Merging traditional contracts (or law) and (smart) e-contracts – a novel approach

Florian Idelberger
European University Institute
florian.idelberger@eui.eu

Abstract

For a long time, there have been various parallel developments in the area of AI and Law, Legaltech more recently or computable law, data exchange formats, e-contracts, agreement systems and similar. Some working more on creating normative or logical systems outside of traditional law such as business logic systems, others on bringing probabilistic approaches to reasoning about law or understanding human language, and as an extension, law written in that language. A particular important development are computable contracts, either through DSLs, understanding of natural language or repurposing of other languages such as logic-based languages. In this paper, the most important competitors for writing computable contracts that translate to the environment of a blockchain's normative environment will be examined in light of their usefulness for creating and merging contracts and e-contracts, functionality and legal looks. Championing the idea that the machine does not actually have to understand anything to be useful, as long as it can comprehensively translate, all the while keeping syntax and semantics intact, making for a more closely integrated contract stack.

1 Introduction

This work compares several kinds of programming languages that are used and developed for working in conjunction with or instead of traditional legal contracts, compared specifically when deployed upon a blockchain based smart contract system or a similar normative environment. Different approaches and methods are available, from simple computer code that in effect also has legal implications but isn't a contract as such and various bridging technologies and (such as SCTs (Wong, Meng 2018)) to the newest advances in writing contracts that have an automatic and accurate electronic representation. "Legal Tech" in various forms and different comprehensions has been all the rage for a while now, and still is. (Braegelmann, Breidenbach, and Glatz 2019) There has been tremendous progress in everything from probabilistic analysis of legal cases to templating systems and contract management systems, as well as the processing of large amounts of cases in specific niches, such as with flight delay claims and similar mass markets. This also brought renewed interest in general AI & Law/computable law as well as more specifically computable contracts. This work then focuses on computable contracts as understood by Surden (Surden 2012), represented by merging a traditional, natural language legal contract with a smart contract or other e-contract systems. The paper at hand will focus on contract law and specifically a licensing contract for software evaluation. This will be implemented and compared in three different systems. The comparison is made between a direct smart contract language (Solidity) (Christian Reitwiessner 2020), Prolog (SWI-Prolog 2020) as an alternative declarative, logic-based approach and the Lexon (Diedrich 2020) Smart Legal Compiler using an NLP-like approach and a smart contract templating system. Each of the approaches will be briefly introduced where after the implementation is introduced, and finally they will be compared and evaluated based on Surden's criteria and general legal and technological considerations defined more accurately later on. After the introduction, there will be a brief 'primer' on previous and existing technologies of AI & law and specifically e-contracting. Thereafter the method and case-selection employed here will be implored in more detail, explaining how implementations were made and tested. Then, the overall case and example license will be presented, where after each implementation is introduced, first by introducing the technology itself and then the implementation that was made. In the end, an analysis is conducted, and a conclusion drawn. After specially designed programming languages, legal interchange formats and different bridging technologies either with or without smart contracts, new natural-language-like systems present a new frontier for legal programming and e-contracting, merging traditional and electronic contracts.

2 Primer on Legal Technologies

For quite some time, various technologies for some kind of digitalization of law have been developed under different headings, such as 'ai & law', 'computable law' , 'computable contracts' , 'e-contracts' or 'smart contracts' , 'contract management', 'rule systems', or even very generally 'legal tech' which has in recent years become more and more popular with the rise of startups and law firms focusing on legal innovation and digitization.

2.1 Legal Markup

On a basic level of textual representation, there are the systems that structure legal documents, with some tailored towards legislative or judicial documents and others targeting contractual documents. This is covered by Legal XML

(OASIS 2008) / Akoma Ntoso (Akoma Ntoso 2015), LKIF (Boer, Winkels, and Vitali 2008) and their variants.

2.2 Logic-Based Languages

These are languages and system to reason about legal arguments or outcomes, where logic-based languages are especially prominent. Specifically, Defeasible logic (Nute 1993; Garcia and Simari 2002) was developed as an extension to simple argumentative logic. More widespread and complete languages like Prolog (SWI-Prolog 2020) and ReasonML (ReasonML 2020) can be used for similar purposes, such as creating computable law and contracts and reasoning about legal cases. Part of logic-based languages, expert systems (a precursor of what today is known as 'AI' or 'ML') based on these (or expert systems based on other languages) are ontologies and knowledge bases. Ontologies are 'explicit format specifications of the terms in the domain and the relations among them' (Natalya F. Noy and Deborah L. McGuinness 2001) and a set of instantiated ontological classes constitutes a knowledge base that can be used by logic-based languages or more generally expert systems.

2.3 E-Contract Systems

In particular, different evolutions of e-contracting systems can be distinguished. Primarily, there are contract management systems and e-contracting technologies as such – where the primary aim is merging or integrating legal contractual form, format and structure with some form of technological function and structure. This can be either for just making contracts more machine readable by enhancing them with markup, or all the way up to integrating logic into a document, then automating monitoring and compliance. In this paper, the classification specified by Meng Wong (Wong, Meng 2018) will be employed, where 1st Gen systems are older, proprietary document management and business automation systems, Second Generation system tend to be newer and more open-source, with many 2.5 Gen Systems having a relation to blockchain based smart contracts and more automation and logic than older systems. Third generation systems are computable law systems such as the CDL/compk by Stanford (unpublished) (Stanford University 2015) and perhaps Lexon.

2.4 Metadata Systems

A further, also very important class are meta data formats and systems that process metadata. This is especially very widely used in the area of copyright. Famously SPDX (Odence, Lamons, and Lovejoy 2013) set up a system for open source software packages of all kinds to specify their license, so that licenses can be processed and assessed for compatibility automatically. This has also been adapted or at least been written about to be used in a similar way for other forms of intellectual property, such as in "Computable Law" where Surden writes about SPDX as a way to implement more specific licenses and DRM. (Surden 2012)

2.5 "Legal Tech"

Finally, practical, modern, commercial examples of 'Legal Tech' are companies such as FlightRight (claims for compensation of delayed flights), WenigerMiete.de (calculating if you are paying too much rent for your area and helping with claims against that). (Ambrogi 2017)

3 What's legal about code? - The pyramid of norms

One topic that frequently comes up, and has been discussed numerous times is, if a 'smart contract' or another 'e-contract' is even a contract in the legal sense. Numerous interpretations narrations abound in literature from 'of course not, nothing legal about it' to 'it will run everything and become sky net'. Fortunately, more nuanced and pragmatic interpretations exist as well. For the present work, the definitions by Allen (Allen 2018), also used by Cohney/Hoffmann (Cohney and Hoffman 2020) are used, as they employ a comprehensive understanding of contractual agreement, focusing on agreement and deriving that from a comprehensive 'contract stack' which incorporates the textual representation with the layers of natural language and legal language, the program code and many other utterances that potentially signify intent such as tweets and marketing communication. Whether these are part of the relevant contract stack and are part of defining the intent depends on the specific circumstances and would ultimately have to be decided on a case by case basis. As contracts never exist in limbo or a closed, encapsulated system, this is a solid approach to understanding the nature of smart contracts specifically and e-contracts generally. The term contract stack is especially fitting, as it can similarly be used to describe traditional contracts that do not have an executable part, as their interpretation in case of doubt also is sought elsewhere. It is also very similar to the 'legislative stack' where communications and especially remarks in preparatory documents are used to gage intent. While there is no clear overarching hierarchy in a contract stack, it can be derived in a specific case, such as that text and executable code are probably always higher in rank than intent signified by marketing communication. For the present work, this analogy is important as it frees the later analysis of the specific decision whether a particular instance of textual contract, executable code and other communication or a part of those constitutes a legal contract at a given or all points in time, in a given or all jurisdictions, as the only important conclusion is – it can – in principle.

4 Method

In this section, the method and case selection will be described, meaning how (and why that way) technological solutions were compared and how those that were chosen were selected. To evaluate the suitability of techniques and compilers this work builds on previous work where a simple license for evaluation of a product was described in natural legal language, then implemented with Solidity and in defeasible logic. In that case, there was a much more detailed investigation of contract theory, and in the end a focus on the benefits of declarative specification. (Governatori et al. 2018) However, it is assumed that because all relevant judicial jurisdictions allow many different forms of signifying contractual intent and agreement, that the model of the 'con-

tractual stack' by Allen (Allen 2018) holds, and thus while there are certainly differences, these can be disregarded for this exercise. From a legal point of view, depending on jurisdiction there is a definite distinction between license and contract, especially in the United States. In Germany on the other hand for example, a license is just a different type of contract. For the example presented here, no specific jurisdiction is assumed, although a contractual nature of the license is presumed. (Andres Guadamuz and Andrew Rens 2013)

Compared to the past work, the license example is extended to make for a more realistic testcase. Most importantly, it was extended to include an actual license grant, header with all the participants and a licensing and breach fee. Additionally, to make it more suitable to an e-contract, an arbiter was added so that in case of a dispute, an external party can judge the facts. Finally, the ability to sublicense was added to mirror the licensing example used by Surden. (Surden 2012) These additions add a minimal increase in size, as is relevant for publication, but maximize the usefulness and completeness of the license example. After a description and explanation of the license at hand and more detailed explanation of its legal nature, the implementations will be presented, such that first the technology itself is described for general understanding and analysed as to its relevance and its specific features that are employed for the task at hand, after which the relevant implementation is presented. To start, the license example is implemented in a current version of solidity. (Christian Reitwiessner 2020) This programming language was chosen because it is still the primary programming language for Ethereum smart contracts (and some others). Secondly, the license example is implemented in Prolog, (SWI-Prolog 2020) a logic based language popular for knowledge bases, natural language work and pseudo-code. This was chosen as opposed to Defeasible Logic (Governatori et al. 2018) or another logic notation as it is much more popular, more easily accessible and reproducible and last but not least with SWISH/SWI-Prolog as a public implementation, the code can easily be tested and run.

Finally, the same license example is implemented in Lexon (Lexon Foundation 2020), a next-gen legal compiler merging legal contracts with smart contracts – without losing NLP-like semantics and information of natural language, while gaining substantial automated powers. This compiler was chosen because it is currently at the forefront of merging legal contracts and e-contracts. It combines the clarity of declarative systems with the possibilities of smart-contracts and thus is great to differentiate against Prolog and Solidity. Finally, in the analysis, the different systems and technologies are contrasted in more detail, based on the criteria by Surden and Cohney/Hoffman. (Cohney and Hoffman 2020)

5 Test Case

A textual test case of a license contract is put forward, that is then used to evaluate against all the technological implementations later on. It is presented in a similar way as the implementations but focuses more on legal aspects.

5.1 Textual "Technology"

The most common and ordinary use of contracts is still by ordinary language, written or spoken words in a given natural language that can be enriched with legal jargon and terms specific to a particular field or industry. The 'medium' is often a piece of paper, but one particular feature of the law is that its medium or articulation are of lesser importance unless otherwise prescribed.

5.2 Test Case

Example 1. *This license is an example evaluation license.*

LICENSEE - The University

SUBLICENSEE - A student as set force in the appendix. (array of persons)

ARBITER - An arbiter or oracle that decides in case of disputes. Can be a natural or legal person or a machine. Art. 2, 3 and 4 especially are evaluated by the arbiter in case of disputes.

ASSET X - An asset to be licensed.

Article 1. *The Licensor grants the Licensee a license to use and evaluate asset X and grant sublicenses among group Y, for use and evaluation. This grant is in exchange for a licensing fee.*

Article 2. *(optional) The (Sub)Licensee is commissioned to publish comments about the use of the product. This allows publication of comments but also requires them.*

Article 3. *The (Sub)Licensee must not publish comments of the use and evaluation of the Product without the approval of the Licensor; the approval must be obtained before the publication. If the Licensee publishes results of the evaluation of the Product without approval from the Licensor, the Licensee has 24 h to remove the material.*

Article 4. *Article 4. This license terminates automatically if the (Sub)Licensee breaches this Agreement. Breach obliges the licensee to pay a fee to Licensor for Breach of the Licensing Terms.*

The test case is a license contract to license a copy of a software or other specified work for use and evaluation, in exchange for a licensing fee. Furthermore, sublicensees can be specified. These grants and license are defined in article 1. The sublicense part was inspired by Surden's description of a licensing system where universities can automatically manage the licenses of their libraries and conclude more tailored licensing agreements. In article 2, it is defined that optionally, the licensee or sublicensee is commissioned to publish comments about the use of the product. This approves publication, but also requires it. In article 3, publishing of comments about the use and evaluation of the asset without approval by the licensor beforehand is prohibited. In case of unauthorized publication, the licensee has 24 hours to remove the published material. This improves the test case, as it requires use of external agents or data sources depending on the system, as otherwise there is no basis on which to automate or act. Additionally, the passing of time is tested

with the time limit. Finally, in article 4, it is laid out that the license agreement terminates automatically upon breach of the licensing agreement, and that the licensee has to pay a fee in case of breach. The resulting license agreement is simplified for publication, but includes the most important features of license contracts.

Compared to the previous work, this example adds important features that make it an actual license. The asset that is licensed is defined and the relation between commenting and publication and how this relates to the possibility of the licensee being commissioned by the licensor is clarified. In reality however, this would still be defined differently, with likely a statement of intent, the spirit of the license in the beginning and much more detailed definitions. Last, it was not clearly defined when the license was breached or would be terminated, so that was improved. In order not to make it too hard on the example implementations, some care is taken that the licensing contract does not constitute or seem like an impossibility. To this end, it was evaluated on the basis of Surden's criteria for computable contracts. (Surden 2012) According to Surden, most of all a computable contract needs some kind of data source, to assess performance, compliance or breach, which then lead to lower transactions costs and increased reliability. Additionally, the computer or the contract on a computer needs to be told what to do, which Surden calls an 'automated prima facie assessment.' In the example, the main points in this regard are the granting of the license itself, the determination of when or how an evaluation and or comments were published, if they were removed in time and finally whether or a given set of facts with regard to these licensing obligations constitutes a breach of contract or not. While parts of this can be automated, in such a general textual form, not all of it can. Thus, to give the computable contracts a fighting chance, an arbiter is introduced to help make determinations. The arbiter is a natural or legal person or possibly also a third-party software agent that can make determinations about facts in case of disagreements. The resulting test case license is still imperfect, but an actually used license in natural, legal language probably could fill a whole paper on its own. In Surden's case with SPDX (Odence, Lamons, and Lovejoy 2013), this license is relatively well suited for computable contracting, as much more fine grained licensing is possible, thanks to being able to define (sub)licensees based on criteria of the student population, clear delineation of the asset being licensed and the parties involved, with a limited number of happy states and failure states. It is furthermore assumed that the actual object of the license is described either in the license, by a graphical interface or extracted from business logic systems. Thus, (sub)licensees are easily accessible, and license status and publishing obligation or permit are available.

In the spirit of Surden's assessment, this presents a good starting point for our experiment of examining different license implementations, as he claims computable contracting is limited by possibilities of automated assessments, but which he claims goes 'far enough' more often than not.

5.3 The criteria of examination

In earlier work the focus of comparison was on detailed legal theoretical comparison, describing the contractual theory of contracts as 'legally binding agreements' with the core tenants of the agreement of the parties, consideration in exchange for something, the competence and the capacity of the parties involved and the legal object and purpose of the agreement. (Governatori et al. 2018) For the present analysis, the focus is on whether the test case and its implementation capture the meaning and spirit of the contract (as well as syntax and semantics), are functional and most of all still 'look like a contract' and can be read and understood by legal professionals and others alike. Now, with the stage set and the criteria explained, let's meet the candidates of this event.

6 The implementations

The e-contracting implementations are presented and assessed. While they are assessed on the basis of a smart contract system, they could also be implemented to work with another e-contracting systems or more general normative environments.

6.1 Solidity

As an example of an imperative language that is used to program 'e-contracts' Solidity is used, which was the first higher level language for such scripts in the blockchain space, and is still dominant at the time of writing.

Technology The solidity language was established by the Ethereum project. (Gavin Wood 2013) It is an imperative language that compiles down to bytecode that is run in the Ethereum Virtual Machine (EVM), borrowing somewhat from JavaScript in syntax, but otherwise being statically typed and otherwise constrained due to the execution environment on the blockchain. This is due to the fact that as the code is necessarily computed by miners when proposing new blocks for the blockchain, it has to be ensured that the output is always the same given the same input and infinite loops or DDOS attacks are to be avoided.This necessitates strict typing, (relatively) fixed arrays and very strict loops that do not run an indeterminate amount of time. For this, each execution step is priced in terms of 'gas' in the EVM, which puts an extra cost on complex programs, but especially on storage. The language is relatively readable, but only in so far as traditional programming languages go. The gas system is imperfect and flawed, but the best solution available so far, using economic forces to secure its network.

Implementation This is an implementation of the license agreement in solidity, with a current version of solidity as of this writing. A lot of space is captured by the simple definition of basic variables and their types. These are necessary for functionality at one hand, and for EVM execution on the other hand, as a system with execution constraints such as the Ethereum Smart Contract system needs this information to price execution and storage. There is also a certain amount of inherent duplication – f.e. it can be seen that variables established for the contract have to be passed again to

the constructor whereas these variables are necessary for the solidity language, they do not serve any purpose in conveying the legal or natural meaning of a contract with regard to the agreement and the 'spirit'. However, it can at least be said that a properly written smart contract in an imperative language such as Solidity does achieve a lot of functionality. For the functionality, it can f.e. be held that important events such as contract creation, termination, mediation (via arbiter), performance and modification as well as monitoring can be automated, or at least made partially functional, using such a language. This might seem obvious, as that is the purpose of the language, but it will become visible that in the comparison performed that is a very relevant distinction. Finally, it is fair to say that no normal legal professional or layperson would even begin to understand the meaning of such a contract or script. Even for some programmers, or smart contract developers even, it can be very hard to fully understand the meaning and all nuances of smart contracts, as has repeatedly held in previous literature.

```
pragma solidity ^0.6.7;
pragma experimental ABIEncoderV2;
contract LicenseContract {
    bytes32 work_hash; string name; address
        payable licensor; address licensee;
        address payable arbiter; bool
        breachFeePaid;
event Status(LicenseProps);
struct LicenseProps {
    int id; int licenseFee; uint breachFee;
        boool isCommissioned; bool
        publicationIsApproved; bool
        requiresComments; uint timeToRemove;
         bool triggeredTimeToRemove; bool
        licenseBreached;}
mapping (uint => LicenseProps) licenses;
uint[] public licensesList;
uint numLicenses;
modifier onlyBy(address _account)
    {require(msg.sender == _account,
          "Sender not authorized."); _;}
function newLicense(int _id, int _licenseFee
    , uint _breachFee, bool _isCommissioned,
     bool _publicationIsApproved, bool
    _requiresComments, uint _timeToRemove,
    bool _triggeredTimeToRemove, bool
    _licenseBreached) public returns (uint
    licenseID) {
        licenses[licenseID] = LicenseProps(
            _id, _licenseFee, _breachFee,
            _isCommissioned,
            _publicationIsApproved,
            _requiresComments, _timeToRemove
            , _triggeredTimeToRemove,
            _licenseBreached);
        licenseID = numLicenses++;
        return licenseID; }
function commissionComments(uint _licenseID)
     public {
    LicenseProps storage l = licenses[
        _licenseID];
    l.publicationIsApproved = true;
    l.requiresComments = true;
  l.isCommissioned = true; }
function grantApproval(uint _licenseID)
    public {
    LicenseProps storage l = licenses[
        _licenseID];
    l.publicationIsApproved = true; }
function evalPublication(uint _licenseID,
    bool isPublished) public {
    LicenseProps storage l = licenses[
        _licenseID];
    if (isPublished) {
        if ((isPublished && !l.
            isCommissioned || !l.
            publicationIsApproved) && !l.
            triggeredTimeToRemove) {
            l.timeToRemove = now;
            l.triggeredTimeToRemove = true;
        } else if (l.triggeredTimeToRemove
            && now > l.timeToRemove + 1 days
             ) {
            declareBreach(_licenseID);
        }    } emit Status(l); }
function declareRemoved(uint _licenseID)
    public onlyBy(arbiter) {
    LicenseProps storage l = licenses[
        _licenseID];
    l.timeToRemove = 0;
    l.triggeredTimeToRemove = false; }
function declareBreach(uint _licenseID)
    public onlyBy(arbiter) {
    LicenseProps storage l = licenses[
        _licenseID];
  l.licenseBreached = true;
  if (!breachFeePaid) {
      breachFeePaid = false;
      licensor.transfer(l.breachFee);
      }     } }
```

Listing 1: This is the Solidity Implementation of the License.

Further, it is on one hand more prone to errors, and on the other hand much more verbose as the format is much more visible. In an actual blockchain use case, especially on a system with such economic incentives, it would probably also make sense to implement a lot of actual evaluation off-chain, because as long as license data is on-chain, it can be safely checked off-chain by 'everybody'. Still, this would necessitate some other connection between legal contract, structure and semantics and the functional transactional scripts. Thus, ideally, independent of characterization of 'smart contracts', there would be a stronger link between the legal part of a smart contract and the functional parts of it. The surrounding environment anyway has to be adapted to contract management systems or end user needs, but a human readable contract that captures the contractual essence is in both cases an asset.

6.2 Logic-based languages (Prolog)

Whereas the previous work (Governatori et al. 2018) focused more on legal lifecycle and logic-based languages as such, with a strong focus on Defeasible Logic because its built-in 'defeasibility' lends itself well to contracts that have no 'undefined' states, the present work uses Prolog due to

accessibility and availability, ease of replication and suitability due to great previous work testing the limits of Prolog for legal reasoning and logic.

Technology Prolog is here used as an example for various kinds of attempts at using primarily logic-based languages for contract like systems or generally reasoning about law and contracts. Reasoning is here understood as making deductions based on a set of inputs. Furthermore, the syntax and structure of Prolog is more easily acquired and tested.

While Prolog is not as tailored to be defeasible and can in principle have unclear states, it otherwise supports everything that DL supports and even much more. (Morelli, Ralph 2011)

The main parts of Prolog used in this research are facts 'exists(paper).', which means paper exists, rules denoted by ':-' which evaluate the left statement to true if the conditions on the right hand side below match, and variable, denoted by every term starting with a capital letter. As the Prolog interpreter, similar to a DL implementation does by default not know anything, everything has to be provided, which also means that some basics are just facts that are assumed to be true or set by the creator as true, as otherwise there would be no starting point. This is opposed to a blockchain based system or many procedural languages, where for example an identity structure or similar might already be implemented. Overall, it can be concluded that Prolog is very useful for sketching logic and processing workflows and can easily be accessed. Care should be taken to make sure the supplied answers make sense and the names or words used for naming variables and facts actually make sense, as compared to imperative programming, the supplied facts and rules are not merely variables but actually make up the knowledge base of the project, such that badly named, too short or too long items can render the whole project hard to read or even useless. For actual implementation however it would probably be seldom used, as too much would have to be written around it. That could be different with more tailored variants such as DR-Prolog (Antoniou and Bikakis 2007), LogicalContracts (Kowalski, Calejo, and Sadri 2018) or if translation to an imperative language becomes possible, that can directly be deployed in a normative system where its facts take on deeper meaning by being connected to identities, values and assets. Thus, at the very least for testing, Prolog and similar languages are very relevant, though it would help to have a system on how to set facts, verbs and rules, as otherwise inputting knowledge can seem arbitrary.

Implementation The Prolog implementation of the textual license example mentioned above can be found below. First, similar to the preamble of the textual License, some basic facts are laid down, such as the natural or legal persons that are a part of the license, if the Licensor commissioned the Licensee and if the comments of the evaluation were published or not and how long ago that was.

In article 1, the licensee is granted a license by the licensor, and it is checked if both parties exist, whether a fee was paid and if there actually was an offer of a license. If either is not true, there is no license. Furthermore, if there is a license, sublicenses are allowed, and it is defined that the licensee may publish comments in case there is a license and the licensor approved the comments. Afterwards, in article 2 it is defined that the licensee counts as commissioned in case the licensor commissioned them, and that if they are counting as commissioned, they also must publish the comments. Thirdly, in article 3, the prohibitions are laid down, whereto the licensee may not publish comments if they were not approved and that they must remove them if they do it anyway and with an extra check of the time, so that this obligation to remove disappears in case it is complied with. Finally, in article 4, it is established that the license terminates if there is a breach by the licensee, and that the licensee has to pay a fee in case of a breach.

The biggest issue with pure logic-based examples is mainly their lack of grey areas and how they learn about facts. How these facts get there – also a big issue with smart contracts in general and solidity in particular – is not solved, so the great properties of a logic-based implementation are always limited by its surroundings. Thus, this only solves the actual logic. The actual input/output still would have to come from somewhere else, an additional layer of implementation for any logic-based use, where errors can be introduced and part of the advantage of the logic-based approach gets lost in the process. Declarative languages used for functionality could help, however it is uncertain if that will really help, as truly self-contained languages are probably too limited, any introduction of uncertainties will make the logic-based approach also more error prone

```
person(arbiter).
person(licensor).
person(licensee).
person(sublicensee).
commissioned(Licensor).
publish(time, 24).
licenses(Licensor, Licensee) :-
    %grants_license(licensor, asset,
        licensee):
    exists(licensor).
  exists(licensee).
  paid_fee(Licensor, Licensee).
  made_offer(Licensor, Licensee).
allow_sublicense(Licensor, Sublicensee) :-
    licenses(Licensor, Licensee).
may_publish(Licensee) :-
    licenses(Licensor, Licensee).
  approves_comments(Licensor).
is_commissioned(Licensee) :-
    commissioned(Licensor).
must_publish(Licensee) :-
    is_commissioned(Licensee).
not(publish(Licensee)):-
    not(may_publish).
  not(approves_comments(Licensor)).
removed_comments(Licensee) :-
    Removal_time > 24.
removed_comments(Licensee) :-
    Removal_time < 24, not(must_remove(
        Licensee, Published)).
license_canceled(Licensee) :-
    breaches(Licensee).
pay_fee(Licensee, Licensor) :-
```

```
breaches(Licensee).
```

Listing 2: This is the prolog logic version.

For logic-based approaches, they can capture most of the meaning and spirit of the contract, however it will only be accessible by experts. Even if it is more precise than imperative programming, languages such as DL and Prolog are still too arcane to fully capture a contract and make it accessible. Partially, this is because the intricacies of syntax and semantics of natural language are not captured in the kind of logic-based 'contract' compared here. The resulting work is functional, in so far as questions can be asked of it, it can be resolved for facts and answers like an equation, and it can be automated and supplied with data in much of the way that Surden envisions. Still, it does not really 'look' like a contract and as said in the beginning would need some connection to a different textual version of the contract to be accessible thereby negating most benefits.

In what Wong calls the 2nd generation of Smart Conctract Templating Systems and their blockchain brethren in the 2.5 generation, no winner has emerged yet, even of those that are still alive. Due to space constraints, this discussion is omitted here.

6.3 Lexon

Finally, Lexon is described as a primary example of a new kind of 'script' specific to e-contracts, developed specifically for merging e-contracts and legal contracts, while being easy to write and read for humans without requiring huge computing resources.

Technology Primarily, Lexon (Diedrich 2020) resembles the structure of legal contracts, and uses similar language. It is already functional but at the time of writing also still under heavy development. The defining unique characteristic is that it uses direct translation of natural language and legal prose to executable code via an Abstract Syntax Tree, without a secondary intermediary layer. This keeps the structure and syntax and higher order meaning of the original document much better. While Lexon documents require a certain structure, it is not foreign to people nor lawyers and very natural language like. So far, it comes closest to a 3rd generation SCT system, according to Wong's definition of the main characteristic being a DSL (Wong, Meng 2018), although with the aim to go even further.

Each document starts with a title, mentions the Lexon version it is built for, and can offer some explanation or other guidance in a preamble, similar to a normal legal contract. In a further section, terms are defined, such as the parties to the contract, amounts and fees and what their role is. Then, the main contract starts via 'CONTRACT(S)'. The contract is then divided into 'clauses' that roughly match onto functions in a solidity coded contract.

This example demonstrates, that first there is the CLAUSE keyword and that Lexon still partly relies on keywords, similar to other, imperative programming languages. However, it was developed in such a way that its keywords are as close as possible to legal language used in ordinary contracts, and its actual content is legible for programmers, legal professionals and everybody else alike. So far, all the 'actionable' words are mapped to terms and phrases in the solidity smart contract language, however in principle they could also map to other normative systems, such as business rule systems.

Implementation Below is an implementation of the license example in Lexon. At the beginning, the header with the name of the contract, the software version and the preamble are shown. Preamble is a non-actionable part of the script but might be very relevant in case of a dispute as earlier literature argued that in most cases there is always a 'contract stack' consisting of multiple expressions of the contract, such as the 'smart contract' as a functional layer and others that could help with interpretation, particularly with finding out the intent of the agreement and the parties involved. In the next section, the terms are fixed, defining who licensor and arbiter are, the fees are established and set and an arbiter is appointed. Then, the actual contract is defined, in this case 'per licensee' so that it is clear that there is one contract per licensee. This is then also deployed like that later, in that executable code is instantiated with actual amounts, assets and identities per licensee, and certain properties like "Permission to Comment" are established, and most importantly, a description or link to an identifying token for the licensed assets are established. This could be just a boolean, a license key or a hash of the licensed work for example. If and how this would be checked is another matter, in case of software it could be checked by the software itself, in cases of artworks it could be checked via a DAPP or centralized website for such work. For all content of the license articles that have an actually "actionable" function, meaning it can at least to some degree be recorded, automated or delegated to the arbiter, there is a 'CLAUSE' that signifies that. When compiled, these would be mapped to solidity 'functions', and are also mentally much closer to a kind of 'declarative function' then clauses, as they are understood by legal professionals. This is also one of the main negative points, that while the resulting code is much closer to ordinary contractual language, still certain keywords and a certain structure should be used. However, it should also be noted that these depend a lot on the underlying normative system that they compile to, meaning as the underlying system uses 'functions' (in Solidity), Lexon needs to map to them. In a different normative system, such as a business rule system, this could be organized differently.

```
LEX: Evaluation License System.
LEXON: 0.2.12
PREAMBLE: This is a licensing contract for a
    software evaluation.
TERMS:
"Licensor" is [ a person ].
"Arbiter" is [ a person ].
"License" is this contract.
"Licensing Fee" is [ an amount ].
"Breach Fee" is [ an amount ].
The Licensor appoints the Arbiter,
fixes the Licensing Fee,
and fixes the Breach Fee.
CONTRACTS per Licensee:
```

```
"Description of Goods" is [ a text ].
"Licensee" is [ a person ].
"Paid" is [ a binary ].
"Commissioned" is [ a binary ].
"Comment Text" is [ a text ].
"Published" is [ a binary ].
"Permission to Comment" is [ a binary ].
The Licensor fixes the Description of Goods.
CLAUSE: Pay.
The Licensee pays the Licensing Fee to the
    Licensor,
and the Breach Fee into escrow.
The License is therefore Paid.
CLAUSE: Commission.
The Licensor may certify this License as
    Commissioned.
CLAUSE: Comment.
The Licensee may register a Comment Text.
CLAUSE: Publication.
The Licensee may certify the License as
    Published.
CLAUSE: Grant Permission to Comment.
The Licensee may grant the Permission to
    Comment.
CLAUSE: Examine for Breach.
The Arbiter may Declare Breach, if:
the License is Commissioned and no Comment
    Text is set;
or the License is Published and there is no
    Permission to Comment.
CLAUSE: Declare Breach.
The Arbiter may pay the Breach Fee to the
    Licensor,
and afterwards terminate this License.
```

Listing 3: This is the Lexon version.

As can be seen, the license as implemented in Lexon is relatively close to the meaning and spirit of the contract, though it still needs a certain structure. The syntax and semantics are close enough to natural language to be readable and require little or no training to grasp meaning and spirit of the contract. Furthermore, the result is as functional as any 'smart contract' of the underlying system. Most importantly however, the Lexon contract still mostly looks and feels like a contract (and could be made to look more natural as well, i.e. fill words and similar do not bother the compiler) so that it can be read and understood by most, but especially also by legal professionals without programming or other in depth knowledge. Thus, the Lexon stack is a at least for now quite promising and the most advanced digital contract stack available, being a proper 3rd gen SCT system although it remains to be seen if it will continue to prosper or stagnate and remain an interesting experiment.

7 Findings

The investigation started with an example of a software licensing contract, which at one hand includes things helpful for electronic contracts such as a provision for an arbiter, but on the other hand also features time and vague uncertain provisions that require such a change and highlight the difficulties in understanding, interpreting and enforcing contracts, not only but especially when in a restricted environment such as that of a smart contract language. If we then compare this to solidity, as representing an imperative, somewhat high-level language, stark differences emerge, as the contract as such is not representable in such a language. Still, even a piece of software, including its documentation, git commit logs and informal communication can be seen as a contract, as supported by Allen. (Allen 2018) The solidity example explicitly describes the actions that the smart contract should support, with a lot of technical jargon and even experienced programmers still frequently get it wrong. This does not mean that Solidity doesn't have its uses, however it has to be taken into account. It is also observable, that there is a lot of repetition in the smart contract, such that not only does a variable have to be initialized, but then it has to be passed to each function and the constructor as well, causing a lot of extra text especially when trying to use descriptive variables and function names. An additional difficulty could arise due to scoping problems – meaning that like in other languages, variables have a local, global or possibly other scopes, such that variables can have different meaning and values depending on where they are. This could maybe be related to context in natural language, but there, people and legal professionals are used to it and there is context, in the context of a programming language it is just foreign and the context is more akin to two brackets instead of three. If defined in terms of the contract stack and our criteria established above, the solidity implementation gets full points in the functionality department, as everything that is possible in principle within the constraints of the Ethereum EVM is possible. It also at the very least has the possibility of the 'meaning and spirit' of a contract. In regard to syntax and semantics, a Solidity based contract does not really capture the nature of the original license contract, and neither does it 'look like a contract' to legal professionals or laypersons. Lastly, the contract is also not read nor understood by them. In the next section, Prolog was looked at, a widely available, accessible and extensible, logic-based language. First, also the Prolog example captures the meaning and spirit of the contract. The syntax and semantics are much more reasonable and readable, and it looks a lot more like natural text, still not like a contract though. However, a big problem is that in general, logic-based languages, including Prolog, offer a lot of flexibility, but without the flexibility and dictionary of natural language. As a result, much depends on the choice of terms and verbs, and on the understanding of the definition of facts, variables and functions/rules in Prolog. More specifically, facts and rules can be very detailed and specific and easy to read depending on how they are chosen, but can also be very useless, if cryptic or confusing terms are chosen. This makes it maybe even more confusing than an imperative programming language, as the differences are more subtle and a system for defining facts and rules would be necessary to have a certain predefined standard. Furthermore, a logic-based contract is 'solvable' in the sense that the interpreter can solve it based on input and determine possible or logical end states. However, by itself it does not have functionality. This means that to implement functionality, it would have to be embedded into

a compiler or other normative environment that translates it to its functionality, so that f.e. a licensee can be tied to something. This is certainly possible, but then this system would have to be assessed as well, as it might impact the assessment. Lastly, Prolog based contracts look more like a contract (f.e. because clauses can be separated more like articles than functions) than one based on an imperative language and is more easily understood by legal professionals and others, but only slightly so and very dependent on how terms and rules are named.Lastly, although not quite natural language, Lexon qualifies as the 'next best thing', assuming it can continue development and not fade away like so many others before it. While it has some of the drawbacks of logic-based and imperative languages – it does not look entirely natural, in that the way clauses are described, requires some change of thinking for non-programmers, and it is dependent on the target environment. However, these can still be easily understood when read and this will probably also improve with time or when used with another normative system as a foundation. Still, however, it resembles a contract in meaning and spirit, and captures a lot of the syntax and semantics of the original through not using an intermediary layer but direct mapping via abstract syntax tree. The contract has the same potential functionality as the underlying normative environment and mostly 'looks like a contract' and can be understood and read by legal professionals and others alike. At least in so far as other contracts are understood today. So, while the Lexon compiler and system does not actually understand anything, the idea is that it doesn't have to do that, in so far as syntax and semantics are kept close to the original, making for a more closely integrated contract stack.

8 Conclusion

In this paper, an example license was presented and implemented in three different tech stacks. The example license was inspired by previous work but now aligned along different guidelines. This license was implemented in Solidity, Prolog, and Lexon. The main characteristics for examination were 'meaning and spirit of the contract', syntax and semantics (of the original license), functionality and looks. Based on these factors, Lexon (barring other developments) is the most promising candidate right now, as it is at its core not too complicated and not reliant on things like statistic-based AI, but at the same time still very much still looks like natural language and is also very functional at the same time.

References

Akoma Ntoso. 2015. Akoma Ntoso - What is it? https://archive.vn/wip/cqOGp.

Allen, J. G. 2018. Wrapped and Stacked: 'Smart Contracts' and the Interaction of Natural and Formal Language. *European Review of Contract Law* 14(4):307–343.

Ambrogi, R. J. 2017. A Golden Age of Legal Tech Start-Ups. *Law Practice* 43(2):34–41.

Andres Guadamuz, and Andrew Rens. 2013. Comparative analysis of copyright assignment and licence formalities for Open Source Contributor Agreements. *SCRIPTed*.

Antoniou, G., and Bikakis, A. 2007. DR-Prolog: A System for Defeasible Reasoning with Rules and Ontologies on the Semantic Web. *IEEE Transaction on Knowledge and Data Engineering* 19(2):233–245.

Boer, A.; Winkels, R.; and Vitali, F. 2008. MetaLex XML and the Legal Knowledge Interchange Format. In *Lecture Notes in Computer Science*. Springer. 21–41.

Braegelmann, T.; Breidenbach, S.; and Glatz, F. 2019. *Rechtshandbuch Legal Tech*. München: C.H. Beck, 2. auflage edition.

Christian Reitwiessner, S. D. 2020. Ethereum/solidity. ethereum.

Cohney, S., and Hoffman, D. 2020. Transactional Scripts in Contract Stacks. *Faculty Scholarship at Penn Law*.

Diedrich, H. 2020. *Lexon Bible: Hitchhiker's Guide to Digital Contracts*. Independently published.

Garcia, A., and Simari, G. 2002. Defeasible Logic Programming: An Argumentative Approach. *Theory and Practice of Logic Programming*.

Gavin Wood. 2013. Ethereum: A secure decentralised generalised Transaction Ledger. *gavwood.com*.

Governatori, G.; Idelberger, F.; Milosevic, Z.; Riveret, R.; Sartor, G.; and Xu, X. 2018. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law* 26(4):377–409.

Kowalski, R.; Calejo, M.; and Sadri, F. 2018. Logic and Smart Contracts.

Lexon Foundation. 2020. Lexon Tech. http://lexon.tech/.

Morelli, Ralph. 2011. PROLOG Facts, Rules and Queries. https://archive.vn/0O3K9.

Natalya F. Noy, and Deborah L. McGuinness. 2001. Ontology Development 101: A Guide to Creating Your First Ontology. https://archive.vn/015LB.

Nute, D. 1993. Defeasible Logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3 of *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press. 353–395.

OASIS. 2008. Legal XML. http://www.legalxml.org/.

Odence, P.; Lamons, S.; and Lovejoy, J. 2013. Advancing the Software Package Data Exchange: An Update on SPDX. *International Free and Open Source Software Law Review* 5:145.

ReasonML. 2020. Reason. https://reasonml.github.io/.

Stanford University. 2015. Computable Contracts Project. http://compk.stanford.edu/.

Surden, H. 2012. Computable Contracts. *University of California Davis Law Review* 46:629–700.

SWI-Prolog. 2020. SWI-Prolog – Manual. https://archive.vn/wip/L1MRC.

Wong, Meng. 2018. Computable Contracts: From Academia to Industry. In *Rechtshandbuch Legal Tech*. C.H. Beck, 1st edition edition.