IMPERIAL COLLEGE LONDON

MENG FINAL YEAR PROJECT

Confis: A Framework for Authoring and Querying Machine-Readable Legal Agreements

Author:
Nicolás D'COTTA

Supervisor:
Prof. William J. KNOTTENBELT

Second Marker:
Dr. Paul A. BILOKON

A report submitted in fulfillment of the requirements for the degree of MEng Computing

in the

Faculty of Engineering
Department of Computing

17th June, 2022

Abstract

Confis: A Framework for Authoring and Querying Machine-Readable Legal Agreements

For more than four thousand years, humankind has utilised contracts to govern economic activity. Since their conception in ancient times, they have been represented as clauses of natural language and how we reason about them has remained unchanged, despite technological leaps and civilization's dependence on them. In the last few decades, since the seminal work on smart contracts by Szabo in 1997, the benefits of formalising and automating contracts have become increasingly obvious. Such benefits include making it easier for parties to understand the legal constraints they are under, as well as the capabilities they have. Rather than replacing existing legal infrastructure, some new technologies like blockchain-based smart contracts have created a new class of self-enforcing software systems which mostly revolve around financial products. Other, more logic-based attempts to automate reasoning with legal documents are very specialist and require understanding of advanced topics, such as logic programming, linear temporal logic, or the event calculus; and are not self-contained solutions. This lack of accessibility of existing technologies means that they have not seen practical adoption.

This project introduces ***Confis***, a Kotlin-based accessible language that specifies legal agreements and immediately answers complex queries. Contracts are written as standalone scripts inside a fully-capable editor, inside which a live preview displays a natural language render of the agreement. A graphical UI allows making questions involving parties' legal capabilities and their compliance, such as *'What needs to happen for my landlord evict me?'* or *'Has the seller breached our agreement?'* – which Confis is able to answer in plain English. In stark contrast with the state of the art, Confis does not require engineering, logical, nor legal expertise for its effective use.

Overall, this project operationalises legal agreements by providing the industry with relevant and pragmatic tools. By formalising the fewest key abstractions, we prove it successfully represents common classes of contracts (such as licenses and sales agreements) much like Symboleo or The Accord project – except with a strong focus on accessibility and in a self-contained solution. We show this accessibility is at the expense of expressiveness, and thus not being able to formalise some more specific legal scenarios, which limits the range of contracts Confis can specify. Next steps include communicating with other services for them to be aware of their legal capabilities (rather than relying on legal advice across teams inside an organisation), as well as improving the compromise between the contracts that can be represented and the complexity of the language.

Acknowledgements

I would like to thank William Knottenbelt, my supervisor for this project. I am grateful for his willingness to take on a project that started as my own proposal, and for his guidance and feedback throughout its evolution.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivations

The first forms of literacy and writing involved bills of sales like the one depicted in Figure 1.1 – documents representing agreements between people, specifying powers, obligations, and serving as proof of their converging wills [1].

FIGURE 1.1: Bill of sale of a male slave and a building in Shuruppak, Sumerian tablet, circa 2600 BC

Contracts are pervasive and drivers of economic change and social progress – it is no overstatement that civilization fundamentally relies on understanding legal documents to function. In more than four millennia, the nature of these documents has not changed at all: they contain clauses in natural language, and we still rely on human interpretation to understand them. In that time, the complexity of the concepts legal documents represent has increased substantially, to the point of having professions that specialise in interpreting legal documents. This lack of change is despite the pervasiveness of contracts, their importance as drivers of economic activity, and significant technological progress.

Some technologies (digital signing, PDF, collaborative drafting tools, etc) have changed the medium of the documents and have facilitated some hassles involved in dealing with them. These technologies mainly target the automation of bureaucratic overheads, rather than dealing with the contents of contracts themselves and trying to reason about their contents.

Since Szabo introduced the concept of smart contracts [2] the benefits of technology that helps us reason about the powers and obligations contracts bring has become clear. Such benefits include democratising understanding of legal documents, self-enforcing agreements, and having services programmatically understand their legal obligations and capabilities. This search for new machine-readable formalisms to represent a contract is evidenced by the extensive state of the art around reasoning about legal documents. Attempts to formalise legal agreements include smart contracts (see Subsection 2.2.1). While revolutionary in their capability to self-enforce agreements between parties and achieve consensus between disagreeing parties, the technology has created a new class of software systems rather than automating and disrupting the operations around existing legal documents. Other attempts include formalisations through programming languages like The Accord Rpoject [3] or logic calculi like Symboleo [4] (see Subsection 2.2.6). These tend to be expressive (in that they are able to formalise a broad range of legal documents), but are usually specified through programs or more complex specification languages or algebras.

This that means in practice they are specialist and require a strong background in engineering or logic programming to be authored and even read, which is a strong barrier to their use. They also tend to make little effort in compromising with existing frameworks for dealing with legal agreements: while being machine-readable is a desirable property of a contract, being human-readable (eg, for use in court) is a necessary property.

1.2 Objectives

From the above motivations we can derive the main objectives of this project. They are motivated by how existing technologies in the state-of-the-art (see Section 2.2 and Section 2.3) fail to address them, and are as follows:

- To develop a specification language both accessible and expressive enough to represent concepts that typically make up contracts (such as parties' obligations and powers)
- To develop an accessible analysis mechanism that helps parties understand their powers, obligations, and given a state of the world, understand whether they are (or how they can become) compliant with the agreement; as well as verifying said agreement is free of logical contradictions.
- To demonstrate the applicability of the developed framework in the context of legal agreements

1.3 Core Contributions

Please see Figure 1.3 for a diagram on how these contributions interact with each other.

1.3.1 A Language for Specifying Legal Agreements

The **Confis DSL** is a domain-specific language that specifies contracts. Like Applescript or Python, is engineered to resemble English. It is parsimonious and formalises a small set of concepts (legal obligations, powers, and the circumstances in which they apply) in order to be simple to learn and cover a broad range of contracts.

See Section 3.2 for more details on the language.

1.3.2 A Prototype Implementation of the Confis DSL

By using Kotlin as a *host language* for the DSL (see Domain-Specific Languages), we present a prototype which allows compiling the Confis language.

See Subsection 3.2.2 for more details on the language implementation.

1.3.3 An Intelligent Editor for the Confis DSL

By using an existing development environment, we greatly improve the accessibility of the Confis Language relative to other specification languages in the literature (like Accord or Symboleo) by also implementing a graphical editor that eases the drafting of Confis agreements and is able to provide human-readable live previews and report errors.

See Section 3.4 for more details on the editor.

1.3.4 A Querying Engine for Confis Agreements

We also introduce a rules engine that compiles a Confis Agreement and is able to answer complex queries by generating and evaluating rules depending on the type of query. It is able to answer queries relating to determining the legal capabilities of parties, finding contradictions in an agreement, determining clause violations depending on past events, and determining necessary steps towards compliance. To better illustrate the extent of the querying, some questions that can be executed as queries are:

- *'Under what circumstances can my landlord evict me?'*
- *'I have paid my rent late – am I in breach of the contract?'*
- *'What do I need to do to in order to be fully compliant?'*
- *'Am I allowed to derive statistics from this licensed dataset with commercial purposes?'*

In contrast with the state of the art, the solution is self-contained and does not require the user to translate a Confis specification into other formalisms.

See Section 4.2 for more details on the querying engine.

1.3.5 A User Interface for Querying Documents

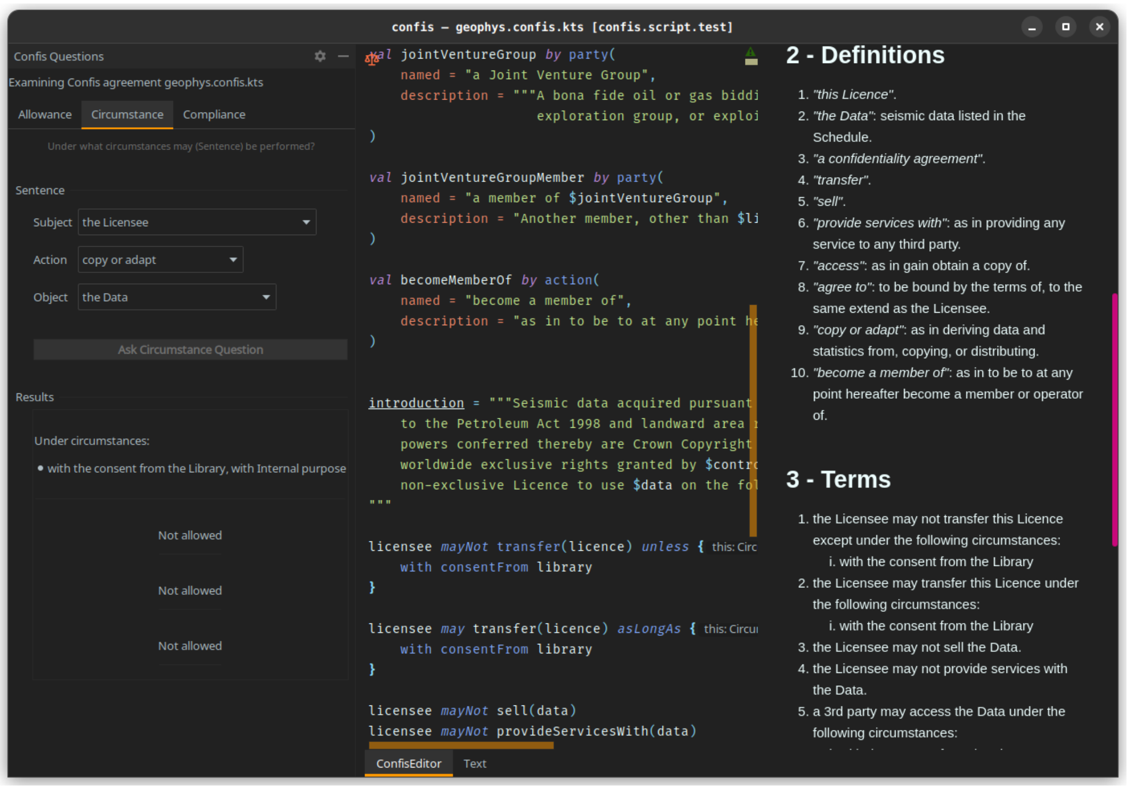
In addition to the formalisms necessary to examine a contract by querying it, we introduce a prototype that allows asking these questions through an accessible graphical user interface, and producing understandable human-readable answers to them.

See Section 4.3 for more details on the querying UI.

1.4 Project Overview

This section hopes to provide a more concrete high-level overview of how the core contributions together form the Confis framework, which combines the different user interfaces, the editor, and the human-readable preview. A screenshot depicting the user interfaces can be found in Figure 1.2.

An architectural diagram of how all components interact with each other in Figure 1.3.


confis – geophys.confis.kts [confis.script.test]
Confis Questions
Allowance
Circumstance
Compliance
Ask Circumstance Question
ConfisEditor
Text

Examining Confis agreement geophys.confis.kts

Under what circumstances may (Sentence) be performed?

Sentence

Subject: the Licensee
Action: copy or adapt
Object: the Data

Results

Under circumstances:

- with the consent from the Library, with Internal purpose

Not allowed

Not allowed

Not allowed

```
val jointVentureGroup by party(
    named = "a Joint Venture Group",
    description = """A bona fide oil or gas biddi
                  exploration group, or exploi
)

val jointVentureGroupMember by party(
    named = "a member of $jointVentureGroup",
    description = "Another member, other than $li
)

val becomeMemberOf by action(
    named = "become a member of",
    description = "as in to be to at any point he
)


introduction = """Seismic data acquired pursuant
    to the Petroleum Act 1998 and landward area
    powers conferred thereby are Crown Copyright
    worldwide exclusive rights granted by $contr
    non-exclusive Licence to use $data on the fol
"""

licensee mayNot transfer(licence) unless { this: Circ
    with consentFrom library
}

licensee may transfer(licence) asLongAs { this: Circu
    with consentFrom library
}

licensee mayNot sell(data)
licensee mayNot provideServicesWith(data)
```

2 - Definitions

1. *"this Licence"*.
2. *"the Data"*: seismic data listed in the Schedule.
3. *"a confidentiality agreement"*.
4. *"transfer"*.
5. *"sell"*.
6. *"provide services with"*: as in providing any service to any third party.
7. *"access"*: as in gain obtain a copy of.
8. *"agree to"*: to be bound by the terms of, to the same extend as the Licensee.
9. *"copy or adapt"*: as in deriving data and statistics from, copying, or distributing.
10. *"become a member of"*: as in to be to at any point hereafter become a member or operator of.

3 - Terms

1. the Licensee may not transfer this Licence except under the following circumstances:
   i. with the consent from the Library
2. the Licensee may transfer this Licence under the following circumstances:
   i. with the consent from the Library
3. the Licensee may not sell the Data.
4. the Licensee may not provide services with the Data.
5. a 3rd party may access the Data under the following circumstances:

FIGURE 1.2: Screenshot of the Confis Editor and Query UI, depicting a Confis version of a license for seismic data [5]
From left to right: the **Query UI** that allows querying the agreement; the **Confis code** specifying the agreement; a **live preview** of the agreement in generated English text.

The main body of this report is divided in two parts: Chapter 3, covering the Confis language, and Chapter 4, covering the querying of documents specified in said language. Both chapters discuss the requirements, design, tooling, and implementation of each part.

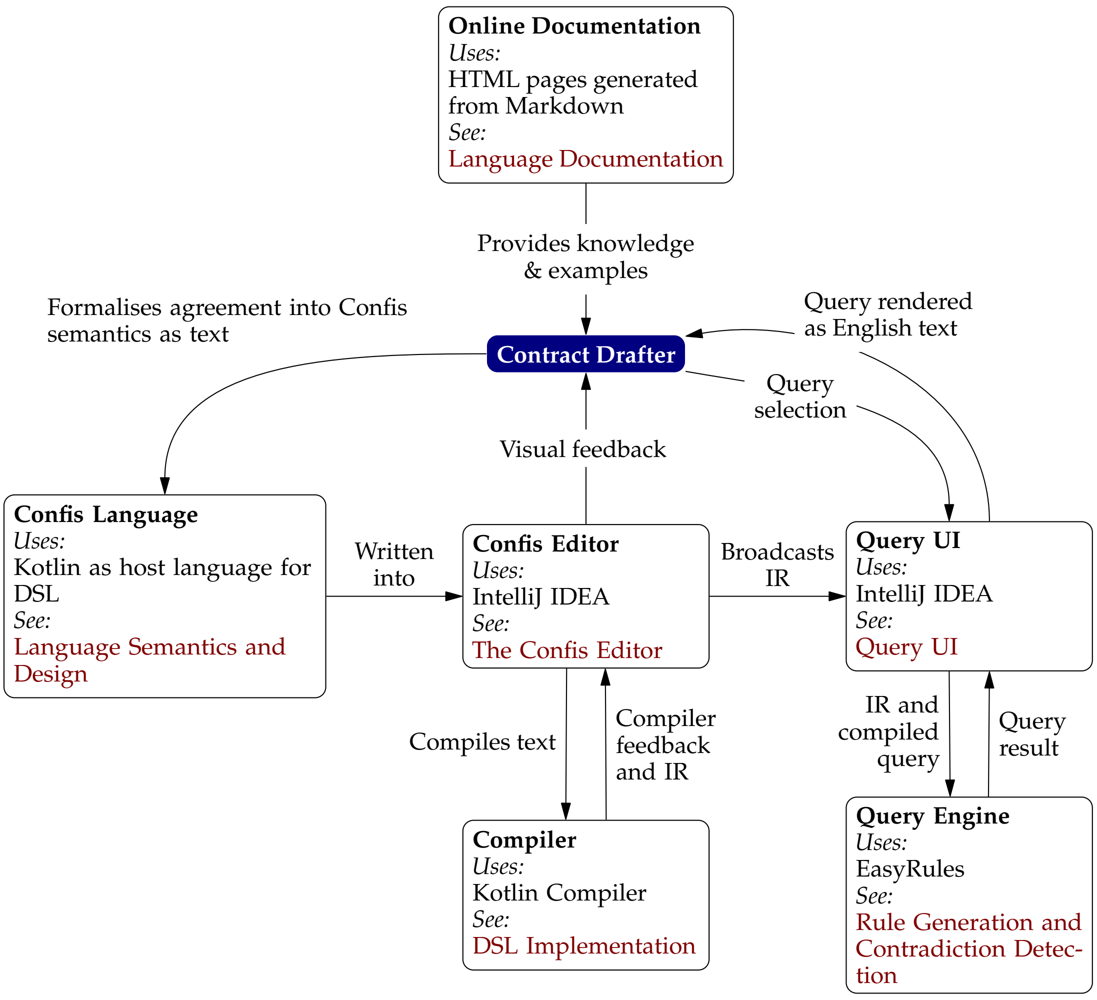


FIGURE 1.3: Overview of the different components of Confis

Chapter 2

Background

2.1 Legal Agreements

A legal agreement is a fairly broad definition which varies with each country's legal system. In common law, most generally it involves [6, 7]:

- An offer by a party
- An acceptance by at least one other party
- Intention to create a legal relationship
- Consideration for the offer – a sort of 'promise' where a price is paid for the offer (monetary or otherwise).

A licence is a specific kind of legal agreement which allows a party to perform an activity under some specific conditions [8].

2.1.1 Dataset Licensing

See [5] for an example of a licence allowing access to data by a library. The licence includes the following clauses, cherry-picked and simplified here for clarity:

- The licence is non transferable.
- The data cannot be sold or used to provide a service
- The licensee can provide access to the data to third parties provided they sign a confidentiality agreement. The licensee becomes liable for any breach of the licence by such third party, so it is in the licencee's interests that the third party also complies to the licence.
- The data cannot be copied or modified for any other use other than the licensee's.

See a licence by The Economist Group [9, §2.1] for an example where a clause enforces confidentiality of the terms of the agreement, not just of the data being shared.

2.1.2 Software Licensing

See [10] for an example of a software licence between a software company (JetBrains) and users for the use of their development software. The licence includes the following clauses, cherry-picked and simplified here for clarity:

- The licence is non-transferable.

- The software must be used only for specific but vague purposes Jetbrains s.r.o. [10, for non-commercial, educational purposes only].
- Only the licensee may use the software, but they may install it in multiple machines simultaneously.
- The software cannot be sold, rented, or modified.
- The licensee may provide feedback - if they choose to do they themselves grant JetBrains a very permissive licence to use it.
- There is a number of exceptions where JetBrains can revoke access - but in general, they may at any point terminate the agreement without a specific reason.
- The user may choose to use an early access development version of the software, subject ot its own separate licence.

While this contract is not machine-readable, JetBrains does have mechanisms in place to enforce its interests. Their software comes with a launcher, *JetBrains Toolbox* [11] that controls access to it depending on the licence status of the user. This is most likely implemented in a centralised server controlled by JetBrains where it keeps records of their licensees.

2.2 Previous Work on Machine-Readable Contracts

In the context of this project, *machine-readable* [12, 13] refers to an encoding for a formalisation of a legal agreement which can be processed by a suitable program. The processing must relate specifically to uses of legal documents (therefore formats like Microsoft Word, plain UTF encoded text, or PDF do not qualify).

See also Section 2.3 for normative rule generation from legal text, and Subsection 2.2.6 for reasoning with existing formalisms.

2.2.1 Smart Contracts and Ethereum

Smart contracts [2], where a protocol formalises a secure agreement or program over a network, are supported to some extent in the first blockchain application, Bitcoin [14] (see C.2) through scripting [15].

A bitcoin script is a list of instructions in the *Script* language recorded with each transaction (see Transactions) that gets executed by every participant that verifies that block and describes how the payee can access the coins being transferred. Bitcoin's Script has several limitations, such as lack of Turing-completeness (not all computable functions can be represented in Script), value blindness (a script cannot easily specify an exact amount to be withdrawn) and lack of internal state (beyond whether a transaction output is spent or unspent) [16].

Ethereum [16] is also a blockchain sharing many of Bitcoin's features, such as transactions and a proof-of-work consensus algorithm. While it also as scripting support, it varies greatly from Bitcoin's in that its built-in programming language is Turing-complete, value-aware and allows keeping internal state. For more details on Ethereum's scripting language, see Section 2.2.1. The name of the coin of the Ethereum ecosystem is *ether*.

Ethereum's state is encoded in *accounts* - objects with their own address that contain the account's balance (in ether), the contract code (if present), the nonce (a transaction counter), and the account's storage (a persistent key-value store which is initially empty).

Ethereum introduces an application of the concept first presented by Szabo in [2], in that it allows defining self-enforceable relationships between entities. It is implemented thanks to the Ethereum Virtual Machine.

The Ethereum Virtual Machine

Ethereum's built-in scripting language is a stack-based low-level bytecode language called *EVM language*, and runs on an execution environment called the *Ethereum Virtual Machine* (EVM). Besides its stack, the EVM does have memory (an infinitely expandable byte array) which resets after each computation. For persisting data, the account's storage is used [16].

There are several high level languages that compile to EVM bytecode that are used to write smart contracts, such as Solidity [17] and Vyper [18] (the most active and maintained languages at the time of writing [19]).

An application built on top of decentralised smart contracts is commonly called a *Decentralised Application* (or DApp) [20].

2.2.2 Juro

Juro [21] is a start-up business seeking to make structured machine-readable data out of contracts. It focuses on:

- Enabling easy collaboration between contract authors.
- Auditing and keeping track of the *workflow* of the agreement (version control of the documents' drafts, signatures, amendments, views, etc.)
- Improving the user experience of employees involved in legal agreements by integrating into the company's CRM *(Customer Relationship Management)* software (such as new hires and the Human Resources department).
- Ease of use by making legal documents searchable.

Juro aims to achieve this by encoding the contract as a JSON document [22], written by end-users with the assistance of a web graphical user interface (GUI). This document keeps all version control information and can be rendered into a human-readable format or exported to PDF [23] or Microsoft Word [24] formats. The JSON document can be further processed to provide extra analytics and tools, like summarization statistics or agreement renewal reminders.

Juro's whitepaper lists 'self-executing' clauses as future work [21, p. 6]. Therefore Juro does not attempt to make any contributions with respect to verification of compliance or enforcement of contract clauses.

2.2.3 The Accord Project

The Accord Project [3] is an open source Linux Foundation project with the aim of enabling machine-readable and machine-executable legal agreements.

Contracts are written pre-signature in a human-readable markup language similar to Markdown [25] where typed variables can be embedded. During drafting, functions can be defined with a Domain Specific Language (DSL) that allow to perform computations (such as compounding interest over a period of time).

After signature, more behaviour can be written in code to check information against a contract (like checking for breach of contract, or what action to take in a specific situation). See Listing 2.1 for an example where the clause checks for the state of a delivery (whether it is timely, has passed inspection, etc).

```
contract SupplyAgreement over SupplyAgreementModel {
  clause acceptanceofdelivery(request : InspectDeliverable) : Response {

    let received = request.deliverableReceivedAt;
    enforce isBefore(received,now()) else
      throw ErgoErrorResponse {
        message : "Transaction time is before the deliverable date."
      }
    ;

    let status =
      if isAfter(now(),
        addDuration(
            received,
            Duration { amount: contract.businessDays, unit: days }
        )
      )
      then OUTSIDE_INSPECTION_PERIOD
      else if request.inspectionPassed
      then PASSED_TESTING
      else FAILED_TESTING
    ;
    return Response {
      status : status,
      shipper : contract.shipper,
      receiver : contract.receiver
    }
  }
}
```

LISTING 2.1: A contract clause encoded in Accord's *Ergo* language, from [26].

Ergo code (the language in which Accord allows specifying contract behaviour [27]) can then be deployed to a permissioned open-source enterprise-oriented blockchain called *HyperLedger Fabric* [28, 29] or to a NodeJS server [30].

While the Accord project does also aim to represent machine-readable contracts, its representation cannot be easily written by a non-software developer (unlike Juro's, Subsection 2.2.2).

2.2.4 Proof of Existence

Among other contributions that ease the process (not unlike Juro, Subsection 2.2.2) of negotiating and drafting a contract, [31, Express Agreement] implemented Proof of Existence, where parties put a hash of a signed copy of a contract in the Bitcoin blockchain (see Section C.2).

This allows irrevocable proof of agreement verifiable by third parties.

2.2.5 Ricardian Contracts

First introduced by [32, 33], Ricardian Contracts aim to encode a legal agreement as a human-readable document (handwritten legal prose) accompanied by machine-readable metadata. Additionally, it is cryptographically signed (see Subsection C.1.2) by parties and uniquely identifiable; giving it some extra cryptographic guarantees (for example, signatures cannot be plausibly denied and terms cannot be tampered with after signing).

Definition 1 (Ricardian Contract)**.** A Ricardian contract can be defined as a document that is [32, §3.1]:

- a contract offered by an issuer to holders,
- for a valuable right held by holders
- such valuable right is managed by the issuer,
- human-written,
- readable by programs,
- digitally signed,
- carrying public keys (necessary to verify signatures),
- uniquely identifiable through a hash digest.

Using human-written legal prose has the advantages of leading to faster dispute resolution, better enforceability and more transparency when operating within the classical framework of agreements – for all purposes, there is no difference between a traditional agreement and a Ricardian contract in this area.

Drawbacks

While Ricardian contracts can represent any type of agreement, they do not actually try to capture the *meaning* of a specific type of agreement.

For example, a Ricardian contract may capture all the necessary information related to a Tenancy Agreement in a format that may allow a machine to enquire about the monthly rent – but such a machine would need to know how tenancy agreements are structured. In short, Ricardian contracts must follow a template and it is instances of this template that are easily read and queried.

2.2.6 Reasoning with Legal Agreements

The process of representing or acquiring knowledge from a legal agreement is typically considered as a process of writing conditional rules with the following basic conditional structure [34, 35]:

$$r : \text{IF } a_1, \ldots, a_n \text{ THEN } c \tag{2.1}$$

Where r is the unique identifier of the rule, $a_1, \ldots, a_n$ are the *antecedent* representing the conditions (which includes the context under which the rule is created) of applicability of the norm, and c is the *conclusion* representing the effect of the norm.

Note how Juro does not try to capture norms at all, and The Accord Project chooses to use the abstraction of a program rather than that of a set of rules. NLP, on the other hand (discussed in Section 2.3) struggles to extract such rules from existing documents.

Contract-Driven Agents

Contract-Driven agents [36] introduces a specification for formalising the behaviours involved in a contract and the interactions between parties

The formalism involves establishing a calculus based in multi-agents systems specifically oriented towards contracts, through the event calculus as first introduced by [37] (a logic-based system). Note how the event calculus conforms to Equation 2.1.

Unlike Ricardian contracts (see Subsection 2.2.5), contract-driven agents attempt to describe the behaviours of all parties as well as how they react to other parties' behaviours. Therefore, unlike Ricardian contracts, they also capture the semantics of the agreement between all the parties.

Note that although contract-driven agents primarily target describing behaviours of agents, rather than representing a legal agreement in the context of a legal jurisdiction (contrary to Symboleo [4] or The Accord Project [3]).

Symboleo

Symboleo [4] is a formal contract specification Language. Much like Contract-Driven Agents it aims to capture the semantics of the agreement. It does this by providing several abstractions relating to legal concepts, such as *powers* and *obligations*, and creating axioms around them.

Time and events are represented by a model based on temporal logic [38] which specifies time instances for events, time intervals for situations, and makes use of pre-state and post-state situations for events. Symboleo also hopes to provide an implementation-agnostic specification by transpiling to blockchain languages, such as Solidity [17] (see Section 2.2.1) – much like The Accord Project. At the time of writing, Accord has such an implementation in place while Symboleo does not.

An example of a contract (a meat sales agreement that can be found in Table A.1) translated into Symboleo is given under the Appendix in Listing A.6. A detailed comparison of Symboleo and Confis can be found in the Evaluation chapter.

2.3 Natural Language Processing and its Limitations

Human-written text, which is how legal agreements are typically persisted[1] can be converted into machine-readable form by attempting to translate the natural language in which they are written through Natural Language Processing (NLP), which uses techniques such as symbolic reasoning and deep learning [39].

Existing technologies trying to process legal text – such as [40], [41], [42], or [43] – try to produce normative rules.

[1]Note other formats like verbal agreements may also constitute legal agreements [1]

At the time of writing, they fall short of fully capturing all relevant information in an existing legal document – therefore they lack the expressiveness property discussed in Section 1.2, because they are unable to represent a contract well. They face the main following challenges [35]:

Cross-referencing Legal text is structured into sections which contain sentences which represent clauses applicable in a specific context, and documents therefore often reference information across sections or across documents. Some of these references may not even be explicit: often, contracts have a 'Definitions' section solely for the role of disambiguating terms which the reader is expected to use in their understanding of the document.

NLP solutions must therefore be able to deal with referencing in order to extract context and resolve lexical ambiguities.

Ambiguity While legal documents aim to remove ambiguity and ideally produce a single interpretation, unintended ambiguities arise from the use of natural language. The most likely ambiguity is *referential ambiguity* where a word or phrase has multiple meanings inferred from current context, from a parent statement (which is unique to legal text), or from other document sections via cross-referencing. Natural language can also represent different logical interpretations, which is referred to as *local ambiguity*. An example of this is using the term "and" to represent a disjunction instead of a conjunction.

Sentence Complexity Sentences used in legal prose tend to be much longer than sentences from other domains. The average number of lexical units in a sentence written in English in Wikipedia is about 19 [44], while sentences from a legal document can have more than 50 [35] as well as have complex syntactic structures (see for example Table 2.1).

Current NLP technologies struggle with such sentences: syntactic analysers and predicate-argument extraction tools sometimes do no correctly capture the scope of coordinate conjunctions, nor the antecedents of subordinate phrases.

On termination of this Licence the Licensee shall cease to have any rights or Licence in respect of the Data or any adaptation thereof and shall cease to use the same and shall erase the Data or any adaptation thereof from any storage apparatus and shall return to the Library without demand the Data and all existing copies thereof made by the Licensee and shall warrant in writing to the Library that all data, plots, displays, results, analyses, variations and modifications derived from the Data are destroyed.

TABLE 2.1: Clause §3.2 of *Sub-License for Seismic Data* [5]

Normative Effects and Modalities Legal documents typically encode norms, and words specific to legal concepts such as *right, power, immunity, liability, privilege*, etc significantly alter the way a sentence is interpreted. Properly capturing modalities or behaviours such as a *requirement* or a *permission* is a major difficulty the state of the art of Artificial Intelligence in Law struggles to deal with [35].

2.4 Domain-Specific Languages

The use-case of representing complex, formalised abstractions in a machine-readable formats is a common one, and general-purpose programming languages (Java, Python, etc) are designed with exactly this goal in mind.

A class of languages made to target a specific problem, rather than being general purpose, also exists: these are *Domain-Specific Languages* (or *DSLs*). According to [45]: "A DSL is a computer language that is targeted to a particular kind of problem, rather than a general purpose language that is aimed at any kind of software problem". In order to write legal agreements and lower the barrier of entry as much as possible (ideally, no software engineering training should be required) a well-documented and easy-to-use DSL is a good solution: expressive enough to be able to represent a large range of agreements, but restrictive enough that language features should not require extensive training.

Several modalities of DSLs exist that can be used. The compromises they make are in cost (for developing a framework for them), ease of writing, readability, and expressiveness [46, 47].

Textual Internal DSLs Textual Internal DSLs make use of a host language, and use that language constructs in particular ways to give the internal DSL a particular feel. They share their syntax with and are a subset of their host language. This solution is the easiest to develop but forces the DSL author to conform to a host language and to extend their DSL within the bounds of what the host language can do.

Textual External DSLs Textual External DSLs have their own custom syntax, and therefore they need a whole new parser to be processed. This solution provides the most flexibility in what the language can express but is more expensive in terms of development efforts for the DSL author.

Graphical DSLs While Textual DSLs require writing and parsing text, graphical DSLs can be achieved with a tool providing a graphical user interface that then builds some internal representation that can be processed. This solution is the most accessible for users of the DSL with non-technical backgrounds but is also expensive in terms of development cost, as both an internal representation and a GUI builder must be developed.

2.4.1 Candidates for Internal DSL Host Languages

This project is concerned with making a DSL with low development costs, and as such it will consider several platforms as host language candidates for an internal DSL.

Definition 2 (DSL Host Traits)**.** Languages desirable to be host for DSLs usually have some of the following features:

- **Operator overloading** for easily including symbols familiar to most users in the language, such as `‘+’`, `‘-’`, `‘&’`, etc.
- **Infix functions** for constructing sentence-like expressions or statements, such as `‘if car is parked’`.
- **Scripting facilities** that allow compiling a small text snipped without having to define a full program that includes an entrypoint.
- **A Tooling Platform** which allows developing tools which enahnce the wiritng experience. Examples of this include GUI builders and IDE features (like syntax highlighting and inline compiler warnings).
- **Scripting** (see Subsection 2.4.2) which allows parsing stand-alone text files in the host language that are interpreted by a DSL-aware host.

Haskell

Haskell [48] is a functional programming language which can be compiled or interpreted. It has advanced features including most of the ones mentioned in DSL Host Traits.

These are demonstrated in Listing 2.2, where Haskell functions are used as commands and statements of the BASIC language. This results in a valid Haskell expression (with some language modifications) that also defines a valid BASIC program – therefore allowing to write and compile BASIC without actually needing a BASIC parser.

Haskell additionally permits defining the precedence of operators and infix functions, allowing to define the direction in which associativity propagates [50]. This can be useful to form English-like sentences in a DSL such as `‘Bob did eat after Alice did eat’`, where `‘after’` can be an infix function with lower precedence that `‘did’`, which forms the other two sentences.

Tooling Haskell tooling is comparatively lacking next to more mainstream general purpose programming languages (such as Kotlin) in that the communities developing its tooling platform are smaller.

Groovy

Groovy [51] is a general purpose language with plenty of features specifically designed for DSLs. One of the most used examples of such DSLs is the Gradle Build Language [52] which specifies configuration for the Gradle build tool. This sets the build tool apart from similar tools, like Maven and Makefiles, which do not use DSLs. An example of a script for such a DSL is given in Listing 2.3.

Unlike to the other candidates in this section, Groovy allows for dynamic typing. This means that the language offers more runtime flexibility when designing a DSL, as new variables and classes can be created as the script is interpreted.

Tooling Groovy has feature-rich tooling that allows parsing standalone scripts with a DSL context, but its external tooling (like intelligent editors) is limited due to the nature of its runtime dynamic typing. In the example of Listing 2.3, an editor would not be able to

```haskell
{-# LANGUAGE ExtendedDefaultRules, OverloadedStrings #-}
import BASIC

main = runBASIC $ do
    10 GOSUB 1000
    20 PRINT "* Welcome to HiLo *"
    30 GOSUB 1000

    100 LET I := INT(100 * RND(0))
    200 PRINT "Guess my number:"
    210 INPUT X
    220 LET S := SGN(I-X)
    230 IF S <> 0 THEN 300

    240 FOR X := 1 TO 5
    250   PRINT X*X;" You won!"
    260 NEXT X
    270 STOP

    300 IF S <> 1 THEN 400
    310 PRINT "Your guess ";X;" is too low."
    320 GOTO 200

    400 PRINT "Your guess ";X;" is too high."
    410 GOTO 200

    1000 PRINT "*******************"
    1010 RETURN

    9999 END
```

LISTING 2.2: A BASIC program written in a Haskell DSL, from [49]

suggest autocompletion for '`useJUnitPlatform()`' inside the configuration of the `test` task.

Kotlin

Kotlin [53] is another general-purpose language. The language supports infix functions and operator overloading. An example DSL that uses Kotlin as the host language is given in Listing 2.4.

Tooling While the language is not as flexible as Haskell or Groovy, it is supported by well-maintained and feature-rich external tooling (in terms of the *IntelliJ IDEA* editor) and has stand-alone scripting support Subsection 2.4.2.

Additionally, the editor IntelliJ IDEA supports live evaluation of Kotlin inside text boxes, allowing to provide hints at the same time it compiles a specific snippet of code [55].

2.4.2 Scripting Support

Depending on the DSL being developed, it may be necessary for the host language to provide scripting support. By *scripting support*, we mean allowing a program to evaluate a file or a string of text, at runtime, and share properties with the file. We will call this file a *script*.

```
plugins {
    id 'groovy'
    id 'application'
}
repositories {
    mavenCentral()
}
dependencies {
    implementation 'org.codehaus.groovy:groovy-all:3.0.9'
    implementation 'com.google.guava:guava:30.1.1-jre'
    testImplementation 'junit:junit:4.13.2'
}
application {
    mainClass = 'demo.App'
}
tasks.named('test') {
    useJUnitPlatform()
}
```

LISTING 2.3: A Gradle Groovy build script, from [52]

In general, interpreted languages offer this functionality through '`eval(...)`' methods or similar. They may not be necessarily well-suited for a DSL as they also tend to have weak or dynamic typing (this is the case of Python, for example). Other compiled languages may offer this functionality:

- Java offers it through JSR 223 [56] and allows languages other than Java in the scripts.
- Kotlin [57].
- Groovy [58], which is widely used inside the Gradle build system (see Listing 2.3 for an example) where Gradle DSL scripts make up the configuration build files (in a way comparable to Makefiles).

For DSLs in particular, there is the requirement that the scripting host is aware of the DSL when evaluating the script [57]. This is what allows using a library-defined function in a stand-alone file without compiling errors. Notice how this is not the case of the Haskell and Kotlin DSL examples in Listing 2.2 and Listing 2.4 (where imports and program entry-points are necessary).

2.5 Language Tooling

Most Integrated Development Environments (or IDEs) have support for *extensions* or *plugins* that allow extending their functionality [59, 60]. A common use-case for this is enhancing developer productivity by providing visual cues and aids to textual development by, for example, reporting compilation errors and warnings inside an editor.

Extensions also typically allow arbitrary graphical user interfaces inside the program window, with access to the contents of the files opened.

2.5.1 Custom Scripting Support in IntelliJ IDEA

The IntelliJ IDEA editor [55] allows a specific extension point for providing script definitions [57, 59] and enabling development features such as syntax highlighting, error reporting, and autocompletion inside custom-defined scripts.

```
import com.example.html.*

fun main(args: Array<String>) {
    val generated = html {
        head {
            title {+"XML encoding with Kotlin"}
        }
        body {
            h1 {+"XML encoding with Kotlin"}
            p  {+"this can be used as an alternative to XML"}
            // an element with attributes and text content
            a(href = "https://kotlinlang.org") {+"Kotlin"}
            // content generated by
            p {
                for (arg in args) +arg
            }
        }
    }
    println(generated.toString())
}
```

LISTING 2.4: Kotlin DSL for building HTML, from [54]

2.6 Rules Engines for the JVM

This project requires both rule evaluation and making a Domain-Specific language with the same host language. This rules out some logic programming languages appropriate for rules evaluation (namely, Prolog [61]) as well as Answer Set Programming solutions (such as the Potassco Project [62]) because of their lack for DSL facilities (see DSL Host Traits for details on what some of the requirements are).

2.6.1 Easy Rules

Easy Rules [63] is a rules engine [64] for Java (therefore it can also be used in Kotlin). It allows programmatically generating rules (through `Rule` instances) as well as evaluating them. This API is particularly swell-suited to this project because it allows defining the predicate that triggers rule execution (the `when` clause in Listing 2.5) at runtime rather than at compile-time.

```
Rule weatherRule = new RuleBuilder()
        .name("weather rule")
        .description("if it rains then take an umbrella")
        .when(facts -> facts.get("rain").equals(true))
        .then(facts -> System.out.println("It rains, take an umbrella!"))
        .build();
```

LISTING 2.5: Programmatic Example of a Rule from Easy Rules, from [63]

Rules evaluation can be customised to add a notion of priority between rules (which might allow to create precedence between clauses in contracts). Additionally, there is support for continuous rule application, where rules are evaluated on known facts continuously until no more rules are applicable. This allows building a knowledge base through inference, rather than just evaluating rules sequentially.

Chapter 3

A Domain Specific Language for Legal Agreements

One of the key focuses of this project is not just to be able to represent legal contracts with specific properties and capabilities (as described in Chapter 4); but also to make it as easy as possible for people with non-technical backgrounds to use such representations. Simply put, a lawyer should not need to learn JSON nor temporal logic.

This is the main motivation for developing tooling which aims to make it easier for people of non-technical backgrounds to produce, modify, and understand Confis legal agreement representations. The core of this tooling is the Confis language (Section 3.2) and an IDE-assisted editor (Section 3.4).

3.1 Motivations For a DSL

3.1.1 Implementation Requirements

The language should fulfil the core requirements set out in Section 1.2. This involves prioritising accessibility while making sure the language stays expressive enough.

Easy to write while still machine-readable

Writing a Confis agreement should be close to writing natural language, while still being machine-readable. For more details on the meaning of *machine-readable* in the context of this project, see Section 2.2.

Data Serialization Language On one extreme, a data serialization text file (like JSON, YAML, or XML) would allow writing text that can be easily parsed by a program. But writing such files requires some data structures knowledge; and because of their key-value nature they do not allow writing sentences, leaving them too far from the readability of human-written legal prose – thus they lack they are not accessible.

Natural Language Processing On the other extreme, legal prose processed through a language processing program allows the drafter to completely ignore the machine-readable aspect of the document. Readers would be able to integrate such documents in their existing workflows – as they would need to make no transition from their existing, non-machine-readable documents. This is approach is discussed in Section 2.3 – we conclude it does not meet the expressiveness property faithfully enough, since it cannot specify contracts accurately.

A compromise between these two solutions would be a language formal enough that it can be parsed by a program, but natural enough that natural language sentences can be recognised in it. Python or AppleScript [65] (see Listing 3.1) are good examples of programming languages (and therefore parseable) that are engineered with the goal of resembling English as much as possible.

```
set the firstnumber to 1
set the secondnumber to 2
if the firstnumber is equal to the secondnumber then
    set the sum to 5
end if
```

LISTING 3.1: Sample code snippet of the AppleScript Language [65]

Easy to develop and extend With pragmatic implementation efforts in mind, this project should not aim to develop its own parser and interpreter or compiler: the project hopes to be more concerned with introducing a suitable abstraction that allows specifying legal documents.

Additional tooling for ease of use and correctness A key aspect of development for a new user of a language is to understand the concepts of syntax, compile errors, and invalid programs. A great aid to developing this understanding are visual cues in editors in integrated development environments (or IDEs).

For the Confis DSL to be successful, it must be easy for the drafter of legal agreements to reason about the correctness of the document within the formalisms set out by this project. Good tooling is therefore a key requirement in order to achieve accessibility, a key priority as discussed in Section 1.2.

3.1.2 Implementation Requirements Conclusion

The requirements of Subsection 3.1.1 lead to the following design choices:

A Textual Domain-Specific Language For a good compromise between readability, flexibility and rigidity of the agreements that can be written, this project therefore proposes developing a DSL (see Domain-Specific Languages) as a suitable compromise that allows working with a human-readable encoding which can then be compiled to a suitable machine-readable representation.

We will call this language ***Confis DSL*** (or *Confis language*) and the machine-readable representation it compiles to ***Confis Internal Representation*** (or *Confis IR*). The Confis IR will be used for processing as discussed in Chapter 4.

An Internal DSL For the sake of development costs, we use a suitable host language to implement the Confis DSL (as opposed to developing an external DSL). The alternative (developing an entirely new language) is discussed in Section 7.1.

Kotlin as a host language Several host languages can serve to build a DSL. A few options are discussed in Subsection 2.4.1, like Groovy and Haskell. The choice to use Kotlin stems from how feature-rich the surrounding tooling is – such as its stand-alone custom scripting [57] and IDE support [55].

3.2 Language Semantics and Design

This section is concerned with the structure, semantics, and design decisions regarding the Confis DSL. For the implementations details of the DSL, see DSL Implementation.

As described in Section 1.2, both accessibility and expressiveness are key priorities in designing the language. Confis introduces the following formalisms in order to limit the complexity of the language. These definitions make up the internal representation (IR) and are used to later generate rules to effectively answer complex queries. The language is designed to easily specify these abstractions. Therefore, this project tries to keep abstractions simple and introduce as few and as simple of them as possible, in order to design a parsimonious language.

Definition 3 (Party)**.** A *Party* to the agreement is a legal person (a real person, an institution, a company, or even a group of people). Parties are unique in an agreement and are identifiable by a name (encoded as a UTF-8 string).

Definition 4 (Action)**.** An *Action* represents something that can be initiated by a Party. It is unique in an agreement and is identifiable by a name (encoded as a UTF-8 string).

Definition 5 (Thing)**.** A *Thing* is unique in an agreement and is identifiable by a name (encoded as a UTF-8 string). An Thing resembles a Party but it does not aim to represent a legal person and it cannot be a subject on a Sentence.

A Sentence is the building block of a Confis agreement. It is combined with other elements in order to form clauses and, combined with a Circumstance Set, it forms an event that includes a context.

Definition 6 (Sentence)**.** A *Sentence* is a tuple like `Sentence = (Subject, Action, Object)`, where a `Subject` is a Party and `Object = Party | Thing`.

Allowance is used to answer questions of the type *'May A do X?'* – where *yes* or *no* are not enough to model all the possibilities given in the legal domain.

Definition 7 (Allowance)**.** *Allowance* is a value like:

```
Allowance = Allow | Forbid | Unspecified | Depends
```

Users are allowed to use Allowance in order to define a Permission, but only with the values `Allow` and `Forbid`. The other values are reserved for answering queries and while they cannot be used, responses that return `Depends` and `Unspecified` can be crafted by defining the answer space accordingly. This is shown in Figure 3.1 and Equation 3.1, where only `may` and `mayNot` keywords are used (corresponding to `Allow` and `Forbid` respectively) but all possible values of Allowance are present in the agreement.

```
val alice by party
val eat by action
val cake by thing

alice may eat(cake) asLongAs {
    with purpose Research
}
alice mayNot eat(cake) asLongAs {
    with purpose Commercial
}
```

LISTING 3.2: Minimal 2-clause agreement with disjoint Permissions

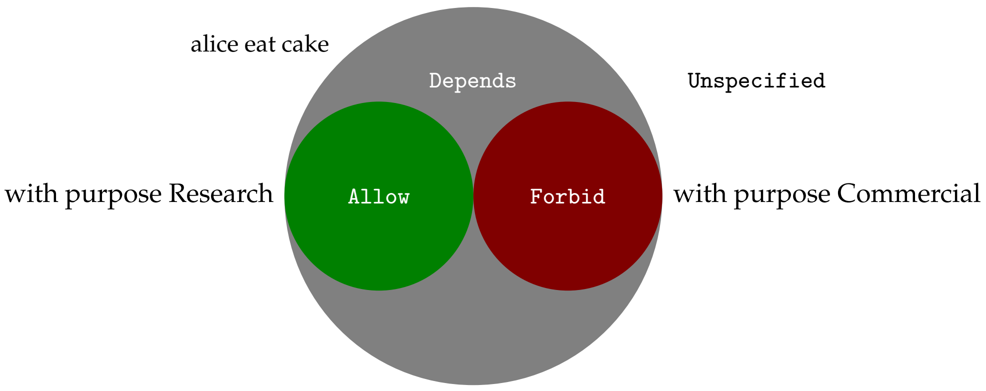


FIGURE 3.1: Venn Diagram of a '`alice eat cake?`' query for Listing 3.2

$$\text{alice eat cake?} \rightarrow \texttt{Depends} \tag{3.1}$$
$$\text{alice eat cake with purpose Research?} \rightarrow \texttt{Allow} \tag{3.2}$$
$$\text{alice eat cake with purpose Commercial?} \rightarrow \texttt{Forbid} \tag{3.3}$$
$$\text{alice eat cake with purpose Internal?} \rightarrow \texttt{Unspecified} \tag{3.4}$$

Definition 8 (Permission)**.** A *Permission* represents a legal capability (or lack thereof) in the language. It is a tuple of a Sentence, an Allowance, and a Circumstance Set (which may be empty). The Circumstance Set of an Allow-Permission specifies in what circumstances it applies, whereas for a Forbid-Permission it specifies in what circumstances it does not apply.

$$\texttt{Allow-Permission} = \texttt{(Sentence, Allowance, CircumstanceSet)}$$
$$\texttt{Forbid-Permission} = \texttt{(Sentence, Allowance, CircumstanceSet)}$$

A requirement is an important abstraction and the first step towards defining Compliance. Confis assumes a that providing a legal obligation entails providing the legal capability to fulfill that obligation, which means that when we have a Requirement we have an `Allow` Permission too.

Definition 9 (Requirement)**.** A *Requirement* represents a legal obligation and is a tuple of a Sentence and a Circumstance Set (which may be empty).

```
Requirement = (Sentence, CircumstanceSet)
```

A requirement r entails an allow-permission p where if $r = (s, C)$ then $p = (\texttt{Allow}, s, C)$

Definition 10 (Event)**.** An *Event* is a tuple of a Sentence and a Circumstance set, describing the context the Sentence happens in, like `Event = (Sentence, CircumstanceSet)`

We use a set of Events to represent a 'state-of-the-world'. Note how we have no concept of precedence or ordering between events. The only way to order them chronologically would be with respect to a time-based Circumstance present in each event's Circumstance Set.

Definition 11 (Compliance)**.** We say that a Requirement r *has been met by* a set of Events W if and only if there is an event $e \in W$ such that they have the same Sentence and the Circumstance Set of e is generalised by the Circumstance Set of r.

We say that a `Forbid` Permission p *is respected by* a set of Events W if and only if there does not exist $e \in W$ such that e has the same Sentence as p and the Circumstance Set of e overlaps with the Circumstance Set of p.

A set of Events W *is compliant with* an agreement if and only if, for every Requirement r and every `Forbid` Permission p that are part of the agreement, r has been met by W and p is respected by W.

For the meanings of 'generalises' and 'overlaps with', see Definition 14 and Definition 15.

3.2.1 Circumstance and Circumstance Set

Circumstances are a core abstraction in Confis, both in the language and in Rule Generation and Contradiction Detection (hence why they deserve their own section).

A Circumstance Set represents the context in which a Sentence may occur. In the context of a legal agreement, it would contain everything in a clause that is not the core sentence, including the parent clause, conditions, etc.

A Circumstance Set is made up of Circumstances.

Definition 12 (Circumstance)**.** c is a *Circumstance* if and only if it is equipped with the following operations for any other circumstance c':

$$\texttt{generalise} : c \rightarrow c' \rightarrow \texttt{Boolean} \tag{3.5}$$

$$\texttt{overlapsWith} : c \rightarrow c' \rightarrow \texttt{Boolean} \tag{3.6}$$

$$\texttt{provideKey} : c \rightarrow k \tag{3.7}$$

$$\texttt{render} : c \rightarrow \texttt{String} \tag{3.8}$$

A circumstance key k is simply an element with a notion of equality. When two circumstances c_1 and c_2 produce they same key k, they are said to be *of the same type*. `render` is a function made to satisfy the requirement of being able to convert a Confis agreement into readable English for the sake of accessibility, and necessary to display query results in natural language.

A Circumstance Set is an indexed set of Circumstances, unique in that only one circumstance of each type can be stored. For example, we can only store a single circumstance representing time and a single circumstance representing purpose, because these are implemented to produce the same key k each.

Definition 13 (Circumstance Set)**.** A *Circumstance Set* is a key-value mapping of Circumstance Keys to Circumstance, like $\texttt{CircumstanceSet} = \{k \to \texttt{Circumstance}\}$.

Defining a `generalise` equivalent for circumstances will be of use when trying to define rules (see Section 4.2) because it will allow checking whether a circumstance from a clause 'matches' that of a question (see Intuition Behind Circumstances for an explanation).

Definition 14 (Circumstance Set Generalisation)**.** Fot two Circumstance Sets C_1 and C_2 we say C_1 *generalises* C_2 if and only if:

$$\forall k\,\forall c_1.[\,(k, c_1) \in C_1 \to \exists c_2.((k, c_2) \in C_2 \wedge\ c_1\ \texttt{generalises}\ c_2)\,] \tag{3.9}$$

That is, if for all circumstances in C_1, there is a circumstance of the same type in C_2 and each circumstance in C_1 generalises the circumstance of the same type in C_2.

Definition 15 (Circumstance Set Overlap)**.** For any two Circumstance Sets C_1 and C_2, we say C_1 *overlaps with* C_1 if and only if

$$\forall k\,\forall c_1.[\,(k, c_1) \in C_1 \to \exists c_2.((k, c_2) \in C_2 \wedge\ c_1\ \texttt{overlapsWith}\ c_2)\,] \tag{3.10}$$

That is, if for all circumstances in C_1, there is a circumstance of the same type in C_2 and each circumstance in C_1 overlaps with the circumstance of the same type in C_2.

This definitions of Circumstance Set generalisation and overlap have the following consequences:

- The empty circumstance set generalises all circumstance sets
- The empty circumstance set overlaps with all circumstance sets

Although nothing requires Circumstance Set overlap to be commutative, it will be commutative if the Circumstances it contains have a commutative implementation of `overlapsWith`. This is the case for every Circumstance built into Confis so far.

Intuition Behind Circumstances

Circumstance Sets can be thought of as contexts for a sentence, which together constitute an event. Defining overlaps and generalisations between these contexts allows us to reason about the same concepts for events: is *'Pay Bob during the month of June'* more general than *'Pay Bob the 10th of June'*? Defining a time range (a day in June, a month within a year) as a Circumstance allows us to answer those questions programmatically.

The empty circumstance set can be thought of as the least specific context for an event (think of it as *anywhere* when thinking about a place). Indeed, the event *'Alice pays Bob'* generalises the event *'Alice pays Bob during June'*. If a contract has a requirement for the former, and we supply it with a state of the world containing the latter, we can say that Alice is compliant with the contract!

Likewise, the converse is true for *disjoint* (not overlapping) events. For two events e_1 = *'Alice pays Bob during June'*, e_2 = *'Alice pays Bob during September'*, we can say the Circumstance Sets of e_1 and e_2 are disjoint, therefore if a contract requires e_1 but we have a state of the world with e_2, we can say Alice still has some legal obligation she has not fulfilled.

Generally, Circumstance Sets describe more specific situations the more elements they have. How different Circumstance Sets can overlap and generalise each other is shown by example through the definitions of the sets A, B, C, D and E below.

$$A = \{\texttt{time} \rightarrow \text{'10th of June 2022 - 12th of June 2022'}\} \tag{3.11}$$
$$B = \{\texttt{time} \rightarrow \text{'1st of June 2022 - 31st of June 2022'}\} \tag{3.12}$$
$$C = \{\texttt{purpose} \rightarrow \text{'Commercial'}\} \tag{3.13}$$
$$D = A \cup C \tag{3.14}$$
$$E = \{\texttt{time} \rightarrow \text{'1st of June 2022 - 11th of June 2022'}\} \tag{3.15}$$

We have:

- The empty Circumstance Set $\emptyset$ generalises and overlaps with A, B, C, D and E
- B generalises A and E
- A overlaps with B, D and E
- B overlaps with A, D and E.
- D overlaps with A, B, C, and E.
- E overlaps with B, D and A
- C overlaps with D

3.2.2 DSL Implementation

This subsection is concerned with the implementation details of the Confis DSL. For design decisions regarding the language (like what is a Sentence or the difference between a Requirement and a Capability) please see Language Semantics and Design.

Because of the nature of internal DSLs (see Section 2.4), the syntax of the Confis DSL must be a subset of the Kotlin language's. Given the Confis DSL and the Confis IR are decoupled this section will not go into too much detail on how the DSL is implemented as most of that is specific to the Kotlin language, and the DSL could have been implemented as an internal DSL with a different host language anyway (like Haskell or Groovy). Therefore, it will just show an example of how a Permission is implemented. Understanding how the Confis DSL is plain valid Kotlin will help understand how the editor (Section 3.4) works (and why it works well).

For more information on how DSLs can be built in Kotlin in general, see Section 2.4.1, and in particular refer to [54].

Permission Builder Implementation

Recall a Permission is a tuple of (Allowance, Sentence, CircumstanceSet), where the CircumstanceSet may be empty. For the sake of brevity, we will leave the construction of the Circumstance Set out and discuss how we can construct a tuple of (Allowance, Sentence) in the DSL. Recall a Sentence is simply a tuple of (Subject, Action, Object).

We first need a DSL to construct this Sentence. We use an infix extension function in order to add the 'may' adverb (which corresponds to Allowance = Allow) between the subject and the action of the sentence. We also use a Kotlin operator function [66] in order to overload the Action, so the user is able to write action(object) as opposed to ActionObj(action, obj). The full example is demonstrated in Listing 3.3, and a usage example is given in Listing 3.4 – which reads like an English sentence.

```
// Sentence tuple declaration
data class Sentence(
    val subject: Subject, val action: Action, val obj: Object,
)
// <Action, Object> tuple declaration
data class ActionObject(val action: Action, val obj: Object)

// this easily builds an ActionObject
// action.invoke(obj) == action(obj)
operator fun Action.invoke(obj: Object) = ActionObj(this, obj)

infix fun Subject.may(s: ActionObject) {
    val permission = Permission(Allow, Sentence(this, s.action, s.obj))
    clauses += permission
}
```

LISTING 3.3: Finished Permission builder

```
val alice: Subject
val eat: Action
val cake: Object

alice may eat(cake)
```

LISTING 3.4: Example of the DSL invocation of the final Permission builder of Listing 3.3

We now can call the function may() in order to create a Permission and add it to clauses. Most of the Confis DSL is implemented similarly. An advantage of using a statically typed language for a DSL with well-defined types like in the examples given above means that only semantically correct sentences can be written. For example, 'cake may eat(alice)' produces a compilation error, because 'cake' is not a Subject.

3.3 Additional Tooling: Legal Prose Rendering

In order to comply with the accessibility requirement, it should be possible to generate a traditional document (perhaps in plaintext or PDF format) in natural language. This generated document should then be usable in place of traditional agreement documents –

for example, a reviewer of an agreement will not need Confis software or to know about the Confis DSL in order to read a copy of the contract. We will call this feature ***natural language rendering***. If NLP (see Section 2.3) aims to translate natural language into a machine-readable format, you can think of natural language rendering as the inverse process: we go from a machine-readable representation to a more human-readable one.

This is implemented with advanced templating (a rendered Permission includes a rendered Sentence, for example) that transforms IR elements into markdown code. Markdown can then be conveniently converted to HTML (for rendering inside the editor, see Figure 3.3) or to PDF [23].

3.4 Additional Editing Tooling: the Confis Editor

One of the key requirements for Confis is accessibility. In order to achieve this the Confis framework includes an extension to the IntelliJ Development Environment [55] - the ***Confis Plugin***. Bundled with this plugin is the ***Confis Editor***. The Confis Editor has two main goals:

Providing context-aware aid for authoring agreements Much like an IDE provides relevant aid to a software engineer (such as reporting compile errors before attempting to compile the program) the Confis Editor should allow an agreement drafter to write in the Confis DSL without needing to deal with command-line utilities or knowing the tooling around Kotlin language.

Providing a human-readable preview of the Confis Agreement The Confis Editor should display a document in plain English, made up from text generated from the Confis agreement (thanks to natural language rendering). This preview should resemble a traditional legal document, and appear side-to-side the code the agreement drafter is writing in order to provide a useful live preview of the contract being authored (much like it is handy to have a PDF preview when drafting TeX documents, an example of which can be found in Figure B.1).

3.4.1 Editor Implementation

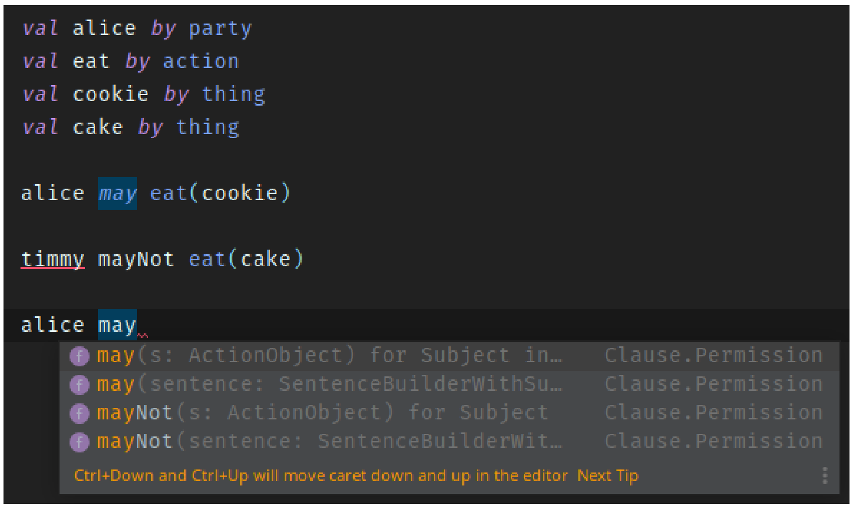
Two things are required in order to be able to draft single Confis code files inside an IDE:

1. It must be possible to write a self-contained, boilerplate-free contract in a single file.
2. The IDE must perform reporting specific to the Confis DSL when editing this file (as opposed to reporting only on the host language, Kotlin).

(1.) is where Kotlin's custom scripting support [57] is crucial. See Scripting Support for a discussion on what is meant by scripting and how it is required for making a stand-alone DSL.

In order to implement writing Confis agreements as stand-alone files, we create a *Kotlin Custom Script definition* [57], which essentially contains information that assigns an instance of a class to a script. By placing DSL builder functions like those introduced in Listing 3.3 inside a class (called `AgreementBuilder`, for example) and assigning an instance of this class to a script, we can let the script author (ie, the Confis agreement author) **mutate** our class when we evaluate their script. See Listing A.1 for an example of such a class, and Listing A.2 for an example of a single script bound to that class.

We can then pass the script definition mentioned above to an extension point present in the IntelliJ API engineered specifically for this purpose [59, 55]. If we include some additional information (such as a file extension for Confis agreements) IntelliJ will now provide visual hints for Confis Scripts.



```
val alice by party
val eat by action
val cookie by thing
val cake by thing

alice may eat(cookie)

timmy mayNot eat(cake)

alice may
    may(s: ActionObject) for Subject in…     Clause.Permission
    may(sentence: SentenceBuilderWithSu…     Clause.Permission
    mayNot(s: ActionObject) for Subject      Clause.Permission
    mayNot(sentence: SentenceBuilderWit…     Clause.Permission
    Ctrl+Down and Ctrl+Up will move caret down and up in the editor  Next Tip
```

FIGURE 3.2: Syntax highlighting in a small Confis agreement

See Figure 3.2 for an example where the Confis editor reports a compile error (where the user attempted to form a Sentence with a Party that does not exist) as well as autocompletion for building a Permission clause.

In order to achieve (2.), we can implement a new editor in IntelliJ based on its existing markdown editor [67] which achieves a very similar purpose: providing a live preview of machine-readable code. The live preview is thus implemented as a preview of an in-memory Markdown document which is in turn generated upon code changes in the editor:

Confis Code → Confis IR → Markdown document → HTML page → Live preview

A screenshot of the Confis Editor, including syntax highlighting and the live preview, can be found in Figure 3.3. For the sake of brevity, this report will omit the implementation details of the live preview – but a starting point can be found in the software archive under the `ConfisEditor` class[1].

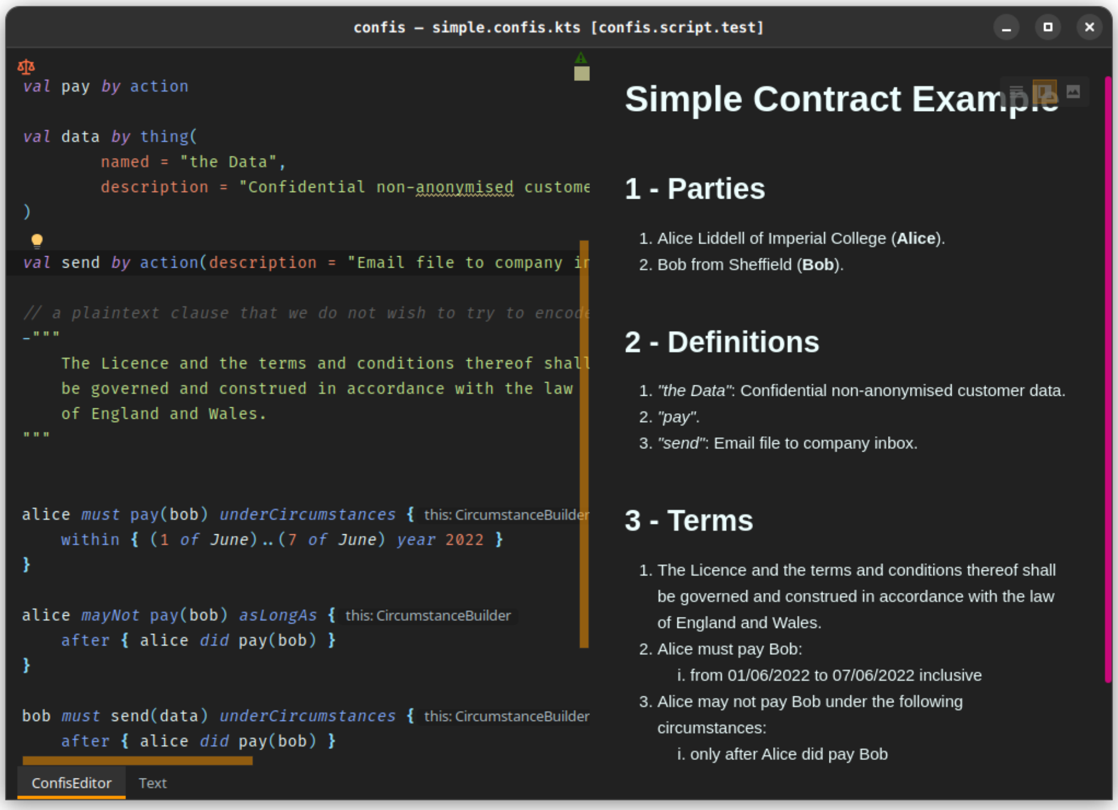
3.5 Language Documentation

In the spirit of software engineering principles and to further work towards meeting the accessibility requirement, this project also introduces a documentation website, which can be found at `confis.dcotta.eu` and is pictured in Figure 3.4. It is implemented as a series of Markdown documents rendered as HTML pages thanks to the MkDocs [68] tool.

The documentation's main aim is to clearly explain Section 3.2 without getting into implementation details, nor the specifics of how Circumstances are formalised.

[1]The `ConfisEditor` class can be found int he software archive under

- `plugin/src/main/kotlin/eu/dcotta/confis/plugin/ConfisEditor.kt`



```
val pay by action

val data by thing(
        named = "the Data",
        description = "Confidential non-anonymised custome
)

val send by action(description = "Email file to company ir

// a plaintext clause that we do not wish to try to encode
-"""
    The Licence and the terms and conditions thereof shall
    be governed and construed in accordance with the law
    of England and Wales.
"""


alice must pay(bob) underCircumstances { this: CircumstanceBuilder
    within { (1 of June)..(7 of June) year 2022 }
}

alice mayNot pay(bob) asLongAs { this: CircumstanceBuilder
    after { alice did pay(bob) }
}

bob must send(data) underCircumstances { this: CircumstanceBuilder
    after { alice did pay(bob) }
```

Simple Contract Example

1 - Parties

1. Alice Liddell of Imperial College (**Alice**).
2. Bob from Sheffield (**Bob**).

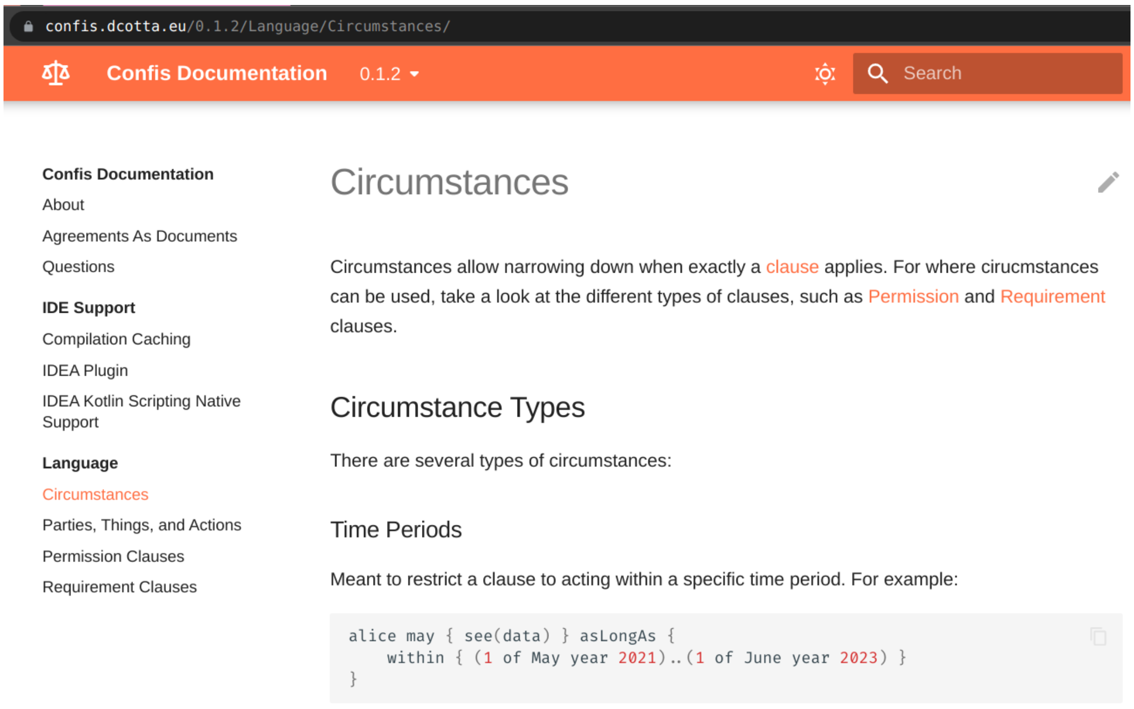
2 - Definitions

1. *"the Data"*: Confidential non-anonymised customer data.
2. *"pay"*.
3. *"send"*: Email file to company inbox.

3 - Terms

1. The Licence and the terms and conditions thereof shall be governed and construed in accordance with the law of England and Wales.
2. Alice must pay Bob:
   i. from 01/06/2022 to 07/06/2022 inclusive
3. Alice may not pay Bob under the following circumstances:
   i. only after Alice did pay Bob

FIGURE 3.3: Screenshot of the Confis Editor, including the natural language preview



Circumstances

Circumstances allow narrowing down when exactly a clause applies. For where cirucmstances can be used, take a look at the different types of clauses, such as Permission and Requirement clauses.

Circumstance Types

There are several types of circumstances:

Time Periods

Meant to restrict a clause to acting within a specific time period. For example:

```
alice may { see(data) } asLongAs {
    within { (1 of May year 2021)..(1 of June year 2023) }
}
```

FIGURE 3.4: Screenshot of the Circumstances section of the Confis online documentation

Chapter 4

Queryable Documents

4.1 Developing a suitable representation

4.1.1 Requirements

Having introduced a formalism for a contract in Chapter 3, we can now leverage this machine-readable abstraction in order to analyse the document. We have chosen to perform this analysis as queries (or questions) made to the contract. By consulting with experts in the field of law, we have also determined that the most desirable processing is the following:

Querying for Legal Capabilities A common use-case is for a party to want to figure out their legal capabilities with respect to a contract, as well as the capabilities of other parties. Take the example of a tenancy agreement: a tenant may want to know whether they are allowed to have pets, or under what circumstances their landlord is allowed to enter the property. Therefore, a successful query system should be able to provide answers to questions such as *'May A do X?'*

Compliance verification If figuring out a party's legal capabilities is a key part of dealing with a contract, so is figuring out their legal obligations. Unlike a legal capability question, a compliance question cannot be formulated as *'May A do X?'* – they should instead be along the lines of *'What does X need to do in order to be compliant?'*. We should also take into account that a party may have already done something to be compliant at the time of performing the query – therefore we need to include some 'state-of-the-world' in our question. Additionally, parties usually interact and actions from more than a single party may be needed to achieve compliance.

4.1.2 Confis Internal Representation

As [36] notes, there is a clear compromise to be made between how complex a contract representation is, and how simple the computations needed to process it are. Contrary to solutions discussed in Section 2.3 such as [43], Confis does not try to generalise a contract into a set of normative rules (as defined in Equation 2.1) straight away. Instead, it tries to preserve all the information that goes into assembling a contract into an Intermediate Representation (that we will call ***Confis IR***). The Confis IR is then converted into different sets of rules depending on the query being performed.

The Confis IR is a set of data structures, mostly tuples and collections, that can be mapped to a JSON or protobuf schema for serialisation. For the sake of brevity, this report does not contain the entire schema, but see an example of a compiled IR at Listing A.4.

4.2 Rule Generation and Contradiction Detection

Confis produces an Internal Representation (IR) from compiling the language, and from there it is able to answer complex queries. It does this by generating normative rules from each clause. The nature of the rules depends on the type of query. Confis supports three different types of queries:

Allowance Questions Their input is an Event (ie, a Sentence S and a Circumstance Set C) and its output is an Allowance. It is meant to represent the question 'What is the allowance of S under the circumstances C'? An example of such a question is *'May I use this dataset for commercial purposes?'*

$$Q_{\texttt{allowance}} : \texttt{Event} \rightarrow \texttt{Allowance}$$

Circumstance Questions Their input is a Sentence S, and the output is two Circumstance Sets, C_{allow} and C_{unless}, or a set of a set of clauses $A_{\text{contradictory}}$. The query is meant to represent the question *'Under what circumstances may S happen?'*. C_{allow} represents the Circumstance Set which answers this question (if it is empty, then S is Forbidden) with the exception of the Circumstances present in C_{unless}. If the contract contains contradictions between clauses, then $A_{\text{contradictory}}$ is returned: it represents the sets of clauses that contradict each other. This mechanism is what allows contradiction detection to aid the contract drafter to produce meaningful contracts.

$$Q_{\texttt{circumstance}} : \\ \texttt{Sentence} \rightarrow \big(\,(\texttt{CircumstanceSet, CircumstanceSet}) \mid \texttt{Set<Set<Clause>>}\,\big)$$

Compliance Questions Their input is a set of Events (meant to represent a state-of-the-world) and the output is another set of Events of what needs to happen in order to become compliant, or a set of clauses that have been not been complied with (see Definition 11) if any.

$$Q_{\texttt{compliance}} : \texttt{Set<Event>} \rightarrow \big(\,\texttt{Set<Event>} \mid \texttt{Set<Clause>}\,\big)$$

Confis helps reason about legal agreements chiefly by accurately answering these types of queries. Therefore generating rules from the IR is a critical part of the project.

There are several types of clauses (see Section 3.2) and each clause generates between one and four rules depending on the query type and its Allowance. We will examine the example of converting a Permission clause with a non-empty Circumstance Set. It is one of the simpler cases, and it generates two rules. Take the clause specified by Listing 4.1 (the full contract can be found in Figure A.1):

```
alice may use(data) asLongAs {
    within { (1 of June)..(30 of June) year 2022 }
}
```

LISTING 4.1: Clause with a Circumstance – extract from Figure A.1

The IR generated by this agreement can be found in Listing A.4.

We will now generate the rules for an *allowance question* for this rule (ie, a query for legal capabilities). As explained in the previous Subsection 4.1.1, the question can be formulated as *'Given circumstances C, may Alice use the data?'*. The inputs are a Sentence and a Circumstance Set, while the output is a set of Circumstance Sets (we require a Circumstance Set for each of the scenarios where Alice may use the data). The resulting rules would

1. Check whether it corresponds to a clause that matches the given question.
2. Check whether the circumstances of the question are a specific case of those of the clause (in order to report a positive result).
3. Check whether the question is not specific enough to be covered by the clause but concerns the same circumstances (in order to report an ambiguous result).

Those rules are given by the pseudocode in Listing 4.2:

```
AllowanceRule(
    // predicate: (Permission, Question, Result) -> Boolean
    when = { p, q, _ ->
        p.sentence == q.sentence &&
        p.circumstanceSet generalises q.circumstanceSet
    }
    // side-effect: (Permission, Result) -> Void
    then = { p, r -> result = Allow }
),
// question general enough to concern us but not narrow enough to match clause
AllowanceRule(
    when = { p, q, _ ->
        p.sentence == q.sentence &&
        !(p.circumstanceSet generalises q.circumstanceSet)  &&
        p.circumstances overlapsWith q.circumstances
    },
    then = { if (result != Allow) result = Depends },
)
```

LISTING 4.2: Rules generated from the clause given in Listing 4.1

Rules can be combined this way in order to construct the Result. Because the rules engine Confis uses allows continuous rule evaluation (see Subsection 2.6.1) we can add more rules that compare the rules generated by different clauses in order to detect contradictions in the clauses. This allows detecting malformed contracts by detecting contradictions that, while they are semantically and syntactically correct, make little sense.

An example for such an agreement can be found in Listing 4.3, and a screenshot of a detected contradiction can be found in Figure B.4. We are able to detect such contradictions thanks to the `overlapsWith` function for Circumstance Sets, which is discussed in Subsection 3.2.1.

4.3 Tooling for Accessible Querying: the Query UI

A core distinctive capability of Confis is its accessibility – a contract drafter should not need to use a command-line tool or an HTTP API in order to perform a query. In order to meet this requirement, the Confis Plugin (which is an extension to the IntelliJ IDE [55, 59]) provides a graphical user interface to construct and display questions.

```
alice mayNot use(data) asLongAs {
    val year2022 = (1 of January)..(31 of December) year 2022
    within { year2022 }
}
alice may use(data) asLongAs {
    within { (1 of June)..(30 of June) year 2022 }
}
```

LISTING 4.3: A syntactically correct Confis agreement that contains detectable contradictions

The user experience is as follows: the contract drafter writes their agreement into the Confis Editor (see Section 3.4) and, on the side, they have a window that allows performing queries on the contract they are writing. This allows them to ask questions while writing, and check whether their answers are within the drafter's expectations.

4.3.1 Query UI Implementaiton

Implementing a UI to build queries involves:

- Constructing a Sentence through a UI
- Constructing a Circumstance Set through a UI
- (therefore) Constructing a Circumstance through a UI
- Compiling and generating rules from the agreement in the Confis Editor
- Displaying the result from the rules' evaluation in an English-readable fashion

Out of all of these, constructing a Circumstance proved to be an interesting challenge. This section will go into more detail about how this was done as an example of overcoming some of the technical difficulty in developing the prototype.

At the time of writing Confis has four different types of Circumstance, and is designed to be able to accommodate any number. It would not have been desirable to require designing a UI for each new Circumstance. According to [47] designing a user interface to assemble queries with the concepts described in Section 3.2 is much like building a new DSL, except it is a visual one instead of a textual one. The solution to the difficulty of making a UI for assembling Circumstances proved to be in combining the textual DSL that is a Confis agreement, and the visual one that is the query UI. The end result allows building questions through a UI without having to require a command-line interface or calling a library from code, and permits writing Circumstances as code just like they would be written in an agreement.

Allowing the user to type in Circumstances in a text box involves creating a synthetic Abstract Syntax Tree (AST) of the Kotlin language (which Confis is based on, see Subsection 3.2.2 for more details) which includes the necessary imports for the DSL, and then compiling this language fragment in order to instantiate a Circumstance Set in memory. This synthetic Circumstance Set can then be passed to the query engine for rule generation.

While with this the goal of assembling a Circumstance has been achieved, the Circumstance editor in the query UI would be far from being on-par with the main Confis Editor (discussed in Section 3.4, it provides compiler error reporting, autocompletion, and syntax highlighting). We decided to make intelligent editing inside the Circumstance editor a priority, because the only other way to inform the user about malformed circumstance code would be to expose them to the Kotlin compiler logs. In order to achieve this, Confis reuses the open-source implementation of IntelliJ's Java expression evaluator [55] (meant to be used inside debugging sessions). It parses the Circumstance editor and constructs a new AST which is then instantiated in a new program.

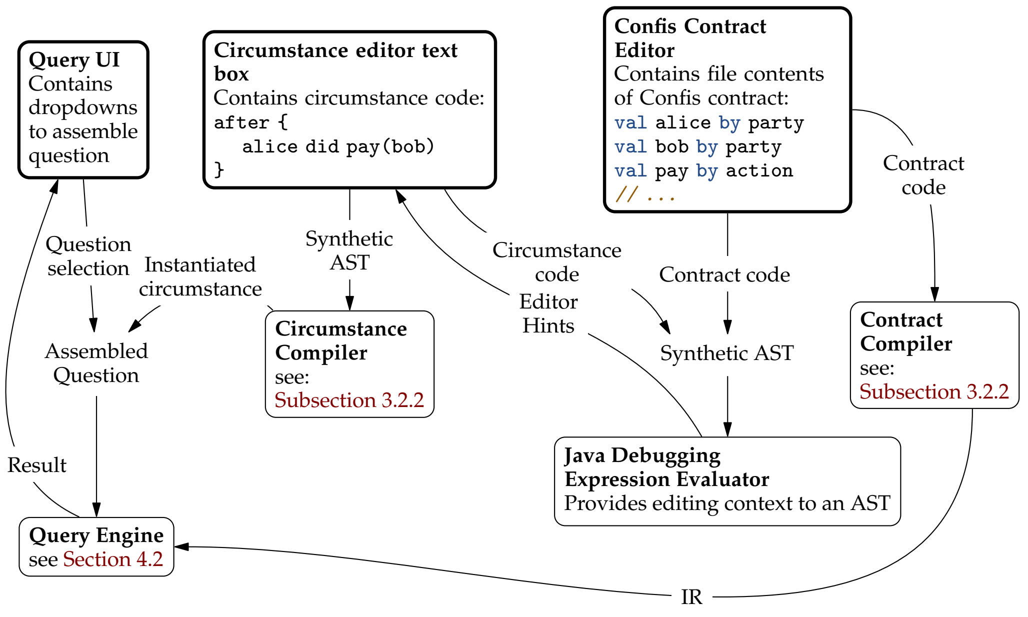


FIGURE 4.1: Diagram showing the architecture behind the Query UI's Circumstance editor
Elements the user interacts with are highlighted with a thicker outline

The program is then fed to the debugging evaluator, which is then able to provide relevant editing hints to the user. Upon changes of the Confis script or the Circumstance test box content, the synthetic AST is reconstructed so the editor can provide up-to-date aid.

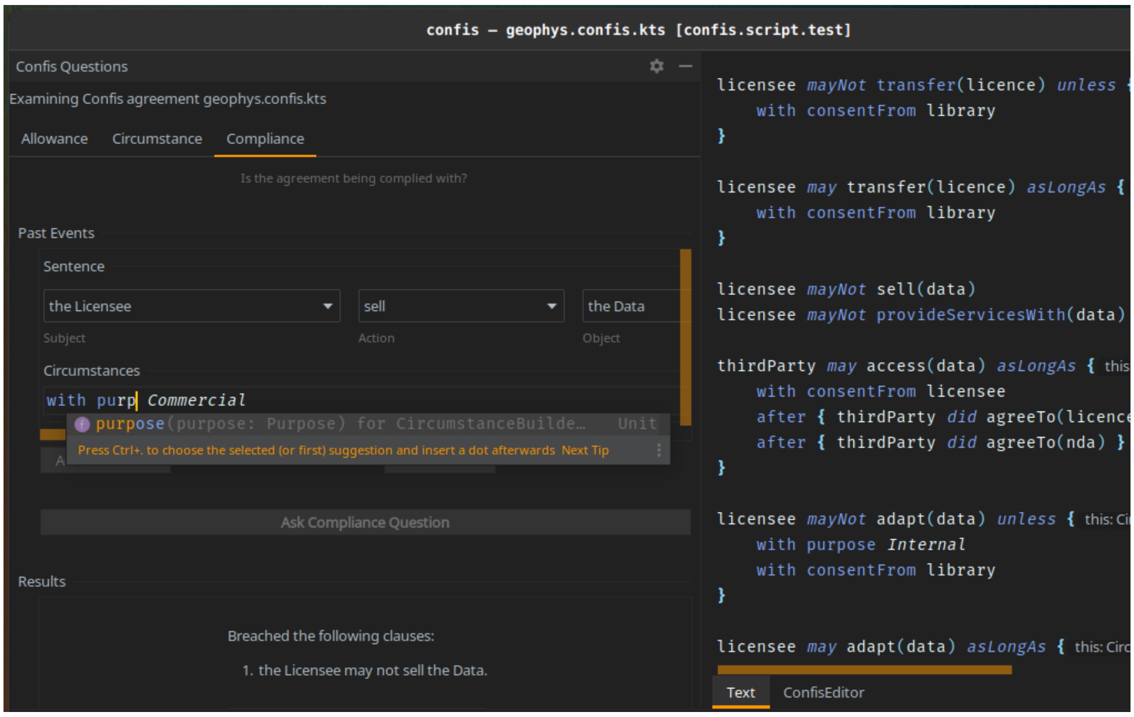
This end result is a smart Circumstance editor on par with the normal contract editor, able to report compilation errors and aware of the symbols present in the contract. This is shown in Figure 4.2, and a diagram of the architecture can be seen in Figure 4.1.

More screenshots can be found in the appendix under Section B.2, demonstrating an Allowance question and a compile error in the Circumstances editor text box.

4.4 Confis as a Generalisation of Ricardian Contracts

Ricardian Contracts [32] (discussed in Subsection 2.2.5) are one of the few technologies in the state of the art discussed in the Background chapter and allow querying a contract and are concerned with accessibility at all (they specify contract semantics as natural language).

Confis tries to capture the meaning of the agreement it encodes. For example, a party should be able to query about their compliance with the contract without needing to know whether the contract is a tenancy agreement, a bond, or a license. A Ricardian contract, on the other hand, is closer to an instance of a specific kind of contract – therefore



Examining Confis agreement geophys.confis.kts

Is the agreement being complied with?

the Licensee | sell | the Data

```
with purp Commercial
purpose(purpose: Purpose) for CircumstanceBuilde…   Unit
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards  Next Tip
```

Breached the following clauses:

1. the Licensee may not sell the Data.

```
licensee mayNot transfer(licence) unless {
    with consentFrom library
}

licensee may transfer(licence) asLongAs {
    with consentFrom library
}

licensee mayNot sell(data)
licensee mayNot provideServicesWith(data)

thirdParty may access(data) asLongAs { this
    with consentFrom licensee
    after { thirdParty did agreeTo(licence
    after { thirdParty did agreeTo(nda) }
}

licensee mayNot adapt(data) unless { this: Ci
    with purpose Internal
    with consentFrom library
}

licensee may adapt(data) asLongAs { this: Circ
```

FIGURE 4.2: Query UI window for agreement given in Listing A.3
Note how the question is built by selecting the *Compliance* tab and assembling the Sentence *'the Licensee sell the Data'* from dropdown menus. Circumstances are built through the text box, featuring autocompletion and compliance checking

knowledge about the contract 'type', or 'template', are necessary to extract the metadata encoded in it.

This project is a generalisation of Ricardian contracts in that Confis agreements meet all the requirements given by definition of Ricardian Contract[1], but it also tries to convey and capture the meaning of the contract, so that both a machine and a human can take every single possible state into account.

Another crucial difference from what was first specified by Grigg in [32] is that Confis does not intend to let a user write legal prose. It instead produces machine-generated but human-readable prose legal prose from its internal encoding of the agreement – see Section 3.3 for a discussion of this feature.

[1]It is worth noting that while Confis as a formalism does meet the requirements given by the definition of a Ricardian Contract, the artefacts included in the project do not implement the public-key signing infrastructure (discussed in Subsection C.1.2).

Chapter 5

Ethical Considerations

Possible Discrepancies Between Machine-Readable Representations and Legal Requirements This project provides new technology to understand and reason about a legal contract. When it is applied in industry, there is always a risk that the contract the drafter means to represent (traditionally written as plain natural language) is not aligned with the contract represented by the Confis specification they actually write. This can stem primarily from misunderstandings of the limitations of what Confis can represent (these are discussed in Subsection 6.1.1).

Liability Additionally, it is possible that instead of being used to draft new contracts, this project is used to translate existing contracts into a formalism. This creates the additional risk that a party may be subject to a natural language contract while checking their legal capabilities and obligations within the scope of the Confis agreement. If these two different representations have discrepancies, then such a Party might inadvertently breach a clause in the original natural language representation.

It is the opinion of this project that the liability of this sort of legal blunder should lie with the party that agreed to be subject to a natural language contract but did not follow legal advice concerning that representation of the contract. This problematic scenario is one of the reasons Confis makes an emphasis on authoring contracts, rather than translating existing agreements into a formalism.

Legal Validity If this project wants to represent or encode real legal agreements, the representations should qualify as such within the legal system they intend to be used within.

Ideally, a court should be able to recognise a Confis representation as well as a traditional natural language one. Parties using Confis agreements should double-check their natural language equivalents qualify as contracts within their legal jurisdiction. The background research this project has undertaken has verified this is commonly the case for Common Law jurisdictions (which includes the United Kingdom) [1, 6, 7].

Copyright The prototype for this project makes use of several licensed, open-source software libraries. The licenses used are:

- **Apache License** Used by the Kotlin Language [53], IntelliJ IDEA [55] and Gradle [52].
- **MIT License** Used by Easy Rules [63].
- **MSD 2-Clause Simplified License** used by MkDocs [68]

All of these licenses allow the distribution of modified and larger works under different licenses. Confis includes copyright and license notices for all the above licenses, therefore it is compliant with all the licenses it has been granted.

Misuse This project does not envision malicious use of the technology it introduces. It is also not particularly vulnerable to remote attack, since data is not transmitted through any networks.

Data Privacy and Compliance Legal agreements between businesses can be confidential both in their contents and their existence – this project preserves these properties in that it does not upload or process any Confis document. Contracts are drafted with an offline editor, and the software prototype process files offline. Rules evaluation results are also not uploaded to the internet.

Therefore, Confis is also General Data Protection Regulation (GDPR) [69] compliant.

Dual Use This project does have military applications: automating workflows around legal agreements could improve supply chains, including military ones; but its focus and motivations are exclusively civilian.

Therefore, this project does fall under the definition of Dual Use.

Environmental Implications This project does not have major environmental implications.

Chapter 6

Evaluation

6.1 Language Formalism Evaluation

The Confis language formalism and semantics are discussed in Section 3.2. We will evaluate the language in how it meets the requirements described in Section 1.2, its expressiveness (how capable it is to represent complex contracts) as well as how it performs in relation to comparable formalisms (like Symboleo [4] or The Accord Project [3]). Due to the nature of legal agreements (and the impossibility of formalising every aspect of existing legal texts) this assessment is a qualitative one.

As for as fulfilling the accessibility requirements, Confis is one of the few formalisms in the literature that makes an effort to penetrate industry by

- Making few assumptions about the user's background (including knowledge such as event fluidity, event calculus, or first order predicate logic).
- Choosing language constructs to purposefully resemble natural language.
- Providing additional tools to ease development and shorten the iteration loop.

The Query UI (Section 4.3), Confis-to-English conversion (Section 3.3), the Confis Editor (Section 3.4) and the structure of the Confis language itself (Section 3.2) were all developed with this goal in mind.

Exceptions to this statement include accessible technologies like Juro (Subsection 2.2.2). Such tools add contract metadata without actually allowing for querying beyond fetching the metadata, nor attempt to capture the semantics of the agreement – like Adobe Signing tools or typical Ricardian Contracts [33], but unlike Symboleo or The Accord Project.

We translate to Confis a sample contract used in Symbolio [4] in order to compare the same agreement in two different languages. For the sake of brevity, we wille examine an extract (given in Table 6.1), but the full original (in plain English) can be found at Table A.1, the full Symbolio specification at Listing A.6, and the full Confis agreement at Listing A.5. The translated extracts are both re-written self-contained examples (rather than text extracts from the original, longer contracts). This is in order to fully reflect the necessary boilerplate and ceremony of each language.

Specifying a legal Obligation Notice how Symboleo allows representing a more complex domain by specifying an *Event* `'Disclosed'`, and constraining the legal capabilities of Buyer by creating an *Obligation* (with a notion of this obligation being *from* Buyer *towards* Seller) which specifies that the `disclosed` event cannot happen before six months after the end of the contract. In Symbolio's model if `disclosed` happens, the breach cannot be attributed to either Seller nor Buyer.

Confidentiality 1. Both Seller and Buyer must keep the contents of this contract confidential during the execution of the contract and six months after the termination of the contract.

TABLE 6.1: Sample confidentiality clause, extracted from Table A.1

```
val effDate = 1 of June year 2022
val reveal by action(
    description = "as in not keeping the contents confidential"
)
val contract by thing("the Contract", description = "this Agreement")
val seller by party("the Seller", description = "Alice Liddell")
val buyer by party("the Buyer", description = "The Meat Supermarket, Inc")

seller mayNot reveal(contract) asLongAs {
    within { effDate..(effDate + 6.months) }
}

buyer mayNot reveal(contract) asLongAs {
    within { effDate..(effDate + 6.months) }
}
```

LISTING 6.1: Confis for Sample confidentiality clause, extracted from Listing A.5

Confis specifies a simpler domain – while it also specifies Seller and Buyer and represents 'disclosing' as an Action, it has no notion of *creditor* and *lender* when it comes to Requirements (defined in Definition 9). On the other hand, Confis can 'blame' specific parties for a breach, as it attributes Actions to Subjects. Confis is also unable to keep track of its own execution time – instead it requires specifying the date in the contract. Notice how this abstracts away the document from the real-world execution date, and how this limitation of Symboleo is omitted in the original paper [4], but present in the sample source code [70].

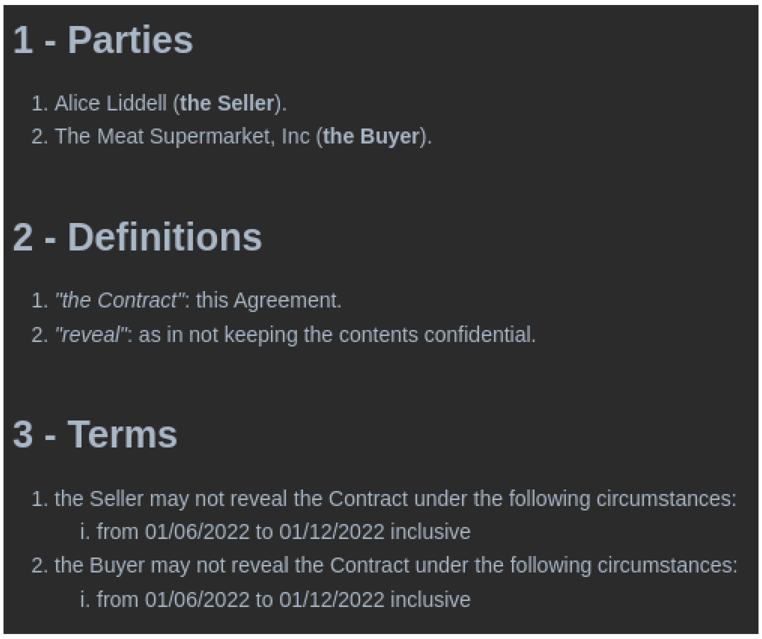
Both models are capable of expressing periods of time, but in Confis they are not part of the algebra (they are instead more general Circumstances).

Accessibility While both contracts require language knowledge to be written, Confis can be read without prior training and its operators (`within`, `mayNot`, `asLongAs`, ...) are easy to memorise. The best example of this is the operation of summing six months to a time period – while Symboleo requires specifying a `Date` library and wrapping the duration with a function ('`Date.add(..., 6, months)`'), Confis uses operator overloading to sum to a date ('`... + 6.months`').

As far as readability goes, Confis goes to further lengths to achieve it by trying to combine metadata and language semantics to provide a natural-language-like preview. This rendering is shown in Figure 6.1. While the preview is not as clear as the original plain-English clause, it conveys the obligations of each party well and is unambiguous thanks to its definition referencing (which is common in real legal agreements).

```
Domain meatSaleDomain
Seller isA Role with returnAddress: String, name: String;
Buyer isA Role with warehouse: String;
Disclosed isAn Event;

endDomain

Contract MeatSale (buyer: Buyer, seller: Seller, effDate: Date)

Declarations
disclosed: Disclosed;

Surviving Obligations
so1 : Obligation(seller, buyer, true,
    not WhappensBefore(disclosed, Date.add(Activated(self), 6, months))
);

so2 : Obligation(buyer, seller, true,
    not WhappensBefore(disclosed, Date.add(Activated(self), 6, months))
);
endContract
```

LISTING 6.2: Symboleo Specification for Sample confidentiality clause, extracted from Listing A.6


1 - Parties

1. Alice Liddell (**the Seller**).
2. The Meat Supermarket, Inc (**the Buyer**).

2 - Definitions

1. *"the Contract"*: this Agreement.
2. *"reveal"*: as in not keeping the contents confidential.

3 - Terms

1. the Seller may not reveal the Contract under the following circumstances:
   i. from 01/06/2022 to 01/12/2022 inclusive
2. the Buyer may not reveal the Contract under the following circumstances:
   i. from 01/06/2022 to 01/12/2022 inclusive

FIGURE 6.1: Confis prose rendering of Table 6.1

For a more detailed comparison of the specifications of this agreement in the Confis and Symboleo languages, please refer torAppendix A, in partcular Subsection A.2.1.

6.1.1 Confis Limitations in its Semantics

While the focus Confis makes in accessibility makes it easier to learn, read, and write; it comes at a price in the guarantees it brings and the expressiveness of the contracts that it can represent. This section provides examples of clauses in existing contracts that Confis struggles to formalise and tries to generalise the situations it cannot encode, as well as suggesting solutions or workarounds for said limitations.

Act Upon Violation

Contracts are sometimes defensively written (not unlike defensive programming software engineering practices) where they may include clauses that specify what should happen upon breach of other clauses (an example of such a clause is given in Table 6.2).

Payment & Delivery
• In the event of late payment of the amount owed due, the Buyer shall pay interests equal to <intRate> the Seller may suspend performance of all of its obligations under the agreement until payment of amounts due has been received in full.

TABLE 6.2: Sample breach clause, extracted from Table A.1

In Confis semantics, this represents a contradiction in the logic of the contract, because we have a rule specifying that a set of states is a breach, and then other rules specifying capabilities when in those states. This compromise between contradiction detection (extremely useful for helping the drafter produce well-formed contracts) and expressiveness (allowing these do-in-case-of-breach clauses) is unavoidable in the current semantics of Confis Allowance (see Allowance for more details).

In order to circumvent this limitation, it is possible to create two 'cases': a Permission that allows paying on time, and different Permission that allows paying at any time with interest. While this removes the contradiction, it does not capture the violation semantics of the original legal agreement.

Symboleo is capable of capturing such a scenario thanks to its `Happens(Violated(_))` predicate, which can create a new obligation when a different obligation is violated.

Confis Does Not Have Time in its Event Algebra

Symboleo (and other logic-based solutions like [36]) relate events through *happensBefore* partial orderings (introduced by [37]). They then can use these relations to model enforcing conditions, pre-conditions, and post-conditions over periods of time.

Confis only has a concept of time as a Circumstance – an event does not necessarily happen at a given time, the same way it does not necessarily happen with a given purpose. Ordering of events is established with an additional circumstance called a Past Action. Again, this Circumstance does not receive special treatment and like the others it is up to the author of the contract to specify it. While this design choice greatly simplifies rule generation and language semantics (thus making Confis easier to learn) it is a sacrifice in expressiveness.

For example, unlike Symboleo, Confis is unable to allow a new Capability only after an event E occurred at a time t and E has happened after another event E_{past}. Instead, Confis only allows a new capability after E and E_{past} have both happened, regardless of whether E happened at time t, or whether E happened before or after E_{past}.

It is possible to mitigate this issue by explicitly forbidding E_{past} to happen before E – but then we would be slightly altering the original contract's semantics by introducing a new clause breach scenario.

Confis Does Not Model Amounts

A key barrier to accessibility in existing formalisms – and even any general-purpose programming language – is their type systems. Other than the absolutely necessary abstractions for the domain (such as Parties or Actions) users need to know what a strictly positive integer or a floating point number are. The language then usually needs to define operations on these types, as well as extend them with other types such as Dates, Lists, etc.

Confis lowers the entry barrier by doing without most of this mental overhead (the only type concerned with numerical values it allows is a Date). It does not require using libraries in order to perform operations – mostly because there are no such operations, but partly thanks to operator overloading like in Listing 6.1.

This greatly limits its expressiveness in terms of numerical amounts and the computations that can be done on them during the contract evaluation. Consider The Accord Project [3] (see Subsection 2.2.3), which models programs computationally with a more general-purpose language. It is able to perform complex floating-point arithmetic in order to compute interest rates and fees, such as the one given in Table 6.3.

On the other hand, Symboleo (and similar logic-based technologies) struggle more to perform such calculations (possibly due to their declarative nature). In the original Symboleo paper [4] which contains a meat sales example hand-picked to showcase the language, the authors do not attempt to perform an interest calculation over time even though their original agreement (given in Table A.1) specifies one – this is shown in Listing 6.3 for convenience.

```
Domain
Currency isAn Enumeration(CAD, USD, EUR);
PaidLate isAn Event with
    amount: Number, currency: Currency, from: Buyer, to: Seller;

Contract MeatSale (
    buyer : Buyer,
    seller : Seller,
    curr : Currency,
    interestRate: Number
)

endDomain

Declarations
paidLate: PaidLate with
    % where interestRate is a constant
    amount := (1 + interestRate / Math.abs(2)),
    currency := curr,
    from := buyer,
    to := seller;
```

LISTING 6.3: Symboleo extract from Listing A.6 concerned with interest rates

In short, while Confis lacks the computational capabilities of a programming language, it is not lacking when compared to Symboleo since it does allow specifying numerical constants (except without having to specify their types). Additionally, traditional contracts also leave such computations to the reader.

Payment Terms
• [. . .] Payments made after the due date may be [. . .] subject to a late fee equal to the lesser of 1.5% per month or the maximum allowed by law.

TABLE 6.3: Clause concerning an interest rate, extracted from [9], a Licence Agreement by The Economist Group

Confis Circumstances Are Domain-Aware

A Circumstance is an instance of an object that implements the functions given in Definition 12. These operations vary greatly depending on the domain that the Circumstance is meant to represent. For example, for time-type Circumstances, the `overlapsWith` function checks for overlaps in time ranges; while in past-event-type Circumstances `overlapsWith` checks the union of the sets of past events. This makes Circumstances very intuitive to use in practice: if the contract specifies Alice must pay this week and she paid this Monday, she expects 'Monday' to be generalised by 'this week', without needing to learn about the abstractions behind Circumstances.

The price to be paid for this ease-of-use is that Circumstances for new domains must be implemented before they can be used. The most simple example is geographical locations. This Circumstance type is not already built into Confis. If we were to make such an extension to the language, we would need to define a location-type circumstance as some coordinates (or perhaps an address). The functions `generalises` and `overlapsWith` would compare coordinates in order to determine if one is a location included in the other (say, a house within a street), or if they are close enough to be the same place.

In short, making domains accessible at the language level requires less general abstractions that require domain knowledge at the implementation level. While Confis is built to be easily extensible implementation-wise, this is a rare limitation in specification languages, which usually aim to be as general as possible in order to cover all possible cases. The trade-off is once again in accessibility and expressiveness, and can be circumvented by encoding a Circumstance inside a Sentence's Action – following the previous example, rather than 'deliver', the Action would be rewritten as 'deliver to 10 Downing St.'.

6.1.2 Gravity of Language Semantics Limitations

This subsection aims to determine to what extent the limitations presented in Subsection 6.1.1 affect negatively the practical usability of Confis. Given how Confis is superior in its accessibility to comparable languages in the literature like Symboleo and The Accord Project (by virtue of the compromises it makes) we have focused on comparing real-world contracts with their Confis representations, including [9, 70, 11, 5].

The main discrepancies between the intended meaning of the traditional natural language agreements and their Confis counterparts concern re-evaluation of legal capabilities and obligations (ie, Confis' Permission and Requirement) when the state-of-the-world changes. This includes the Act Upon Violation and Confis Does Not Have Time in its Event Algebra limitations. Other lacks in expressiveness (like Confis Circumstances Are Domain-Aware) can be more easily circumvented by adapting Sentences to convey the intended meaning and by splitting clauses into several such Sentences.

Full examples of such discrepancies can be found in the code archive, as well as in Appendix A. We come to the conclusion that, despite the shortcomings of Confis as previously listed, we successfully represented the aforementioned contracts in the Confis specification.

6.2 Software Deliverables

Key software components like the language implementation, rule generation, and legal prose rendering have been thoroughly tested for correctness through unit and integration tests, in the spirit of rigorous software engineering. The methodology used in integration tests involves drafting plaintext contracts with the DSL implementation and comparing the contract and the internal representation it produces, as well as comparing the internal representation and the responses to different queries (which stem from rules generated from said internal representation). A brief test coverage report can be found in Table 6.4.

Component	**Coverage**
Language implementation	90.6%
Rule generation & Evaluation	85.0%
Natural language rendering	94.2%
Internal representation	73.5%

TABLE 6.4: Coverage for Confis components

Some deliverables of other specification languages (like the querying engine) are left unimplemented by competing technologies like Symboleo [4], therefore it is hard to perform quantitative comparisons. In order to still demonstrate the tractability and the practical applicability of the computations Confis performs, we measure compilation time and evaluation time of a Confis agreement. We measure the following:

- **Kotlin compilation time**, which converts text to a compiled Kotlin binary.
- **IR generation**, which converts Kotlin binaries to the Confis IR
- **Rule generation and evaluation**, which converts Confis IR to normative rules, and then evaluates the rules according to a given query (for all three kinds of queries)

The specifications of the machine these measurements are performed on are given in Table 6.5.

Hardware Model	Dell XPS 9300
Processor	1 core of Intel Core i7-1065G7
OS	Linux 5.18.3
JVM	OpenJDK RE (build 11.0.15+10)
Kotlin Compiler	version 1.7.0

TABLE 6.5: Specifications of host environment for performance evaluation

The methodology is as follows:

- We evaluate the Confis agreements `minimal` (see Listing A.2), `geophys` (see Listing A.3), and `meat` (see Listing A.5).

- No compilation caching of any kind is used (although the prototype does support caching).
- We measure Kotlin compilation, IR generation, and rule generation and evaluation (for all three query types discussed in Section 4.2). We evaluate rule generation and evaluation as a single measurement because they are both query-dependent (as in different rules are generated for different queries).
- To average out random variations caused by the kernel scheduler, we perform the measurements a hundred times and report the averages.
- Measurements are performed on a warm JVM [71].
- Confis code will be loaded into memory before measuring, so that fetching from disk is not taken into account.

The results of these measurements can be found in Table 6.6. Please refer to the code archive for the benchmarking code[1].

Step	**Duration**		
	`minimal`	`geophys`	`meat`
Kotlin compilation	280.9ms	314.9ms	342.9ms
IR generation from binaries	4.47us	3.9us	3.9us
Rule generation & evaluation (allowance)	182.4us	242.8us	354.1us
Rule generation & evaluation (compliance)	64.8us	242.9us	442.4us
Rule generation & evaluation (circumstance)	112.3us	116.7us	164.8us
Total	281.3ms	315.5ms	343.9ms

TABLE 6.6: Results of performance evaluation for `simple` and `geophys` agreements

We conclude that the bottleneck for converting a Confis agreement as text into a meaningful response for a query is the compilation time of the host language for the DSL, Kotlin; and that answers can be expected in the order of seconds even in pessimistic scenarios. Given the prototype was not designed for performance, this proves the applicability of the approach Confis takes (in the context of program running time).

[1]The code that performs the benchmarks can be found under:

- `script/src/test/kotlin/scripting/PerformanceTests.kt`

Chapter 7

Conclusions

This project has introduced:

- Confis, an accessible specification language for legal agreements, implemented as an internal DSL [45] using Kotlin [53] as a host language.
- An intelligent and language-aware editor equipped with all the utilities of the IDE of a general-purpose programming language, implemented as an IntelliJ [55] editor.
- Conversion from Confis back to natural language, encoded in Markdown and rendered as a live preview side-to-side with the editor.
- A query engine that generates normative rules from a Confis agreement, evaluates them, and answers complex queries; accompanied by a graphical interface that allows assembling such queries and rendering query results in plain natural language.
- A user guide as an online website to complement the above tools.

Confis is novel and in stark contrast with the state of the art in that

- It sets a precedent for successfully reconciling rule-based specification formalisms for legal agreements with accessible software and abstractions.
- It does not attempt to generate rules from text, nor introduce a specification which needs to be translated to other formalisms in order to be utilised. Confis is self-contained and flexible.

We find that it largely succeeds at introducing a machine-readable formalism for representing contracts (aided by novel features such as graphical query UIs and converting a formalism back to natural language), but at the cost of compromising on the expressiveness of its language and the contracts it is able to represent. This lack of expressiveness stems primarily from its lack of temporal logic and its lack of ability to deal with contract violations after they happen.

7.1 Future Work

Future work has two main directions: improving the existing language semantics and software prototype, and building upon the querying engine in order to programmatically deal with legal agreements.

7.1.1 A Better Language

This project chose to compromise on that area for the sake of a leaner learning curve – but it does not conclude new, still accessible abstractions with stronger guarantees are not possible.

More Circumstances and Temporal logic

A better version of Confis that does not compromise on accessibility may be possible with more Circumstance implementations, as well as perhaps integrating time and causality in the IR. New Circumstances could include geographical location, *force majeure* [72], repeated time periods, or possession of assets.

An 'On Breached' Allowance

As discussed in Section 6.1.1, Confis struggles to deal with planning for a scenario after the contract has been breached. This stems from the fact that a scenario can only result from an `Allow` Allowance, but breaches fall under the `Forbidden` state space.

Addressing this limitation could be possible by introducing a fifth type of Allowance that would account for the intersection of these two possible states.

More Complex Sentences

Confis sentences are extremely simple tuples following a Subject, Action, Object model. While it is possible to represent complex clauses by dividing them into several Sentences, Confis agreements would better reflect natural language if Sentences could contain several objects or several subjects (through disjunctions or conjunctions). The language would become more expressive, although rule generation would become more complex.

Confis Implemented as an External Language

As discussed in Subsection 3.2.2, Confis is developed as an internal DSL that uses Kotlin as its host language. This places constraints on the language's syntax (since it needs to be in line with Kotlin's). Writing a new parser and compiler for the language would remove this limitation, allowing for a grammar including sentences like `alice may eat cookie with commercial purposes` as opposed to the longer, more convoluted current syntax (an example of which can be found in Listing A.2). The new DSL could even be graphical instead of textual for even more ease of use [47].

This is possible thanks to having developed the Confis IR, which separates the language implementation from the rule generation that underpins querying.

7.1.2 A Query API

One of the key contributions of the project is the rule-based querying that allows asking a contract complex questions like *'Under what circumstances may I access this data set?'*. While Confis makes a strong focus on human interaction through accessible graphical interfaces, it has a binary interface that makes the prototype usable in any JVM-based application [71].

We envision distributed systems where services query and act according to the legal capabilities of their organisations given the circumstances they find themselves in. An example would be dataset access control and GDPR for an organisation that holds sensitive personal data: services would be able to update and process data while aware of what the organisation is and is not allowed to do it. Similarly, employees' access to customers' data can be automatically permissioned depending on how or why they are accessing the data, or which customer's data they are tampering with.

The notion of services acting following the specifications of a contract is in line with Szabo's original vision of smart contracts [2], much like Knottenbelt's work on contract-driven agents [36] – except Confis aims for non-logicians to write and review the contract specifications.

7.1.3 An Implementation for Public-Key Signing Infrastructure

Digital signatures (see Subsection C.1.2) are key to verifying the identity of parties of a contract. While Confis has all of the traits of a Ricardian Contract (see Definition 1) the public-key signing infrastructure is not implement by this project's prototype.

Developing such infrastructure could be a first step towards persisting verifiable Confis agreement metadata in a blockchain – which in turn also implements Proof Of Existence, much like what Express Agreement provides [31].

Appendix A

Confis Code Samples

A.1 Confis Software Archive Code Extracts

The following is a stripped-down version of the `AgreementBuilder` class (the full version can be found in the software archive). It is meant ot demonstrate how normal Kotlin functions can come together inside a single class that can be bound to a script in order to write agreements like that of Listing A.2.

```
open class AgreementBuilder {

    // metadata
    var title: String? = null
    var introduction: String? = null

    private val clauses = mutableListOf<Clause>()

    operator fun Action.invoke(obj: Obj): ActionObject = ActionObject(this, obj)

    /**
     * Specifies that [Subject] may perform [sentence]
     */
    infix fun Subject.may(s: ActionObject): Permission {
        val permission = Permission(Allow, Sentence(this, s.action, s.object)
        clauses += permission
        return permission
    }

    /**
     * Specifies that [Subject] may NOT perform [sentence]
     */
    infix fun Subject.may(s: ActionObject): Permission {
        val permission = Permission(Forbid, Sentence(this, s.action, s.object)
        clauses += permission
        return permission
    }
    // allows declaring parties, actions, and things to use in Sentences
    val party = oneTimeProperty<Any?, Subject> { prop -> Party(prop.name) }
    val action = oneTimeProperty<Any?, Action> { prop -> Action(prop.name) }
    val thing = oneTimeProperty<Any?, Obj> { prop -> Obj.Named(prop.name) }

}
```

LISTING A.1: A minimal example of the `AgreementBuilder` DSL

A.2 Confis Agreements Samples

Software Archive code path for Listing A.2:
`script/src/test/resources/scripts/minimal.confis.kts`

```
title = "Minimal example"

val alice by party
val eat by action
val cookie by thing

alice may eat(cookie)
```

LISTING A.2: A minimal example of a contract, writable with the `AgreementBuilder` from Listing A.1

Software Archive code path for Listing A.3:
`script/src/test/resources/scripts/geophys.confis.kts`

```


title = "SUB-LICENCE FOR SEISMIC DATA"

val licensee by party("the Licensee", description = "Oil & Gas Ltd")
val controller by party("the Controller",
    description = "The Controller of Her Majesty's Stationery Office"
)
val library by party("the Library",
    description = "UK Onshore Geophysical Library"
)

val data by thing("the Data", description = "seismic data listed in the Schedule")
val licence by thing("this Licence")
val nda by thing("a confidentiality agreement")

val provideServicesWith by action(
    "provide services with",
    description = "as in providing any service to any third party"
)
val transfer by action
val sell by action
val adapt by action(
    "copy or adapt",
    description = """as in deriving data and statistics from,
                     copying, or distributing"""
)
val thirdParty by party("a 3rd party", description = "not employed by $licensee")

val access by action(description = "as in gain obtain a copy of")

val agreeTo by action(
    "agree to",
    description = "to be bound by the terms of, to the same extend as $licensee"
)

val jointVentureGroup by party(
    named = "a Joint Venture Group",
    description = """A bona fide oil or gas bidding group,
                     exploration group, or exploitation group"""
)

val jointVentureGroupMember by party(
    named = "a member of $jointVentureGroup",
    description = "Another member, other than $licensee, of $jointVentureGroup"
)

val becomeMemberOf by action(
    named = "become a member of",
    description = "as in to be to at any point hereafter become a member or operator of"
)


introduction = """Seismic data acquired pursuant to operations conducted subject
    to the Petroleum Act 1998 and landward area regulations made in exercise of
    powers conferred thereby are Crown Copyright material. $library, pursuant to
    worldwide exclusive rights granted by $controller hereby grants $licensee a
    non-exclusive Licence to use $data on the following terms
"""

licensee mayNot transfer(licence) unless {
    with consentFrom library
}

licensee may transfer(licence) asLongAs {
    with consentFrom library
}

licensee mayNot sell(data)
licensee mayNot provideServicesWith(data)

thirdParty may access(data) asLongAs {
    with consentFrom licensee
    after { thirdParty did agreeTo(licence) }
    after { thirdParty did agreeTo(nda) }
}

licensee mayNot adapt(data) unless {
    with purpose Internal
    with consentFrom library
}

licensee may adapt(data) asLongAs {
    with purpose Internal
    with consentFrom library
}

licensee may becomeMemberOf(jointVentureGroup)

jointVentureGroupMember may access(data) asLongAs {
    after { licensee did becomeMemberOf(jointVentureGroup) }
    after { jointVentureGroupMember did becomeMemberOf(jointVentureGroup) }
    with consentFrom library
    after { jointVentureGroupMember did agreeTo(licence) }
}
```

LISTING A.3: A prototype Confis representation of a seismic data license (given in [5])

From Agreement, to IR, to Rules

The following is an example of a very simple Confis Agreement:

```
val alice by party
val use by action
val data by thing
alice may use(data) asLongAs {
    within { (1 of June)..(30 of June) year 2022 }
}
```

FIGURE A.1: Minimal Confis agreement with a circumstance

This clause is simple and generates a single rule when querying for a circumstance question (*'Under what circumstances may Alice use data?'*). The clause is translated into the following IR:

```
Agreement(
    clauses = [
            PermissionWithCircumstances(
                allowance = Allow,
                sentence = Sentence(
                    Party("alice"),
                    Action("use"),
                    Obj.Named("data")
                ),
                circumstanceAllowance = Allow,
                circumstances = CircumstanceSet(
                    TimeRange.Key -> TimeRange(01/07/2022..30/01/2022),
                ),
            ),
        ],
    parties = [Party("alice")],
    title = null,
    description = null,
)
```

LISTING A.4: IR of minimal Confis agreement with a circumstance from Figure A.1

A.2.1 Meat Contract Comparison

This subsection hopes to illustrate the main differences between different representations of the same contract (given in Table A.1). These differences are discussed under Evaluation under Section 6.1.

This agreement is entered into as of the date <effDate>, between <party1> as Seller with the address <retAdd>, and <party2> as Buyer with the address <delAdd>.

1. **Payment & Delivery**
   1.1. Seller shall sell an amount of <qnt> meat with <qlt> quality ("goods") to the Buyer.
   1.2. Title in the Goods shall not pass on to the Buyer until payment of the amount owed has been made in full.
   1.3. The Seller shall deliver the Order in one delivery within <delDueDateDays> days to the Buyer at its warehouse.
   1.4. The Buyer shall pay <amt> ("amount") in <curr> ("currency") to the Seller before <payDueDate>.
   1.5. In the event of late payment of the amount owed due, the Buyer shall pay interests equal to <intRate> the Seller may suspend performance of all of its obligations under the agreement until payment of amounts due has been received in full.
2. **Assignment**
   2.1. The rights and obligations are not assignable by Buyer.
3. **Termination**
   3.1. Any delay in delivery of the goods will not entitle the Buyer to terminate the Contract unless such delay exceeds 10 Days.
4. **Confidentiality**
   4.1. Both Seller and Buyer must keep the contents of this contract confidential during the execution of the contract and six months after the termination of the contract.

TABLE A.1: Sample of a meat purchase and sales contract, from [4]

Software Archive code path for Listing A.5:
`script/src/test/resources/scripts/meat.confis.kts`

```
title = "Meat Purchase contract"

val effDate = 1 of June year 2022
val payDueDate = 10 of June year 2022
val deliveryDueDate = 15 of June year 2022

val seller by party("the Seller", description = "Alice Liddell")
val buyer by party("the Buyer", description = "The Meat Supermarket, Inc")

val meat by thing("the Goods", description = "30kg of beef")

val giveMeatTo by action("give $meat to",
    description = "as in pass on the title in $meat to"
)

val deliverMeatTo by action("deliver $meat to",
    description = "as in deliver $meat in one delivery at the warehouse of"
)

val contract by thing("the Contract",
    description = "this legal agreement"
)

val amt = 20
val curr = "EUR"
val interest = "3%"

val payMeatPriceTo by action("pay for $meat to",
    description = "as in pay $amt in $curr to"
)
val payInterestMeatPriceTo by action("pay for $meat to",
    description = "as in pay $amt in $curr with $interest interest to"
)

buyer may payMeatPriceTo(seller) asLongAs {
    at { payDueDate }
}
buyer mayNot payMeatPriceTo(seller) unless {
    at { payDueDate }
}

seller must giveMeatTo(buyer) underCircumstances {
    after { buyer did payMeatPriceTo(seller) }
}

seller must deliverMeatTo(buyer) underCircumstances {
    after { buyer did payMeatPriceTo(buyer) }
    within { deliveryDueDate..(deliveryDueDate + 10.days) }
}
buyer may payInterestMeatPriceTo(buyer) asLongAs {
    after(payDueDate)
}

// termination
val terminate by action
buyer mayNot terminate(contract) asLongAs {
    // we do not need to specify 10 day delivery because a
    // longer delivery is outside the capabilities of the Seller
    after { seller did deliverMeatTo(buyer) }
}

// confidentiality
val reveal by action(
    description = "as in not keeping the contents confidential"
)
seller mayNot reveal(contract) asLongAs {
    within { effDate..(effDate + 6.months) }
}

buyer mayNot reveal(contract) asLongAs {
    within { effDate..(effDate + 6.months) }
}
```

LISTING A.5: Confis Agreement for Sample meat sales contract

```
Domain meatSaleDomain
```

```
Seller isA Role with returnAddress: String, name: String;
Buyer isA Role with warehouse: String;
Currency isAn Enumeration(CAD, USD, EUR);
MeatQuality isAn Enumeration(PRIME, AAA, AA, A);
PerishableGood isAn Asset with quantity: Number, quality: MeatQuality;
Meat isA PerishableGood;
Delivered isAn Event with
    item: Meat,deliveryAddress: String, delDueDate: Date;
Paid isAn Event with
    amount: Number,
    currency: Currency,
    from: Buyer,
    to: Seller,
    payDueDate: Date;
PaidLate isAn Event with
    amount: Number, currency: Currency, from: Buyer, to: Seller;
Disclosed isAn Event;

endDomain

Contract MeatSale (
    buyer : Buyer,
    seller : Seller,
    qnt : Number,
    qlt : MeatQuality,
    amt : Number,
    curr : Currency,
    payDueDate: Date,
        delAdd : String,
        effDate : Date,
        delDueDateDays : Number,
        interestRate: Number
)

Declarations
goods: Meat with quantity := qnt, quality := qlt;
delivered: Delivered with
    item := goods,
    deliveryAddress := delAdd,
    delDueDate := Date.add(effDate, delDueDateDays, days);
paidLate: PaidLate with
    amount := (1 + interestRate / Math.abs(2)),
    currency := curr,
    from := buyer,
    to := seller;
paid: Paid with
    amount := amt,
    currency := curr,
    from := buyer,
    to := seller,
    payDueDate := payDueDate;
disclosed: Disclosed;

Preconditions
IsOwner(goods, seller);

Postconditions
```

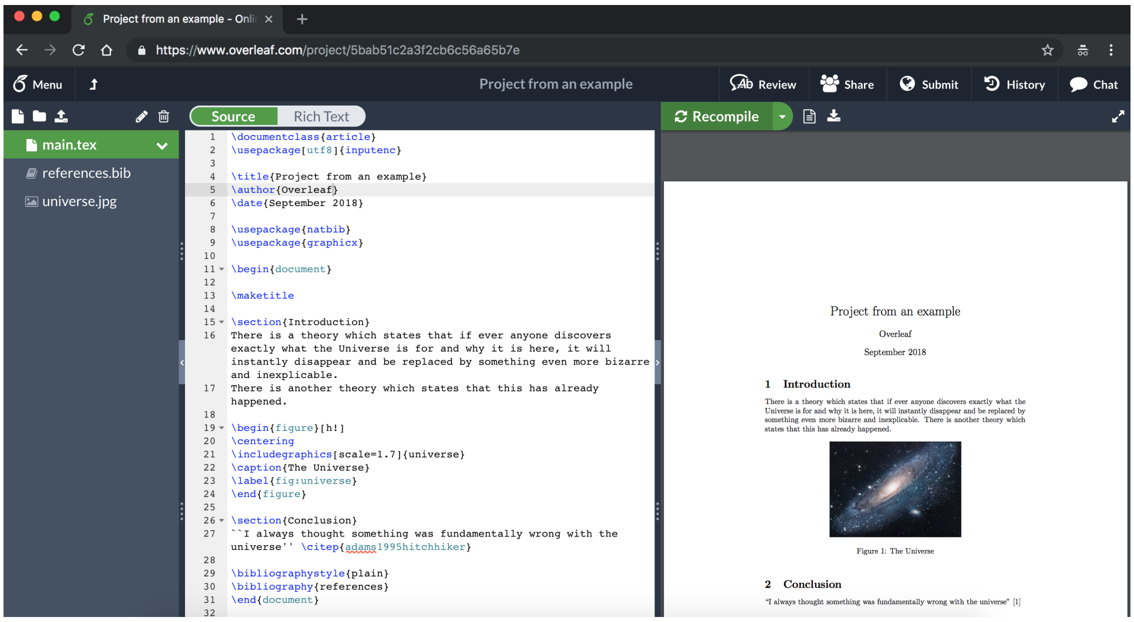
```
IsOwner(goods, buyer) and not(IsOwner(goods, seller));

Obligations
delivery:
    Obligation(seller, buyer, true, WhappensBefore(delivered, delivered.delDueDate));
payment: O(buyer, seller , true, WhappensBefore(paid, paid.payDueDate));
latePayment:
    Happens(Violated(payment)) -> O(buyer, seller, true, Happens(paidLate));

Surviving Obligations
so1 : Obligation(
    seller,
    buyer,
    true,
    not WhappensBefore(disclosed, Date.add(Activated(self), 6, months))
);

so2 : Obligation(
    buyer,
    seller,
    true,
    not WhappensBefore(disclosed, Date.add(Activated(self), 6, months))
);

Powers
suspendDelivery
    : Happens(Violated(payment))
        -> Power(seller, buyer, true, Suspended(delivery));
resumeDelivery
    : HappensWithin(paidLate,Suspension(delivery))
        -> P(buyer, seller, true, Resumed(delivery)
);
terminateContract
    : Happens(Violated(delivery)) -> P(buyer, seller, true, Terminated(self));

Constraints
not(IsEqual(buyer, seller));
CannotBeAssigned(suspendDelivery);
CannotBeAssigned(resumeDelivery);
CannotBeAssigned(terminateContract);
CannotBeAssigned(delivery);
CannotBeAssigned(payment);
CannotBeAssigned(latePayment);
delivered.delDueDate < paid.payDueDate;

endContract
```

LISTING A.6: Symboleo Specification for Sample meat sales contract, from [70, 4]

Appendix B

Editor Previews

B.1 Third Party Editors



```
\documentclass{article}
\usepackage[utf8]{inputenc}

\title{Project from an example}
\author{Overleaf}
\date{September 2018}

\usepackage{natbib}
\usepackage{graphicx}

\begin{document}

\maketitle

\section{Introduction}
There is a theory which states that if ever anyone discovers
exactly what the Universe is for and why it is here, it will
instantly disappear and be replaced by something even more bizarre
and inexplicable.
There is another theory which states that this has already
happened.

\begin{figure}[h!]
\centering
\includegraphics[scale=1.7]{universe}
\caption{The Universe}
\label{fig:universe}
\end{figure}

\section{Conclusion}
``I always thought something was fundamentally wrong with the
universe'' \citep{adams1995hitchhiker}

\bibliographystyle{plain}
\bibliography{references}
\end{document}

```

Project from an example

Overleaf

September 2018

1 Introduction

There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened.

Figure 1: The Universe

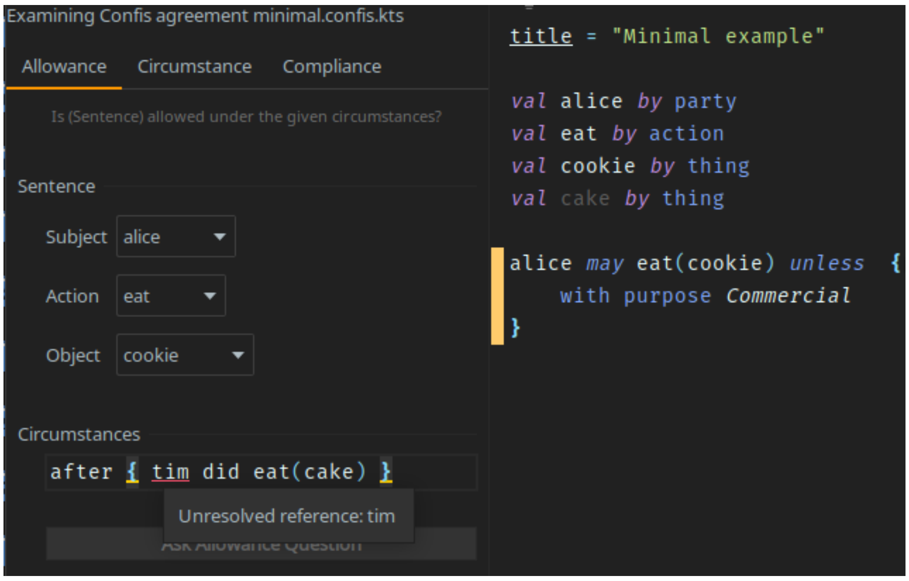
2 Conclusion

"I always thought something was fundamentally wrong with the universe" [1]

FIGURE B.1: The LaTeX editor Overleaf, along with a preview of the document being authored, from [73]

B.2 Confis Query UI

The following are screenshots of the Confis Query UI (discussed in Section 4.3). They have been picked to demonstrate different features of the querying UI and types of questions and responses available.



Examining Confis agreement minimal.confis.kts

Is (Sentence) allowed under the given circumstances?

Sentence

Subject: alice

Action: eat

Object: cookie

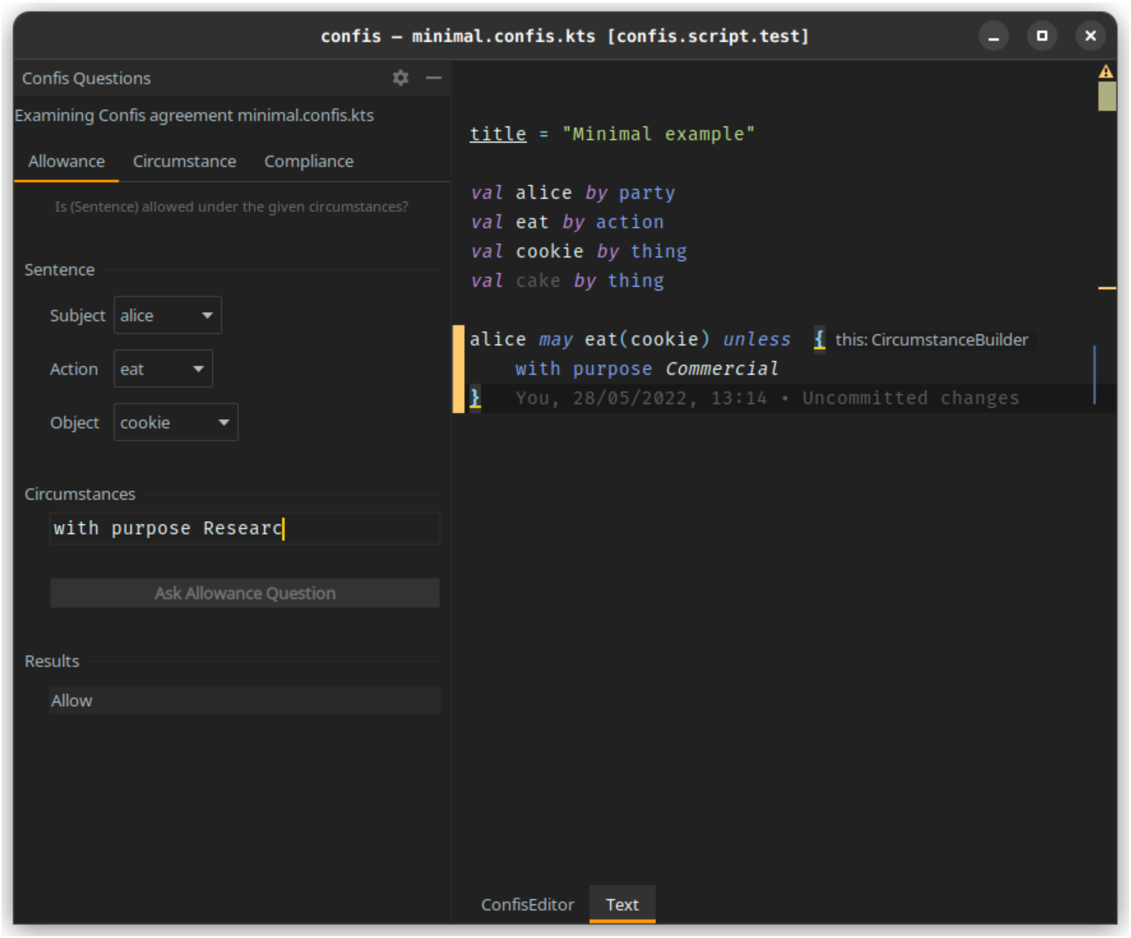
Circumstances

```
after { tim did eat(cake) }
```

Unresolved reference: tim

```
title = "Minimal example"

val alice by party
val eat by action
val cookie by thing
val cake by thing

alice may eat(cookie) unless {
    with purpose Commercial
}
```

FIGURE B.2: Query UI window featuring a compile error when assembling a Circumstance



Examining Confis agreement minimal.confis.kts

Is (Sentence) allowed under the given circumstances?

Sentence

Subject: alice

Action: eat

Object: cookie

Circumstances

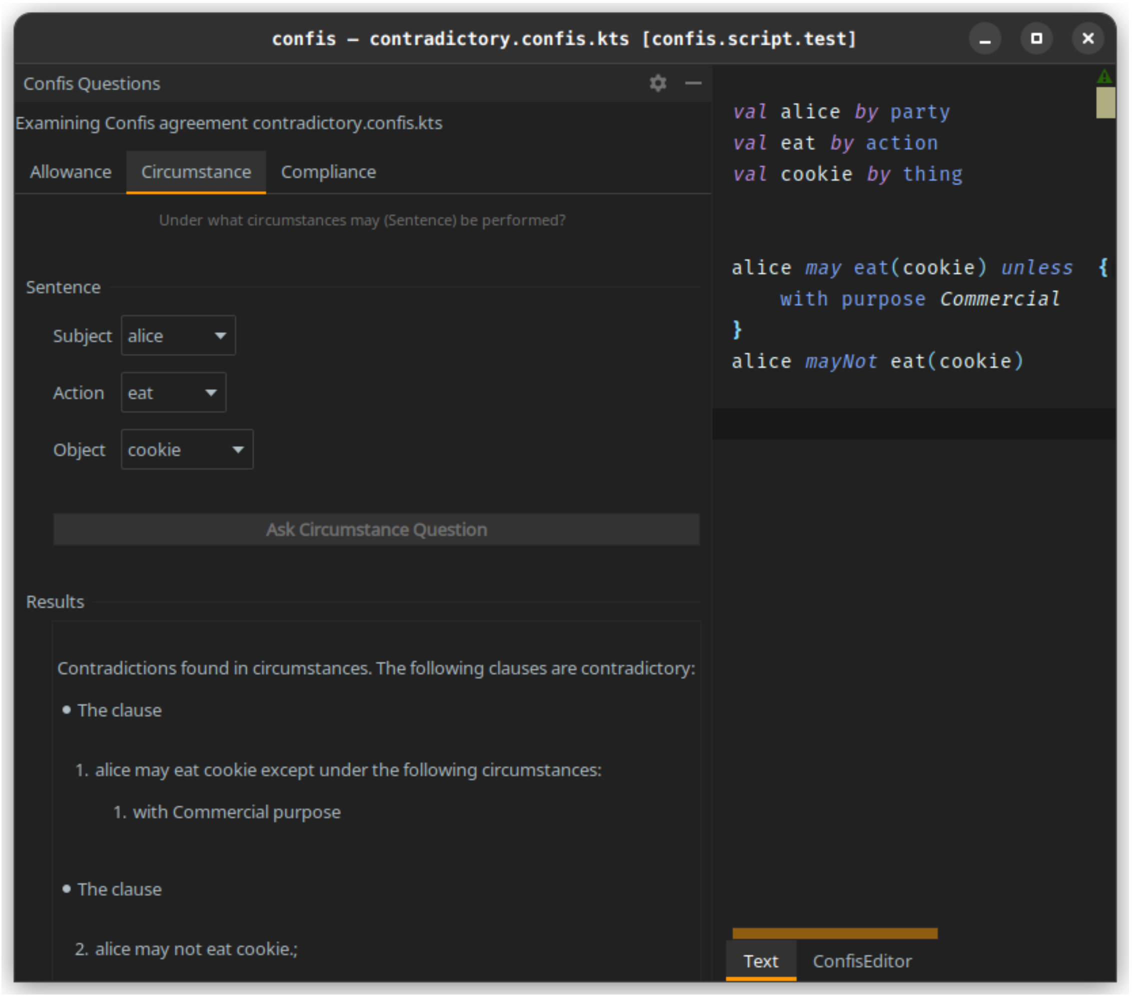
```
with purpose Researc
```

Results

Allow

```
title = "Minimal example"

val alice by party
val eat by action
val cookie by thing
val cake by thing

alice may eat(cookie) unless { this: CircumstanceBuilder
    with purpose Commercial
}   You, 28/05/2022, 13:14 • Uncommitted changes
```

FIGURE B.3: Full view of the query UI featuring an Allowance question


confis – contradictory.confis.kts [confis.script.test]
Confis Questions
Allowance
Circumstance
Compliance
Ask Circumstance Question
Text
ConfisEditor

Examining Confis agreement contradictory.confis.kts

Under what circumstances may (Sentence) be performed?

Sentence

Subject: alice

Action: eat

Object: cookie

Results

Contradictions found in circumstances. The following clauses are contradictory:

- The clause

  1. alice may eat cookie except under the following circumstances:
     1. with Commercial purpose

- The clause

  2. alice may not eat cookie.;

```
val alice by party
val eat by action
val cookie by thing


alice may eat(cookie) unless {
    with purpose Commercial
}
alice mayNot eat(cookie)
```

FIGURE B.4: Query UI featuring a contradiction

Appendix C

Cryptography and Distributed Ledgers Background

This chapter is dedicated to background on cryptography primitives and distributed ledgers. While such concepts are not required to understand the semantics or the implementation of Confis, they are helpful to understand blockchain-based smart contracts (an application of the original concept by Szabo [2]) which are discussed in Subsection 2.2.1 and are a significant part of the state of the art when it comes to machine-readable of agreements between parties – which does concern this project.

C.1 Cryptography Primitives

C.1.1 Hash Functions

Hash functions are cryptographic functions designed to behave like random functions [74]. When building a security proof, they can be assumed to have the following properties [74, 75]:

Determinism A hash function always produces the same output for the same input

One-way It is computationally impossible to compute the preimage for some output of a hash function

Uniformity Outputs of a hash function are expected to be uniformly distributed. In practice, the output space of a hash function is finite, so *collisions* (where two inputs produce the same output) are possible, but uniformity ensures this is an unlikely scenario.

C.1.2 Public Key Cryptography

In public key cryptography, two communicating parties (say Alice and Bob) can communicate privately by using pairs of numbers that are related mathematically and which allow converting cleartext into cyphertext and back [76]. This pair of numbers is called an asymmetric keypair, and is composed of a **public key** e and a **private key** d.

In this example, if Alice wishes to communicate with Bob, Bob can generate a keypair (d, e) and publish e. Alice can then encrypt her cleartext with e, and only Bob will be able to decrypt it (because only Bob knows d).

Conversely, the same keypair can be used by Bob to send a message to Alice where Alice can verify that only Bob could have written the message. This is called *signing* [77] and, more generally, it allows a sender of a message to prove they are the authors of the message to a recipient.

C.1.3 Secure Digital Timestamps

Trusted (digital) timestamping is the process of securely proving that a document (for our purposes, a blob of bytes) was created, was modified at, or existed at a certain point in time. In industry this is commonly implemented by trusting a Time Stamping Authority (TSA) [78] that signs (see Public Key Cryptography) a concatenation of the hash of the document and a timestamp representing some time t. Therefore, a party that trusts the TSA to provide the right timestamp can verify that, when the TSA made the signature, the current time was t.

This method can be used for confidential data because the TSA does not perform the hashing of the original document themselves - they are exposed only to its digest.

Additionally, the requester of the timestamp cannot deny they were not in possession of the original data at the time t given by the timestamp, because it was them that produced its hash digest.

Decentralised Timestamps

Secure timestamps can also be achieved without relying on trusted parties by publishing the document digest to a blockchain [79]: blocks are public and cannot be tampered with (see Proof-Of-Work), so putting a signed digest in a block shows that the signer knew the original document at the time the block was accepted by the network.

C.2 Bitcoin Protocol

Blockchain technology was introduced in [14] as a decentralised system allowing for electronic cash payments. Blockchains are immutable distributed ledgers where participants' balances can be verified by every other participant, and it is computationally hard to tamper with balances to perform attacks (such as performing a transaction where a participant spends more funds than what they own).

I will provide a brief overview of how Bitcoin provides these guarantees.

C.2.1 Transactions

[14] defines an *electronic coin* as a chain of signatures: a payer can use their private key, the hash of the previous transaction, and the payee's public key to create a signed hash that can be verified by the payee (and used by them for *their* next transaction). This is illustrated in Figure C.1.

This ensures that, as long as a participants sign transactions at most once:

- By verifying the chain of signatures, every participant can verify which participant owns which coin
- Only the owner of a coin can initiate a transaction with that coin

Bitcoin enforces that participants can only sign transactions once thanks to its proof-of-work (see C.2.2) algorithm.

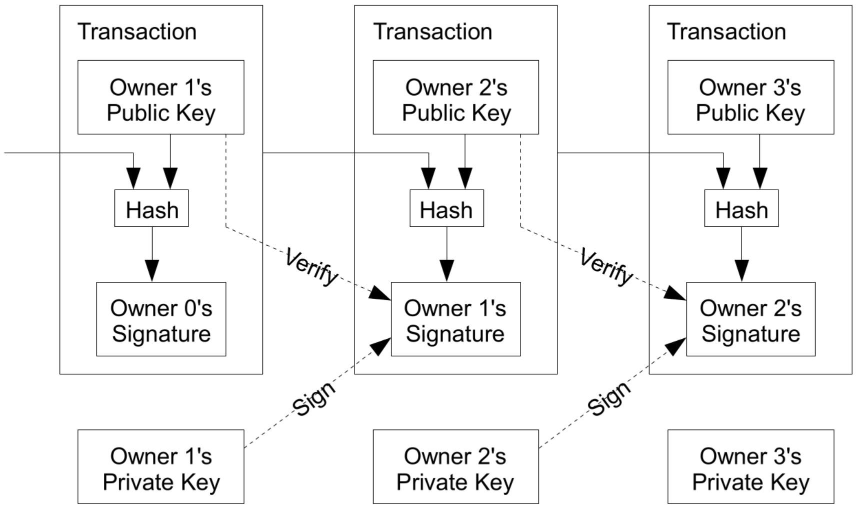


FIGURE C.1: Transfer of ownership signature chain, from [14]

C.2.2 Proof-Of-Work

Bitcoin ensures 'unique signatures' in transactions by grouping transactions in immutable, public *blocks*. Participants can then verify a payer has not signed a hash of a single transaction twice by looking at all existing transactions.

Blocks are made immutable by including in them a value (called a *nonce*) and the hash of the previous block. The protocol then accepts only blocks where the n first bits of its hash are zeroes.

Thus, in order to publish a block a participant must do work to find a nonce such that the block's hash meets this condition - then other participants can verify its validity with a single hash operation. This guarantees that a block cannot be changed (ie, a new copy published) without redoing the computational work. Because blocks are chained (they include the hash of the previous block), in order to modify a transaction in the past an adversary needs to redo the computational work for every block since that transaction (see Figure C.2). Additionally, participants that successfully find a suitable nonce and propose new blocks (also referred to as *miners*) are allowed to add a specific transaction to the block where they own a newly created coin (also referred to as *mining reward*).

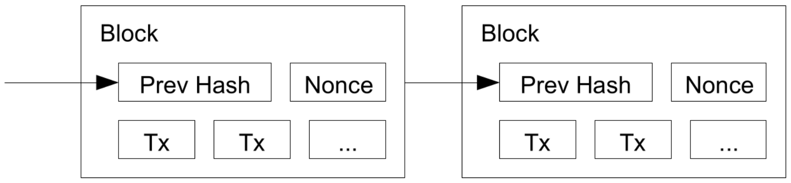


FIGURE C.2: Two last blocks in a blockchain, from [14]

This model of consensus ensures that

- Participants have a monetary incentive to stay honest with respect to the protocol
- An honest chain will out-compete an adversary's chain as long as the majority of computing power is honest

C.2.3 Further details of the Bitcoin protocol

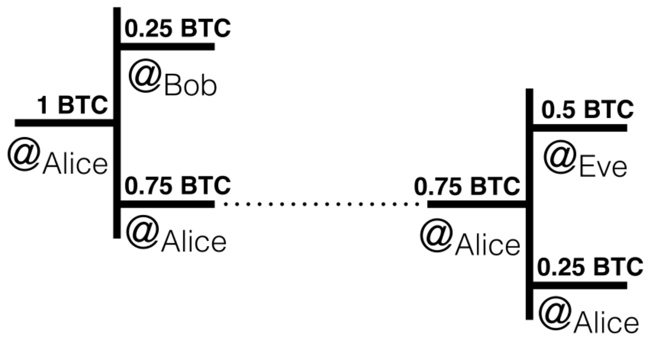


FIGURE C.3: The outputs of a transaction correspond to the input of the next transaction (miner's fee not represented) from [80]

While I provide a high-level overview of what makes the protocol function, there are many more details that combined allow for more efficiency and usability:

- Transactions may have several inputs and outputs, so participants can transfer amounts rather than single *electronic coins*. When performing a payment, a typical transaction by Alice has two outputs: the amount she is paying Bob and the remainder, which makes up the rest of her funds (see Figure C.3). Thus, a participant's balance is the sum of all the unspent outputs of previous transactions (the set of unspent outputs is commonly called the *UTXO* set).
- By modifying how many of the leading bits of a blocks' hash must be zeroes, the average computation necessary to produce a new block can be adjusted by the protocol.
- By using Merkle Trees [81] transactions with fully spent transaction outputs can be discarded without breaking the block's hash. This allows compacting old blocks to reclaim disk space.
- A participant that does not wish to mine or hold a copy of the entire blockchain can still verify payments. It can keep a copy of the block headers of the longest chain and link a transaction to where it is on-chain and check that other network nodes have accepted it. This process is called *Simple Payment Verification* (SPV) [14].
- Miners have an additional incentive (other than the mining reward) to verify transactions: the *transaction fee*. The difference of the sum of a transaction's inputs and the sum its outputs corresponds to the transaction fee, which goes to the miner.

  $$\text{fee}_{\text{miner}} = \sum \text{inputs} - \sum \text{outputs}$$

  This provides an incentive for the miner to place this specific transaction in the block they propose.

For more information on all the workings of the Bitcoin protocol, please refer to [14].

Bibliography

1. Larson A. Contract Law - An Introduction. 2016 May 24. Available from: https://www.expertlaw.com/library/business/contract_law.html [Accessed on: 2022 May 23]
2. Szabo N. Formalizing and securing relationships on public networks. First monday 1997. DOI: 10.5210/FM.V2I9.548. Available from: https://journals.uic.edu/ojs/index.php/fm/article/view/548/469
3. Accord Contributors. Accord Project. 2022. Available from: https://docs.accordproject.org/ [Accessed on: 2022 Jan 19]
4. Sharifi S, Parvizimosaed A, Amyot D, Logrippo L, and Mylopoulos J. Symboleo: Towards a Specification Language for Legal Contracts. *2020 IEEE 28th International Requirements Engineering Conference (RE)*. IEEE, 2020 Aug. DOI: 10.1109/re48521.2020.00049. Available from: https://doi.org/10.1109%2Fre48521.2020.00049
5. UK Onshore Geophysical Library. Sub-License for Seismic Data. 2022. Available from: https://ukogl.org.uk/data-licensing/ [Accessed on: 2022 Jan 19]
6. Winter V Nemethj. Case - Contract Law. 2018 May 31. Available from: https://www.ruleoflaw.org.au/contract-law/ [Accessed on: 2022 Jan 18]
7. Lehman J and Phelps S. "contract". *West's encyclopedia of American law*. Thomson/Gale, 2004. Available from: https://legal-dictionary.thefreedictionary.com/Contract [Accessed on: 2022 Jan 18]
8. Lehman J and Phelps S. "license". *West's encyclopedia of American law*. Thomson/Gale, 2004. Available from: https://legal-dictionary.thefreedictionary.com/License [Accessed on: 2022 Jan 18]
9. Economist Intelligence Unit. Terms and Conditions: Licensing Agreement. The Economist Group. 2016 May. Available from: https://www.economistgroup.com/pdfs/terms_and_conditions/Terms_and_Conditions-Data_Licensing_Agreement-13.May.2016.pdf [Accessed on: 2022 Jan 24]
10. Jetbrains s.r.o. Toolbox Subscription Agreement for Students and Teachers. 2021 Sep. Available from: https://www.jetbrains.com/legal/docs/toolbox/license_educational/ [Accessed on: 2022 Jan 18]
11. JetBrains s.r.o. JetBrains Toolbox App. Version 1.22. 2021. Available from: https://www.jetbrains.com/toolbox-app/ [Accessed on: 2022 Jan 20]
12. 'Machine-readable'. *Cambridge Dictionary*. Ed. by Cambridge University Press. 2022. Available from: https://dictionary.cambridge.org/dictionary/english/machine-readable [Accessed on: 2022 May 23]
13. 'Machine-readable'. *Open Data Handbook*. Ed. by Open Knowledge Foundation. 2022. Available from: http://opendatahandbook.org/glossary/en/terms/machine-readable/ [Accessed on: 2022 May 23]
14. Nakamoto S et al. Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review 2008 :21260. Available from: https://bitcoin.org/bitcoin.pdf [Accessed on: 2022 Jan 15]
15. Bitcoin Wiki. Script. 2022 Jan. Available from: https://en.bitcoin.it/wiki/Script [Accessed on: 2022 Jan 16]

16. Buterin V. A Next-Generation Smart Contract and Decentralized Application Platform. 2015. Available from: https://ethereum.org/en/whitepaper/ [Accessed on: 2022 Jan 15]
17. Solidity Team. Solidity Github Repository. Version 0.8.11. Git Repository. 2022. Available from: https://github.com/ethereum/solidity [Accessed on: 2022 Jan 16]
18. Vyper Team. Vyper Github Repository. Version 0.3.1. Git Repository. 2022. Available from: https://github.com/vyperlang/vyper [Accessed on: 2022 Jan 18]
19. Ethereum Contributors. Smart Contract Languages. Ethereum Developer Documentation. 2021 Dec 22. Available from: https://ethereum.org/en/developers/docs/smart-contracts/languages/ [Accessed on: 2022 Jan 18]
20. Antonopoulos AM and Wood G. Glossary. *Mastering Ethereum: Building Smart Contracts and DApps*. O'reilly Media, 2018 :xvii
21. Mabey R and Kovalevich P. Machine-Readable Contracts: a New Paradigm for Legal Documentation. Juro. 2019. Available from: https://info.juro.com/machine-readable-contracts [Accessed on: 2022 Jan 19]
22. Ecma International. Introducing JSON. 2017. Available from: https://www.json.org/json-en.html [Accessed on: 2022 Jan 19]
23. Adobe, ISO. Document management — Portable document format — Part 2: PDF 2.0. ISO 32000-2:2020. Tech. rep. ISO, 2020 Feb. Available from: https://www.iso.org/standard/75839.html [Accessed on: 2022 Jan 19]
24. Microsoft. Microsoft Word. 2022. Available from: https://www.microsoft.com/en-gb/microsoft-365/word [Accessed on: 2022 Jan 19]
25. Gruber J and Swartz A. Markdown. 2004 Dec 17. Available from: https://daringfireball.net/projects/markdown/ [Accessed on: 2022 Jan 19]
26. Accord Contributors. Accord Project Template. After Signature. 2022. Available from: https://docs.accordproject.org/docs/accordproject-template#after-signature [Accessed on: 2022 Jan 19]
27. Accord Contributors. Ergo. 2022. Available from: https://accordproject.org/projects/ergo/ [Accessed on: 2022 Jan 20]
28. HyperLedger Foundation. HyperLedger Fabric Github Repository. Version 2.4.1. Git Repository. 2022. Available from: https://github.com/hyperledger/fabric [Accessed on: 2022 Jan 20]
29. Accord Contributprs. Deploying on Hyperledger Fabric. 2022. Available from: https://docs.accordproject.org/docs/advanced-hyperledger.html [Accessed on: 2022 Jan 20]
30. Accord Contributors. Template Execution in Node.js. 2022. Available from: https://docs.accordproject.org/docs/tutorial-nodejs.html [Accessed on: 2022 Jan 20]
31. Zhang T. Express Agreement: A Framework for Efficient Contract Negotiation and Blockchain-based Agreement Verification. MSc Project. Department of Computing, Imperial College London, 2015-09
32. Grigg I. The ricardian contract. *Proceedings. First IEEE International Workshop on Electronic Contracting, 2004*. IEEE. 2004 :25–31. DOI: 10.1109/WEC.2004.1319505
33. WebFunds. Ricardian Contracts. 2004. Available from: http://webfunds.org/guide/ricardian.html [Accessed on: 2022 Apr 27]
34. Sartor G. Legal reasoning. *A Treatise of Legal Philosophy and General Jurisprudence*. Ed. by Roversi C. Vol. 5. Springer Netherlands, 2005. DOI: 10.1007/1-4020-3505-5. Available from: https://doi.org/10.1007%2F1-4020-3505-5
35. Ferraro G, Lam HP, Tosatto SC, Olivieri F, Islam MB, Beest N van, and Governatori G. Automatic Extraction of Legal Norms: Evaluation of Natural Language Processing

Tools. *New Frontiers in Artificial Intelligence*. Springer International Publishing, 2020 Sep 11:64–81. DOI: 10.1007/978-3-030-58790-1_5. Available from: https://doi.org/10.1007%2F978-3-030-58790-1_5

36. Knottenbelt J and Clark K. Contract Driven Agents. *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, USA*. Ed. by Russell I and Markov Z. AAAI Press, 2005 :832–3. Available from: http://www.aaai.org/Library/FLAIRS/2005/flairs05-145.php
37. Kowalski R and Sergot M. A logic-based calculus of events. *Foundations of knowledge base management*. Springer, 1989 :23–55
38. Allen JF. Towards a general theory of action and time. Artificial Intelligence 1984 Jul; 23:123–54. DOI: 10.1016/0004-3702(84)90008-0. Available from: https://doi.org/10.1016%2F0004-3702%2884%2990008-0
39. Goldberg Y. A Primer on Neural Network Models for Natural Language Processing. Journal of Artificial Intelligence Research 2016; 57:345–420. DOI: 10.1613/jair.4992. Available from: https://doi.org/10.1613%2Fjair.4992
40. Angeli G, Premkumar MJJ, and Manning CD. Leveraging linguistic structure for open domain information extraction. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2015 :344–54
41. Dong L and Lapata M. Language to Logical Form with Neural Attention. 2016. DOI: 10.48550/ARXIV.1601.01280. Available from: https://arxiv.org/abs/1601.01280 [Accessed on: 2022 May 23]
42. Dong L and Lapata M. Coarse-to-Fine Decoding for Neural Semantic Parsing. 2018. DOI: 10.48550/ARXIV.1805.04793. Available from: https://arxiv.org/abs/1805.04793
43. Sleimi A, Sannier N, Sabetzadeh M, Briand L, and Dann J. Automated Extraction of Semantic Legal Metadata using Natural Language Processing. *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 2018. DOI: 10.1109/re.2018.00022. Available from: https://doi.org/10.1109%2Fre.2018.00022 [Accessed on: 2022 May 24]
44. Woodsend K and Lapata M. WikiSimple: Automatic Simplification of Wikipedia Articles. Proceedings of the AAAI Conference on Artificial Intelligence 2011; 25:927–32. Available from: https://ojs.aaai.org/index.php/AAAI/article/view/7967 [Accessed on: 2022 May 23]
45. Martin Fowler RP. Domain-Specific Languages. Pearson Education, 2010
46. Fowler M. Domain-Specific Languages Guide. 2019 Aug 28. Available from: https://martinfowler.com/dsl.html [Accessed on: 2022 May 24]
47. Fowler M. Language Workbenches: The Killer-App for Domain Specific Languages? 2005 Jun 12. Available from: https://martinfowler.com/articles/languageWorkbench.html [Accessed on: 2022 May 24]
48. the Haskell Community. Haskell Documentation. 2022. Available from: https://www.haskell.org/documentation/ [Accessed on: 2022 May 23]
49. Augustss. BASIC Reimplemented in Haskell via DSL. 2009 Feb 07. Available from: http://augustss.blogspot.com/search/label/BASIC [Accessed on: 2022 May 24]
50. DevTut. Haskell Fixity Declarations. 2022. Available from: https://devtut.github.io/haskell/fixity-declarations.html#fixity-declarations [Accessed on: 2022 May 22]
51. Apache Software Foundation. The Groovy Language. 2022. Available from: http://www.groovy-lang.org/index.html [Accessed on: 2022 May 25]

52. Gradle, Inc. Gradle Build Language Reference. Version 7.4.2. 2022. Available from: https://docs.gradle.org/7.4.2/dsl/index.html [Accessed on: 2022 May 25]
53. JetBrains, Inc. The Kotlin Language. 2022. Available from: https://kotlinlang.org/ [Accessed on: 2022 May 25]
54. JetBrains, Inc. Type-Safe Builders. 2022 Feb 02. Available from: https://kotlinlang.org/docs/type-safe-builders.html [Accessed on: 2022 May 25]
55. JetBrains, Inc. IntelliJ Community Source. Git Repository. Version 222.2680.4. 2022 May 25. Available from: https://github.com/JetBrains/intellij-community/tree/idea/222.2680.4 [Accessed on: 2022 May 26]
56. Oracle, Inc. The Java Scripting API. Tech. rep. Version Java 8. Oracle, 2022. Available from: https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/api.html [Accessed on: 2022 May 26]
57. JetBrains, Inc. Kotlin Scripting Support Proposal. Kotlin Evolution & Enhancement Process Proposal. Version 61b840d. Kotlin, 2021 Aug 30. Available from: https://github.com/Kotlin/KEEP/blob/master/proposals/scripting-support.md [Accessed on: 2022 May 26]
58. The Apache Foundation. Groovy Scripting Documentation. Tech. rep. Version 4.0.0. Apache, 2022. Available from: https://docs.groovy-lang.org/next/html/documentation/#_custom_script_class [Accessed on: 2022 May 26]
59. JetBrains, Inc. Plugin Structure Documentation. Extensions. 2022. Available from: https://plugins.jetbrains.com/docs/intellij/plugin-extensions.html [Accessed on: 2022 May 26]
60. Microsoft. Visual Studio Code Extension API. 2022. Available from: https://code.visualstudio.com/api [Accessed on: 2022 May 26]
61. Kowalski RA. The early years of logic programming. Communications of the ACM 1988; 31:38–43. DOI: 10.1145/35043.35046. Available from: https://doi.org/10.1145%2F35043.35046
62. Potassco Project Team. The Potassco Project. 2022. Available from: https://potassco.org/ [Accessed on: 2022 Jul 01]
63. Hassine MB. Easy Rules. Git Repository. Version 4.1.0. 2021. Available from: https://github.com/j-easy/easy-rules [Accessed on: 2022 Jul 01]
64. Fowler M. Rules Engine. 2009 Jan 07. Available from: https://martinfowler.com/bliki/RulesEngine.html [Accessed on: 2022 Jul 01]
65. Sanderson H and Rosenthal H. Interacting with the Unix Command Line. *Learn AppleScript*. Apress, 2010 :863–96. DOI: 10.1007/978-1-4302-2362-7_27. Available from: https://doi.org/10.1007%2F978-1-4302-2362-7_27
66. JetBrains, Inc. Operator Overloading. 'invoke' operator. Tech. rep. Version 1.6.21. Available from: https://kotlinlang.org/docs/operator-overloading.html#invoke-operator [Accessed on: 2022 May 26]
67. JetBrains, Inc. Markdoown. HTML Preview. Tech. rep. Version 2022.1. 2022. Available from: https://www.jetbrains.com/help/idea/markdown.html#preview [Accessed on: 2022 May 29]
68. Christie T and The MkDocs Team. MkDocs. 2014. Available from: https://www.mkdocs.org/ [Accessed on: 2022 Jun 07]
69. European Commission. Data protection in the EU. The General Data Protection Regulation (GDPR), the Data Protection Law Enforcement Directive and other rules concerning the protection of personal data. 2022. Available from: https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en [Accessed on: 2022 Jan 20]
70. Contract Smart Modelling Group. Symbolio for Meat Sale Sample. Github Repository. Version 8d5d6ee. University of Ottawa. Available from: https://github.com/Smart-

Contract-Modelling-uOttawa/Symboleo-IDE/blob/master/samples/MeatSale.symboleo [Accessed on: 2022 Jun 05]

71. Venners B. The java virtual machine. Java and the Java virtual machine: definition, verification, validation 1998
72. Lehman J and Phelps S. "force majeure". *West's encyclopedia of American law*. Thomson/Gale, 2004. Available from: https://legal-dictionary.thefreedictionary.com/Force+Majeure [Accessed on: 2022 Jan 10]
73. Overleaf, Inc. Creating a document in Overleaf. 2022. Available from: https://www.overleaf.com/learn/how-to/Creating_a_document_in_Overleaf [Accessed on: 2022 May 26]
74. Smart NP. Computational Models: The Random Oracle Model. *Cryptography made simple*. Springer, 2016 :221–2. DOI: 10.1007/978-3-319-21936-3. Available from: https://link.springer.com/book/10.1007%2F978-3-319-21936-3
75. Smart NP. Hash Functions, Message Authentication Codes and Key Derivation Functions. *Cryptography made simple*. Springer, 2016 :271–8. DOI: 10.1007/978-3-319-21936-3. Available from: https://link.springer.com/book/10.1007%2F978-3-319-21936-3
76. Smart NP. Public Key Cryptography. *Cryptography made simple*. Springer, 2016 :202–3. DOI: 10.1007/978-3-319-21936-3. Available from: https://link.springer.com/book/10.1007%2F978-3-319-21936-3
77. Smart NP. Secure Digital Signatures. *Cryptography made simple*. Springer, 2016 :333–42. DOI: 10.1007/978-3-319-21936-3. Available from: https://link.springer.com/book/10.1007%2F978-3-319-21936-3
78. Adams C, Cain P, Pinkas D, and Zuccherato R. RFC3161: Internet X. 509 Public Key Infrastructure Time-Stamp Protocol (TSP). Tech. rep. RFC Editor, 2001. DOI: 10.17487/RFC3161. Available from: https://www.rfc-editor.org/info/rfc3161 [Accessed on: 2022 Jan 15]
79. Gipp B, Meuschke N, and Gernandt A. Decentralized Trusted Timestamping using the Crypto Currency Bitcoin. *Proceedings of the iConference 2015 (to appear)*. 2015. DOI: 10.5281/ZENODO.3547488. Available from: https://arxiv.org/abs/1502.04015 [Accessed on: 2021 Jan 15]
80. Gervais A. Transactions in Bitcoin. Principles of Distributed Ledgers, Lecture 1. Imperial College London, Deptartment Of Computing Lecture. 2022 Jan. Available from: https://materials.doc.ic.ac.uk/download/2122/70017/Lecture%20Notes/1 [Accessed on: 2022 Jan 18]
81. Merkle RC. Protocols for public key cryptosystems. *1980 IEEE Symposium on Security and Privacy*. 1980 :122–2. DOI: 10.1109/SP.1980.10006