Towards an Executable Methodology for the Formalization of Legal Texts

Tomer Libal[1] [0000−0003−3261−0180] and Alexander Steen[2] [0000−0001−8781−9462]

[1] American University of Paris
[2] University of Luxembourg

Abstract. A methodology for the formalization of legal texts is presented. This methodology is based on features of the NAI Suite, a recently developed formalization environment for legal texts. The ability of the tool to execute queries is used in order to drive a correct formalization until all queries are validated. The approach is studied on a fragment of the *Smoking Prohibition (Children in Motor Vehicles) (Scotland) Act 2016* of the Scottish Parliament.

Keywords: Legal Reasoning · Deontic Logic · Automated Reasoning

1 Introduction

The generation and maintenance of knowledge bases as a formalized representation of domain specific information is a well-established approach for enabling the employment of automated procedures that process this information in a suitable way. In the context of Computational Law, knowledge bases may act as large repositories of interpretations of legal documents and therein contained norms, for the purpose of providing semantic access to them, e.g., for employment in legal drafting, compliance checking, and legal reasoning. While earlier research in Legal Informatics focused on structured document representations and information retrieval, e.g. [3], more recent work also addresses the logical structure of the legal documents' semantical content [13]. This structure is thereby captured by logical rules of some adequate logical formalism which describe the contained obligations, permissions, prohibitions, etc. and may then, in conjunction with a concrete state of affairs, be used to calculate the legal consequences with respect to the given normative document using a deductive reasoning procedure.

In this paper, we follow the idea of using (semi-)automated reasoning technology for legal norms, but focus on the validation of legal knowledge bases themselves. Usually knowledge bases are engineered by IT professionals since quite some expertise about its underlying technical details, e.g. knowledge about the computer-readable input format, is necessary. How can we make sure that the representation of the legal norms actually captures the intended meaning? The knowledge engineering can of course include erroneous inputs because there is usually only limited domain knowledge available. Regardless of the domain expertise, general errors and inaccuracies may, of course, occur in any case. State-of-the-art methodologies for building legal ontologies [8] and for validating formal representations of legal texts [1] rely on communication between domain experts and IT experts for ensuring correctness. In the approach of Bartolini et al., the knowledge is modeled by IT experts and then translated algorithmically to a natural language representation. The latter representation is then given to domain experts who assess its correctness. Potential errors or problems are then reported back to the IT experts which try to accommodate the feedback. This whole process is repeated until certain quality criteria are met. Of course, communication between different domains is error-prone and laborious; the natural language translation result is still quite complex and might hinder the proper assessment of the data.

In this paper we describe ongoing work towards a methodology that, intuitively speaking, treats the creation of legal knowledge bases as a domain-specific agile software engineering process. The methodology aims at enabling non-technical domain experts, here legal professionals, to control the knowledge engineering process by using a graphical and interactive interface that uses automated reasoning technology for providing real-time assessments of the given inputs. The methodological approach in this paper is prototypically implemented in the new normative reasoning framework NAI. NAI features a graphical annotation-based editor which abstracts from the underlying logical language of the knowledge base. It also incorporates easily accessible functionality for assessing the quality requirements of the presented methodology, including consistency, non-redundancy and functional correctness.

Additionally, the architecture of NAI is modular in the sense that it allows using different logics and reasoning engines that seem fit for the task at hand. It also provides an API, which can be used by other tools in order to reason over the formalized legislation. NAI is a web application and is readily available at `https://nai.uni.lu`. It is open-source and its source code is freely available at GitHub[3] under GPL-3.0 license.

The contributions of the paper are: A description of a new agile methodology inspired by behaviour-driven Development (BDD) for the creation and validation of legal knowledge bases. Furthermore, we show how the NAI tool can be used to implement this methodology by exemplarily formalizing a fragment of a concrete legal document and testing the resulting knowledge basis for correctness.

2 Preliminaries

The logical formalism underlying the NAI framework is based on a universal fragment first-order variant of the deontic logic **DL*** [7], denoted $\mathbf{DL*}_1$. Its syntax is given by

Definition 1 (Syntax of $\mathbf{DL*}_1$). *Let V, P and F be disjoint sets of symbols for variables, predicate symbols (of some arity) and function symbols (of some arity), respectively. $\mathbf{DL*}_1$ formulas ϕ, ψ are given by:*

$$\begin{aligned}\phi, \psi ::= &\; p(t_1, \ldots, t_n) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \\ &\mid \mathsf{Id}\,\phi \mid \mathsf{Ob}\,\phi \mid \mathsf{Pm}\,\phi \mid \mathsf{Fb}\,\phi \mid \phi \Rightarrow_{\mathsf{Ob}} \psi \mid \phi \Rightarrow_{\mathsf{Pm}} \psi \mid \phi \Rightarrow_{\mathsf{Fb}} \psi\end{aligned}$$

where $p \in P$ is a predicate symbol of arity $n \geq 0$ and the t_i, $1 \leq i \leq n$, are terms. Terms are freely generated by the function symbols from F and variables from V. ⌟

[3] See `https://github.com/normativeai`.

$\mathbf{DL}^*_1$ extends Standard Deontic Logic (SDL) with the normative concepts of ideal and contrary-to-duty obligations, and contains predicate symbols, the standard logical connectives, and the normative operators of obligation (Ob), permission (Pm), prohibition (Fb), their conditional counter-parts, and ideality (Id). Free variables are implicitly universally quantified at top-level.

This logic is expressive enough to capture many interesting normative structures. For details on its expressivity and its semantics, we refer to previous work [7].

3 The NAI Suite

The NAI suite integrates novel theorem proving technology into a usable graphical user interface (GUI) for the computer-assisted formalization of legal texts and applying automated normative reasoning procedures on these artifacts. In particular, NAI includes

1. a legislation editor that graphically supports the formalization of legal texts,
2. means of assessing the quality of entered formalizations, e.g., by automatically conducting consistency checks and assessing logical independence,
3. ready-to-use theorem prover technology for evaluating user-specified queries wrt. a given formalization, and
4. the possibility to share and collaborate, and to experiment with different formalizations and underlying logics.

NAI is realized using a web-based Software-as-a-service architecture, cf. Fig. 1. It comprises a GUI that is implemented as a Javascript browser application, and a NodeJS application on the back-end side which connects to theorem provers, data storage services and relevant middleware. Using this architectural layout, no further software is required from the user perspective for using NAI and its reasoning procedures, as all necessary software is made available on the back end and the computationally heavy tasks are executed on the remote servers only. The results of the different reasoning procedures are sent back to the GUI and displayed to the user. The major components of NAI are described in more detail in the following.

3.1 The Reasoning Module

The NAI suite supports formalizing legal texts and applying various logical operations on them. These operations include consistency checks (non-derivability of falsum), logical independence analysis as well as the creation of user queries that can automatically be assessed for (non-)validity. After formalization, the formal representation of the legal text is stored in a general and expressive machine-readable format in NAI. This format aims at generalizing from concrete logical formalisms that are used for evaluating the logical properties of the legal document's formal representation.

There exist many different logical formalisms that have been discussed for capturing normative reasoning and extensions of it. Since the discussion of such formalisms is still ongoing, and the choice of the concrete logic underlying the reasoning process strongly influences the results of all procedures, NAI uses a two-step procedure to employ automated reasoning tools. NAI stores only the general format, as mentioned above, as

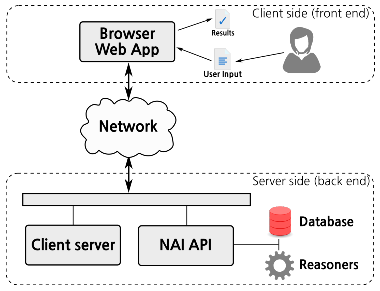


Fig. 1: Software-as-a-service architecture of the NAI reasoning framework. The front end software runs in the user's browser and connects to the remote site, and its different services, via a well-defined API through the network. Data flow is indicated by arrows.

result of the formalization process. Once a user then chooses a certain logic for conducting the logical analysis, NAI will automatically translate the general format into the specific logic resp. the concrete input format of the employed automated reasoning system. Currently, NAI supports only the $\mathbf{DL}^*_1$ logic from §2; however, the architecture of NAI is designed in such a way that further formalisms can easily be supported. Possible extensions are described in §5.

The choice in favor of $\mathbf{DL}^*_1$ is primarily motivated by the fact that it can be effectively automated using a shallow semantical embedding into normal (bi-)modal logic [7]. This enables the use of readily available reasoning systems for such logics; in contrast, there are few to none automated reasoning systems available for normative logics (with the exception of [5]). In NAI, we use the MleanCoP prover [10] for first-order multimodal logics as it is currently one of the most effective systems and it returns proof certificates which can be independently assessed for correctness [9]. It is also possible to use various different tools for automated reasoning in parallel (where applicable). This is of increasing importance once multiple different logical formalisms are supported.

3.2 The Annotation Editor

The annotation editor of NAI is one of its central components. Using the editor, users can create formalizations of legal documents that can subsequently used for formal legal reasoning. The general functionality of the editor is described in the following. A more detailed exemplary application on a concrete legal document is presented in §4.3.

One of the main ideas of the NAI editor is to hide the underlying logical details and technical reasoning input and outputs from the user. We consider this essential, as the primary target audience of the NAI suite are not necessarily logicians and it could greatly decrease the usability of the tool if a solid knowledge about formal logic was required. This is realized by letting the user annotate legal texts and queries graphically and by allowing the user to access the different reasoning functionalities by simply clicking buttons that are integrated into the GUI. Note that the user can still inspect the logical formulae that result from the annotation process and also input these formulae directly. However, this feature is considered advanced and not the primary approach put forward by NAI.

The formalization proceeds as follows: The user selects some text from the legal document and annotates it, either as a term or as a composite (complex) statement. In the first case, a name for that term is computed automatically, but it can also be chosen freely. Different terms are displayed as different colors in the text. In the latter case, the user needs to choose among the different possibilities (which roughly correspond to logical connectives) and the containing text can be annotated recursively. Composite statements are displayed as a box around the text. An example of an annotation result is displayed in Fig. 4

The editor also features direct access to the consistency check and logical independence check procedures (as buttons). When such a button is clicked, the current state of the formalization will be translated and sent to the back-end provers, which determine whether it is consistent resp. logically independent.

User queries are also created using such an editor. In addition to the steps sketched above, users may declare a text passage as *goal* using a dedicated annotation button, whose contents are again annotated as usual. If the query is executed, the back-end provers will try to prove (or refute) that the goal logically follows from the remaining annotations and the underlying legislation.

3.3 The Abstract Programming Interface (API)

All the reasoning features of NAI can also be accessed by third-party applications. The NAI suite exposes a RESTful (Representational state transfer) API which allows (external) applications to run consistency checks, checks for independence as well as queries and use the result for further processing. The exposure of NAI's REST API is particularly interesting for external legal applications that want to make use of the already formalized legal documents hosted by NAI. A simple example of such an application is a tax counseling web site which advises its visitors using legal reasoning over a formalization of the relevant tax law done in the NAI suite.

4 A Methodology for the Creation of Correct Formalizations

The formalization process essentially consists of translating an informal natural language text into a formal logical formula or code. As mentioned before, this step is essential for being able to apply automated reasoning techniques.

We can choose various formulae in the logic $\mathbf{DL}^{*}_{1}$ which seem to describe a text at hand. Each of these formulae differs in the cases in which it holds, and in the consequences which can be derived from it.

A correct formalization means that the right formula is chosen. How can we pick this formula? In [1], Bartolini et al. define a methodology for the validation of the formal representation of legal texts by a backward translation to a human-readable text. The text is then being validated by legal experts. Mockus and Palmirani [8] define a method for the iterative refinement of ontologies, which is inspired by a previous work by Peroni [12]. Peroni's work adapts approaches from the agile methodology in software engineering. The above approaches still depend on humans for validation. In this section we describe a new methodology which is based on Behaviour Driven Development (BDD) [4]. The ”behaviours” defined by this methodology are validated by machines, similarly to those in software engineering.

4.1 Behaviour-driven Development in Software Engineering

Behaviour-driven development (or BDD for short) [5] emerged from the process known as test-driven development (TDD). The concept behind BDD is to provide development and management teams with a shared process and shared tools, so that they can effectively collaborate while developing software. To this end, it combines the basic principles of TDD with object-oriented analysis and domain-driven design, to make the process of creating software as optimized and effective as possible.

In its core, BDD is simply the idea that software development should be governed by both technical proficiencies and business interests alike. However, besides the ideological concept, BDD does make use of specialized software in order to achieve the desired goals. The main tool of the method is a simple domain-specific language (DSL): Instead of complex lines of code, this language uses normal English words and logical constructs to express how the software should behave.

Using BDD in Software Engineering. BDD is a branch of the test-driven development method, which also uses domain-specific language to convert natural language phrases and statements into executable tests. We are talking about sentences that start with a conditional word (should, given, when, if, etc.) and define an outcome. For example:

– If I have two apples
– And my friend takes one
– Then I will have one apple

Basic Principles of BDD. BDD follows the basic principle that each unit of software must be individually tested. The process usually goes like this:

1. A test is designed for the specific software unit
2. The test is made to fail
3. The unit is then implemented into the test
4. The test is done again, verifying that the implementation of the unit makes it succeed

This basic outline is perfect, because it allows the testing of both high and low-level software, as well as anything in between. When using the BDD methodology, the tests should be specified in terms of the desired behaviour of the unit in question. This behaviour is basically the requirements set by the business entity that commissioned the creation of the software.

[4] `https://www.agilealliance.org/glossary/bdd/`
[5] The description is based on the definition in `http://behaviour-driven.org/`

Benefits of BDD. There are various benefits of using BDD in software engineering. In [11] they identify seven themes in which research has shown the advantages of BDD over other methods. Among the themes, three are especially relevant to legal formalization and are discussed below.

Cost. Some research suggests that BDD can help keep projects within budget. Findings are inconclusive about that. The same advantages can exist when formalizing legal texts.

Time. There is much evidence that BDD can reduces the development time. One of BDD's main goals is to keep implementation limited to passing the tests and therefore reduces implementation time. In addition, tests assure that changes to the code still conform to previous requirements. These benefits hold for legal formalization as well. There is much flexibility when formalizing legal text, from very specific and detailed formalization to a more abstract one. The level of detail depends on the intended use of the formalization. For example, if the intended queries we plan on executing over the text do not deal with specific laws regarding the age of the offender, there is no need to formalize those. Similarly, changes to formalizations are required from time to time, either because of changes in the law or the need to use a more detailed level. Tests ensure that those changes are compatible with previous requirements.

People. BDDs help bridge the gap between stack holders and programmers. The tests are normally written by stack holders and are automatically converted to code. This point is even more relevant to the legal domain. One of the main difficulties in legal formalization is the need to have both legal and technical/logical skills. The most popular approach to legal formalization depends on Prolog programming skills, for example. Using BDD in legal formalization will allow legal experts to write the tests while programmers and logicians will focus on the implementation of a formalization which satisfies them.

4.2 Towards Behaviour-driven Development of Legal Formalization

BDD enjoys very high success in software engineering and we believe that it can be adapted to legal text formalization as follows.

The lawyer writes down different scenarios which should be true (or false), given her interpretation of the legal text. The lawyer then annotates these scenarios in order to translate them into test formulae. In the last step, a person needs to annotate the legal text in a way such that all the test formulae will be validated. It should be noted that the person in the last step must not have a full legal understanding of the text and that in principle, this last step can even be executed by a machine, which tries different formalization possibilities until all test formulae are satisfied.

More formally and in alignment with the BDD process, we define the behaviour-driven development of a legal formalization as follows:

1. The legal expert writes down a set of legal cases and their intended resolution.
2. The cases naturally fail since they are not yet handled by the formalization. The failure is determined by the execution of the tests in a theorem prover.

3. Programmers or logicians then attempt to formalize the specific part of the legislation which covers those cases. If the number of possibilities is finite, this step can be automated in the future. Even if full automation is not possible, approaches like machine learning can make it more feasible.
4. The cases are executed again to verify that the formalization correctly captures the elements of the legislation which corresponds to the cases.

We need therefore to start with a comprehensive list of scenarios and their outcomes based on our legal interpretation. It should be noted that such scenarios are normally based on many articles or even on the whole text. In our example, we will derive them from one article only.

4.3 Case Study: Scottish Smoking Regulation

In this section we are going to demonstrate how the NAI suite can be used for implementing the above methodolgy on a legal text. The text we will use is the ”Smoking Prohibition (Children in Motor Vehicles) (Scotland) Act 2016” [6]. We have chosen this text as it makes a perfect candidate for legal reasoning, being short and relatively self contained. It has also featured in previous research [15].

This legislation contains 19 articles which go from describing the conditions of committing the offence to how a fine can be given and contested. In this example, we will focus on article 1 only. A more comprehensive formalization which includes sentences of the second part as well, is available online [7].

Article 1: Offence of smoking in a motor vehicle with children

1. It is an offence for an adult to smoke in a private motor vehicle when: (a) there is a child in the vehicle, and (b) the vehicle is in a public place.
2. Subsection (1) does not apply to a private motor vehicle that is designed or adapted for use as living accommodation and which, at the time the smoking occurs, is parked and is being used as living accommodation.
3. A person who commits an offence under subsection (1) is liable on summary conviction to a fine not exceeding level 3 on the standard scale.

In order to be able to apply automated reasoning to this text, we first need to formalize our understanding of its meaning. In other words, we need to formalize a legal interpretation of the text.

There are various interpretations possible even for this, relatively simple, text. For the purpose of this example, we interpret the article as prohibiting adults to smoke in a private motor vehicle in case: (1) there is a child in the vehicle, (2) the vehicle is in public space and (3) the vehicle is not adapted or designed to be used, and at the same time is being used, as living accommodation.

Violating this prohibition, the adult is liable to a fine via a summary conviction.

Here we describe just a few of these scenarios. The reader is referred to the live example in the application for more cases.

[6] `https://www.legislation.gov.uk/asp/2016/3/contents`

[7] Please visit `nai.uni.lu` and log in with the credentials: smoking@nai.lu / nai

The first step in the methodology is to create the vocabulary used in the formalization. As mentioned in Section 3.2, this is being done by using the *term* annotation on the text. The annotated terms can then be seen on the ”Vocabulary” tab of the NAI suite. Figure 3a summarizes those for Article 1.

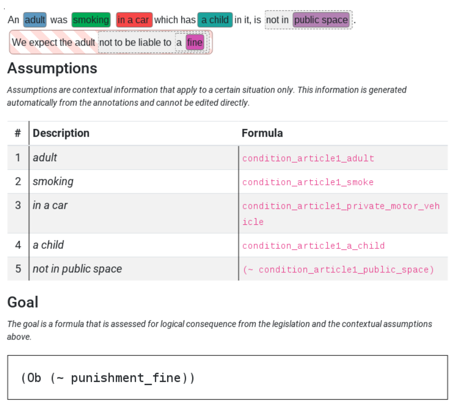
The test queries can now be created based on this vocabulary. The task of the lawyer is to consider different terms from the vocabulary and decide what is the expected outcome of them.

Scenario 1. An adult was smoking in a car which has a child in it and is not in public space. We expect the adult not to be liable to a fine.

Scenario 2. An adult was smoking in a car which has a child in it, is in public space and was not designed as living accommodation. We expect the adult to be liable to a fine.

The lawyer now uses the queries tab in the NAI suite in order to enter these two scenarios. In order to differentiate the test queries from case queries (queries written in order to solve a specific case), the test queries names are prefixed with ”Test ”.

We can now annotate the two scenarios. We proceed first by annotating the conditions with the terms from the vocabulary. The user needs to select those from a drop down list. The expectation is then annotated as a goal. Within the goal, we annotate our expectation that the person is liable to a fine by using the *Permission* connective over the *punishment_fine* term. The two annotated scenarios, as well as their formalization, can be seen in figures 2a and 2b. When executing these queries, they naturally may fail. When annotating the legal text in the next phase, we must make sure that all the queries are now being validated.



An adult was smoking in a car which has a child in it, is not in public space.
We expect the adult not to be liable to a fine

Assumptions

Assumptions are contextual information that apply to a certain situation only. This information is generated automatically from the annotations and cannot be edited directly.

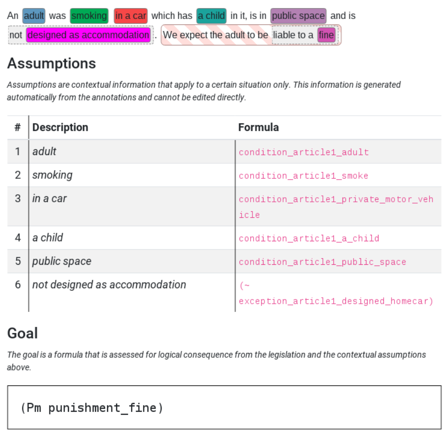
#	Description	Formula
1	*adult*	`condition_article1_adult`
2	*smoking*	`condition_article1_smoke`
3	*in a car*	`condition_article1_private_motor_vehicle`
4	*a child*	`condition_article1_a_child`
5	*not in public space*	`(~ condition_article1_public_space)`

Goal

The goal is a formula that is assessed for logical consequence from the legislation and the contextual assumptions above.

```
(Ob (~ punishment_fine))
```

(a) Scenario 1.



An adult was smoking in a car which has a child in it, is in public space and is not designed as accommodation. We expect the adult to be liable to a fine

Assumptions

Assumptions are contextual information that apply to a certain situation only. This information is generated automatically from the annotations and cannot be edited directly.

#	Description	Formula
1	*adult*	`condition_article1_adult`
2	*smoking*	`condition_article1_smoke`
3	*in a car*	`condition_article1_private_motor_vehicle`
4	*a child*	`condition_article1_a_child`
5	*public space*	`condition_article1_public_space`
6	*not designed as accommodation*	`(~ exception_article1_designed_homecar)`

Goal

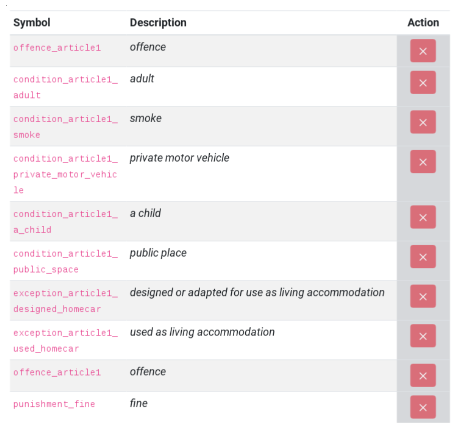
The goal is a formula that is assessed for logical consequence from the legislation and the contextual assumptions above.

```
(Pm punishment_fine)
```

(b) Scenario 2.

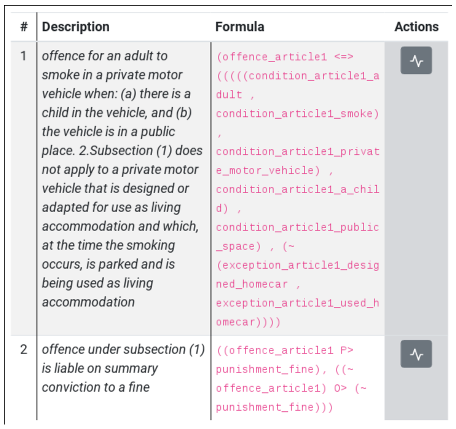
Fig. 2: Annotations and $\mathbf{DL}*_1$ formulae.

We can now proceed with the last step - the annotation of Article 1. After some trial and error, we have ended up with the annotation in Figure 4. This annotation passes all of our test queries and we therefore conclude that it is a faithful formalization of our interpretation of Article 1. The **DL**$*_1$ formulae are shown in Figure 3b.



Symbol	Description	Action
offence_article1	*offence*	×
condition_article1_adult	*adult*	×
condition_article1_smoke	*smoke*	×
condition_article1_private_motor_vehicle	*private motor vehicle*	×
condition_article1_a_child	*a child*	×
condition_article1_public_space	*public place*	×
exception_article1_designed_homecar	*designed or adapted for use as living accommodation*	×
exception_article1_used_homecar	*used as living accommodation*	×
offence_article1	*offence*	×
punishment_fine	*fine*	×

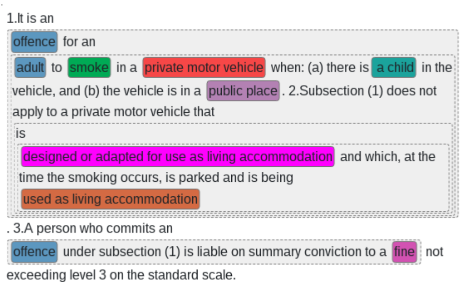
(a) Vocabulary



#	Description	Formula	Actions
1	*offence for an adult to smoke in a private motor vehicle when: (a) there is a child in the vehicle, and (b) the vehicle is in a public place. 2.Subsection (1) does not apply to a private motor vehicle that is designed or adapted for use as living accommodation and which, at the time the smoking occurs, is parked and is being used as living accommodation*	`(offence_article1 <=> (((((condition_article1_adult , condition_article1_smoke) , condition_article1_private_motor_vehicle) , condition_article1_a_child) , condition_article1_public_space) , (~ (exception_article1_designed_homecar , exception_article1_used_homecar))))`	
2	*offence under subsection (1) is liable on summary conviction to a fine*	`((offence_article1 P> punishment_fine), ((~ offence_article1) O> (~ punishment_fine)))`	

(b) **DL**$*_1$ formulae

Fig. 3: Smoking legislation article 1



1.It is an offence for an adult to smoke in a private motor vehicle when: (a) there is a child in the vehicle, and (b) the vehicle is in a public place. 2.Subsection (1) does not apply to a private motor vehicle that is designed or adapted for use as living accommodation and which, at the time the smoking occurs, is parked and is being used as living accommodation. 3.A person who commits an offence under subsection (1) is liable on summary conviction to a fine not exceeding level 3 on the standard scale.

Fig. 4: Smoking legislation article 1: annotation

It should be mentioned that on each step, we are advised to check the consistency of our annotations as well as those of the queries. The reasoning engine can find automatically inconsistencies in our annotations, which can lead to wrong results. In addition, it is recommended to check, on the "Formalization" tab, that each **DL**$*_1$ formula is independent. Dependent formulae are normally a sign of an incorrect formalization.

Case queries Once we are confident that our formalization is faithful to our interpretation, we can trust it to resolve legal questions with regard to specific cases. Writing case queries is identical to the writing of test queries. As an example, consider the following case.

Case 1. A client got a fine while driving his home car while smoking. His teen daughter was sitting next to him. Is there a case to appeal this decision?

Here we want to check if there was an obligation in the law not to give our client the fine. In case it is true, an appeal should be successful. When we annotate the case above, we get that a conclusion cannot be drawn (the query is counter-satisfiable). The reason for that is because some of the conditions are not used. Since there might be two different values to these conditions which result in two different conclusions, the reasoner cannot determine if the query holds. In this case, we can find in the ”Vocabulary” tab one further condition - the car should be in public space - and one further exception - the car should also be used as a home car, and not only be designed as one. We therefore ask the client to share more information about the case.

Case 2. The client adds further that he was indeed driving in public space. The home car though, was not used as a home car at the time. The client has removed the home facilities and is using the car for transportation of goods.

The addition of the new annotations gives us the answer that the policeman was indeed permitted to give the fine. The client could enjoy the exception of subsection (b), but he failed to use the car for accommodation. It seems better not to appeal the fine.

5 Conclusion and Future Work

In this paper we have described a new methodology for validating legal knowledge bases that is inspired by the behaviour-driven development approach from the field of software engineering. As a first step towards implementing this methodology, the NAI suite for normative reasoning is introduced and its application is demonstrated on an exemplary regulation.

The presented case study suggests that the NAI tool can be used by people without a strong IT background, as only few technical details are exposed to the user and most of the task is supported by a graphical user interface. In fact, one could argue that our approach also enables a broad range of users to contribute to the built-up of a reliable legal knowledge base; once the intended behaviour of the formalized norms are agreed upon (by legal experts), it is easy to automatically check compliance of the generated knowledge with the afore stated goal.

The tools presented in this paper are prototypes. Further work is required on both the tools and their supporting theories in order to make the formalization of legal texts easier and more intuitive. Among those improvements, the most notable ones relate to the supporting theory and to the usability of the user interface. We mention several such improvements here.

Currently, the NAI suite supports an expressive deontic first-order language. This language is rich enough to describe many scenarios which appear in legal texts. Nevertheless, more work is required in order to capture all such scenarios. Among those features with the highest priority, we list support for exceptions, temporal sentences and arithmetic. In this paper, we overcame the fact that subsection 1(b) is an exception to subsection 1(a) by explicitly mentioning the values of the conditions of the exception. This solution is not optimal since it requires the setting of values to these properties in all tests and cases. Possible support for these features already exists in the form of tools such as non-monotonic reasoners [6], temporal provers [14] and SMT solvers [4].

On the level of usability, the tool currently does not give any information as to why a query is counter-satisfiable. The user needs to look on the vocabulary in order to determine possible reasons. Integrating a model finder, such as Nitpick [2], will help "debugging" formalizations.

NAI's graphical user interface (GUI) aims at being intuitive and easy to use and tries to hide the underline complexities of the logics involved. A continuously updated list of new features can be found on the GUI's development website [8] .

References

1. Bartolini, C., Lenzini, G., Santos, C.: An interdisciplinary methodology to validate formal representations of legal text applied to the gdpr. In: JURISIN (2018)
2. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: ITP. pp. 131–146. Springer (2010)
3. Boella, G., Di Caro, L., Humphreys, L., Robaldo, L., Rossi, P., van der Torre, L.: Eunomos, a legal document and knowledge management system for the web to provide relevant, reliable and up-to-date information on the law. Artificial Intelligence and Law **24**(3), 245–283 (2016)
4. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: verit: an open, trustable and efficient smt-solver. In: CADE. pp. 151–156. Springer (2009)
5. Governatori, G., Shek, S.: Regorous: a business process compliance checker. In: Proc. of the 14th Int. Conf. on Artificial Intelligence and Law. pp. 245–246. ACM (2013)
6. Kifer, M.: Nonmonotonic reasoning in flora-2. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 1–12. Springer (2005)
7. Libal, T., Pascucci, M.: Automated reasoning in normative detachment structures with ideal conditions. In: Proc. of ICAIL. pp. 63–72 (2019). https://doi.org/10.1145/3322640.3326707
8. Mockus, M., Palmirani, M.: Legal ontology for open government data mashups. In: 2017 Conference for E-Democracy and Open Government (CeDEM). pp. 113–124. IEEE (2017)
9. Otten, J.: Implementing connection calculi for first-order modal logics. In: IWIL@ LPAR. pp. 18–32 (2012)
10. Otten, J.: Mleancop: A connection prover for first-order modal logic. In: 7th International Joint Conference, IJCAR. pp. 269–276 (2014). https://doi.org/10.1007/978-3-319-08587-6_20
11. Park, S., Maurer, F.: A literature review on story test driven development. In: International Conference on Agile Software Development. pp. 208–213. Springer (2010)
12. Peroni, S.: A simplified agile methodology for ontology development. In: OWL: Experiences and Directions–Reasoner Evaluation, pp. 55–69. Springer (2016)
13. Robaldo, L., Bartolini, C., Palmirani, M., Rossi, A., Martoni, M., Lenzini, G.: Formalizing gdpr provisions in reified i/o logic: The dapreco knowledge base. Journal of Logic, Language and Information pp. 1–49 (2019)
14. Suda, M., Weidenbach, C.: A pltl-prover based on labelled superposition with partial model guidance. In: IJCAR. pp. 537–543. Springer (2012)
15. Wyner, A.Z., Gough, F., Lévy, F., Lynch, M., Nazarenko, A.: On annotation of the textual contents of scottish legal instruments. In: JURIX. pp. 101–106 (2017)

[8] `https://github.com/normativeai/frontend/issues`