages 161–182, 1996.
- [131] Luca Olivieri and Luca Pasetto. Towards compliance of smart contracts with the european union data act. In *CEUR Workshop Proceedings*, volume 3629. CEUR-WS, 2023.
- [132] Luca Olivieri, Luca Pasetto, Luca Negrini, Pietro Ferrara, et al. European union data act and blockchain technology: Challenges and new directions. In *CEUR Workshop Proceedings*, volume 3791, 2024.
- [133] Vernon Valentine Palmer. Excused performances: Force majeure, impracticability, and frustration of contracts. *The Am. J. of Comp. Law*, 70(Supplement_1):i70–i88, 2022.
- [134] Monica Palmirani, Michele Martoni, Arianna Rossi, Cesare Bartolini, and Livio Robaldo. Pronto: Privacy ontology for legal reasoning. In *International Conference on Electronic Government and the Information Systems Perspective*, pages 139–152. Springer, 2018.
- [135] Monica Palmirani and Fabio Vitali. Akoma-ntoso for legal documents. In *Legislative XML for the semantic web: Principles, models, standards for document management*, pages 75–100. Springer, 2011.

- [136] Alexandros A Papantoniou. Smart contracts in the new era of contract law. *Digital LJ*, 1:8, 2020.
- [137] Alireza Parvizimosaed, Sepehr Sharifi, Daniel Amyot, Luigi Logrippo, Marco Roveri, Aidin Rasti, Ali Roudak, and John Mylopoulos. Specification and analysis of legal contracts with symboleo. *Software and Systems Modeling*, 21(6):2395–2427, 2022.
- [138] Giuseppe Peano. *Arithmetices principia, nova methodo exposita*. Fratres Bocca, 1889.
- [139] Henry Prakken. An abstract framework for argumentation with structured arguments. *Argument and Computation*, 1:93–124, 2010.
- [140] Henry Prakken and Giovanni Sartor. A dialectical model of assessing conflicting arguments in legal reasoning. *Artificial Intelligence and Law*, 4:331–68, 1996.
- [141] Henry Prakken and Giovanni Sartor. A formal framework for combining legal reasoning methods. In *Proceedings of the Nineteenth International Conference on Artificial Intelligence and Law*, pages 227–236, 2023.
- [142] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 174–189. Springer, 2007.
- [143] Lahiru Rajapakshe. Understanding the imperative programming paradigm in software development, February 13 2025. Accessed: 2025-08-25.
- [144] Pauline Ratnasingham. Edi security: The influences of trust on edi risks. *Computers & Security*, 17(4):313–324, 1998.
- [145] Norbert Reich, Hans-Wolfgang Micklitz, Peter Rott, and Klaus Tonner. *European consumer law*. Intersentia, 2014.
- [146] Stefano Rodotà. *Le fonti di integrazione del contratto*, volume 20. Roma TrE-Press, 2024.

- [147] Tom Routen and Trevor Bench-Capon. Hierarchical formalizations. *International Journal of Man-Machine Studies*, 35(1):69–93, 1991.
- [148] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25(27):79–80, 1995.
- [149] Galileo Sartor, Adam Wyner, Giuseppe Contissa, and Jacinto Davila. Mind the gap – the rules of the road for humans and machines. In *Proceedings of the JURISIN 2024 Workshop (Local Proceedings)*, pages 55–70, 2024.
- [150] Giovanni Sartor. Normative conflicts in legal reasoning. *Artificial intelligence and law*, 1:209–235, 1992.
- [151] Giovanni Sartor. Defeasibility in legal reasoning. In *Informatics and the foundations of legal reasoning*, pages 119–157. Springer, 1995.
- [152] Giovanni Sartor. Defeasibility in law. In *Handbook of legal reasoning and argumentation*, pages 315–364. Springer, 2018.
- [153] Giovanni Sartor. Argumentation in AI and law. In *Elgar Research Handbook on Legal Argumentation*. Elgar, 2025.
- [154] Ken Satoh, Kento Asai, Takamune Kogawa, Masahiro Kubota, Megumi Nakamura, Yoshiaki Nishigai, Kei Shirakawa, and Chiaki Takano. Proleg: an implementation of the presupposed ultimate fact theory of japanese civil code by prolog technology. In *JSAI international symposium on artificial intelligence*, pages 153–164. Springer, 2010.
- [155] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 11th edition, 2015.
- [156] Marek J. Sergot, Fariba Sadri, Robert A. Kowalski, Frank Kriwaczek, Peter Hammond, and H Terese Cory. The british nationality act as a logic program. *Communications of the ACM*, 29(5):370–386, 1986.
- [157] Jeremy M Sklaroff. Smart contracts and the cost of inflexibility. *U. Pa. L. Rev.*, 166:263, 2017.

- [158] S. Sprague and J. Steadman. *Codice civile in inglese*. Wolters Kluwer Italia, 2021.
- [159] Josh Stark. Making sense of blockchain smart contracts. CoinDesk, June 2016. Accessed: 2025-07-01.
- [160] Andrea Stazi. Protection of weaker parties in smart contracts. *Journal of European Consumer and Market Law*, 12(6):265–266, 2023.
- [161] Lin Sun, Kai Zhang, Fule Ji, and Zhenhua Yang. Toi-cnn: A solution of information extraction on chinese insurance policy. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Industry Papers)*, pages 174–181, 2019.
- [162] Harry Surden. Computable contracts. *UCDL Rev.*, 46:629, 2012.
- [163] Richard E. Susskind. Expert systems in law: A jurisprudential approach to artificial intelligence and legal reasoning. *The Modern Law Review*, 49(2):168–194, March 1986.
- [164] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2, 9 1997.
- [165] Nick Szabo. Winning strategies for smart contracts. *foreword by Don Tapscott, Blockchain Research Institute*, 4, 2017.
- [166] Nick Szabo et al. Smart contracts, 1994.
- [167] Roger Tagg, Zoran Milosevic, Sachin Kulkarni, and Simon Gibson. Supporting contract execution through recommended workflows. In *Database and Expert Systems Applications: 15th International Conference, DEXA 2004, Zaragoza, Spain, August 30-September 3, 2004. Proceedings 15*, pages 1–12. Springer, 2004.
- [168] UK Jurisdiction Taskforce. Legal statement on cryptoassets and smart contracts, November 2019.

- [169] Jasper Verstappen. *Legal Agreements on Smart Contract Platforms in European Systems of Private Law*. Springer, 2023.
- [170] Andrea Vitaletti and Marco Zecchini. A tale on decentralizing an app: the case of copyright management. In *DLT*, 2023.
- [171] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2/3):167–188, 1996.
- [172] Georg Henrik Von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.
- [173] Ian Waiden. Edi: contracting for legal security. *International Review of Law, Computers and Technology*, 6(1):23–35, 1992.
- [174] Angela Walch. 58in code(rs) we trust: Software developers as fiduciaries in public blockchains. In *Regulating Blockchain: Techno-Social and Legal Challenges*. Oxford University Press, 06 2019.
- [175] David S Warren. Introduction to prolog. In *Prolog: The Next 50 Years*, pages 3–19. Springer, 2023.
- [176] Seng Joe Watt, Oliver Goodenough, and Meng Weng Wong. Deontics and time in contracts: An executable semantics for the l4 dsl. In *Legal Knowledge and Information Systems. JURIX 2023*, Frontiers in Artificial Intelligence and Applications, pages 119 – 124. IOS press, 2023.
- [177] Stephen Weatherill. Eu consumer law and policy. In *EU Consumer Law and Policy*. Edward Elgar Publishing, 2005.
- [178] Kevin Werbach and Nicolas Cornell. Contracts ex machina. *Duke LJ*, 67:313, 2017.
- [179] Tian Xu. Smart contracts: The limits of autonomous performance. In *Smart Legal Contracts: Computable Law in Theory and Practice*. Oxford University Press, 04 2022.
- [180] Shuo Zhang, Junzhou Zhao, Pinghui Wang, Nuo Xu, Yang Yang, Yiting Liu, Yi Huang, and Junlan Feng. Learning to check contract inconsistencies. In

BIBLIOGRAPHY

- Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 14446–14453, 2021.
- [181] Reinhard Zimmermann. *The law of obligations: Roman foundations of the civilian tradition*. Juta and Company Ltd, 1990.