A Software Tool for Legal Drafting*

Daniel Gorín
Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Argentina
dgorin@dc.uba.ar

Sergio Mera
Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Argentina
smera@dc.uba.ar

Fernando Schapachnik
Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Argentina
fschapachnik@dc.uba.ar

Although many attempts at automated aids for legal drafting have been made, they were based on the construction of a new tool, completely from scratch. This is at least curious, considering that a strong parallelism can be established between a normative document and a software specification: both describe what an entity should or should not do, can or cannot do. In this article we compare normative documents and software specifications to find out their similarities and differences. The comparison shows that there are distinctive particularities, but they are restricted to a very specific subclass of normative propositions. The rest, we postulate, can be dealt with software tools. For such an enterprise the FormaLex tool set was devised: an LTL-based language and companion tools that utilize model checking to find out normative incoherences in regulations, contracts and other legal documents. A feature-rich case study is analyzed with the presented tools.

1 Introduction

Although many attempts at automated aids for legal drafting have been made (e.g., [17, 24, 15, 25, 18, 10]), they were based on the construction of a new tool, completely from scratch. This is at least curious, considering that a strong parallelism can be established between a normative document and a software specification: both describe what an entity should or should not do, can or cannot do. In the case of normative documents, it is a legal entity. In the case of software specifications, it is a piece of software. Is a software specification so different from a normative document? If it is not, why do not reuse the already existing machinery that can successfully analyze specifications?

In this article we compare normative documents and software specifications to find out their similarities and differences. The comparison shows that there are distinctive particularities, but they are restricted to a very specific subclass of normative propositions. The rest, we postulate, can be dealt with temporal model checkers. For such an enterprise the FormaLex tool set was devised: an LTL-based language, `FL`, and companion tools that utilize model checking to find out normative incoherences in legal documents.

`FL` is based on the following key-concepts:

- It provides a *background theory* to state matters about the real world, such as event precedence (e.g., sunrise before dawn), uniqueness (each person is born only once), etc., that would otherwise had to be accommodated into unnatural deontic rules. Said background

*Partially supported by PICT 2007 532, UBACyT 20020090200116, 2002009020084 and 20020100200103.

E. Pimentel, V. Valero (Eds.): Workshop on Formal Languages and Analysis of Contract-Oriented Software 2011 (FLACOS'11)
EPTCS 68, 2011, pp. 71–86, doi:10.4204/EPTCS.68.7

theory is translated into an automaton that determines the class of models over which the rest of the language predicates. Section 4.1 provides the details.

- Deontic rules are translated into LTL, but the input language, that is, the way the original deontic rules are written, is preserved, so this information can be used to perform a set of analysis, at a meta-logical level. See Section 4.3 for details.
- The combination of an automata-based formalism plus a logic that can refer to it is very powerful, and the software community knows it. `FL` takes advantage of that to easily express properties that are generally difficult to pose in other formalisms, or lead to computational complexity problems. These features can be seen in Sections 3 and 5.

A comparison between specifications and legal documents is presented in Section 2. Section 3 highlights `FL`, our LTL-based language, by describing its use to express otherwise hard-to-write properties, and Section 4 gives details of the tool's inner workings and formal semantics. A case study, where `FL` is used to expose problems in a feature-rich university's regulation, is discussed in Section 5.[1] Section 6 compares related work and Section 7 concludes the article.

2 Specifiable Regulations?

It is often understood that regulations can be abstractly represented using the three well-known deontic operators for obligation (O), prohibition (F) and permission (P). Specifications can be also thought of as using the same operators. "*The system must activate the brakes in no more than three seconds after the emergency stop button is pressed*" is clearly an example of obligation. Prohibitions are also found: "*it is forbidden that the server sends unencrypted passwords through the wire*". Permissions are not that frequent, but also possible: "*if out of resources the system can drop requests until the processor is freed*".

Nevertheless, some types of statements found in legal documents are not common in software specifications. We divide them in two groups. The first one is composed of statements that have equivalents in specifications, but under a somehow different structure:

Contrary-To-Duty Obligations. Contrary-To-Duty (CTD) obligations are a way to model phrases like "*The agent is required to do action X. If she does not, then action Y should be performed*". The first sentence is called the *obligation* and the last one the *reparation*, and we denote that as $O_Y(X)$. Software specifications equivalents are conceivable, but the key difference is how to treat $O(X) \wedge O_Y(X)$. A Software Engineering perspective could read the formula as "*obligation to perform X and obligation to perform X, plus reparation Y in case of failure of X, equals obligation to perform X plus reparation Y*", considering that $O_Y(X)$ somehow supersedes plain $O(X)$. However, a legal point of view is that $O_Y(X)$ means that a behavior that does not satisfy X but performs Y is still legal, while the same behavior is illegal for the formula $O(X)$, that does not contemplate a reparation.

Amendments to Deal with Contradictions. A legal corpus might contain the formula F(kill), and then be modified to allow self-defense by the addition of P(kill in self-defense). It is hard to consider that such a corpus is contradictory, yet a software engineer will rather use the specification F(kill unless in self-defense) that does not entitle a logical contradiction.

[1] Another `FL` case study was presented in [13], yet it was much smaller both in size and concepts.

Permissions. Besides their use as exceptions to prohibitions, what are permissions exactly? The question has been raised and analyzed many times before (e.g., [2, 5, 21, 19]), so let us here only say that in software a permission is little more than no-determinism, while in a normative system a permission is a much complex individual. It is worth noticing that the ability of a user of some application to use or not some functionality is not a permission, it is an obligation: the specification would probably say that the software is obliged to behave in a way or some other depending on the user's choices. Some phrases lend themselves to confusion: "*The user may print the displayed listing*", in the context of a software specification is just a simpler wording for stating that in order to comply with the specification the software is obliged to present the printing option, and, if chosen, print the listing.

Hierarchies. Legal corpora have hierarchies. For example a regional law might set a tax to $10 while the national law sets the same tax to $20. If national laws override regional ones in said normative system, the outcome is clearly $20. A software engineer might be tempted to say that such system is equivalent to one that sets the tax to $20. However, the real normative system permits that if the national tax-setting law is canceled, the tax is still set at $10 by the regional one, while the software engineer model's does not.

Ontologies. In legal corpora ontologies are of common use. For instance, a law might set standards for animals such as pets, while there might be another, more specific for dogs, that might set different conditions. Although software engineers are familiar with inheritance and subclassing, it is not common to specify the requirements for a general class of actors and separately others, possibly contradicting the former, for a subclass. In a software specification they will be treated in an ad-hoc manner, for example, with a requirement for dogs and another for pets that are not dogs, thus avoiding the contradiction.

There are other types of statements that we believe have no equivalent in software specifications:

Nesting of Deontic Operators. Nesting of deontic operators, as in obligation to obligate (or to forbid, or to permit). In normative propositions such as "*The judge is obliged to oblige the citizen to do X*" there are two obligations (and two responsible parties): if the judge does not comply to oblige the citizen, she is to blame. If the judge complies but the citizen does not, then the citizen is at fault. The specification of a security system might use a similar phrase: "*The system is obliged to oblige other users to do Y*", with Y being something like "*not access each others private files*". In this case, if the system does not enforce Y, then the system is at fault. Also, if users fail to do Y, then the same system is also responsible. This type of predicates seem to be just a complex wording for only one obligation.

Care should be exercised when analyzing propositions where there is an apparent nesting of obligations, but can be rewritten without nesting. E.g., "*The voting system should oblige users to deposit the ballot in the case in less than one minute or else face prosecution*". Such a phrase has a very different meaning if found in the legal norm that regulates electronic vote, or in the software's specification. In the first case, it binds both developers and voters, while in the second only developers, as a software specification has no power over voters. In such a case, it can be

rewritten as "*The voting system has to give one minute for the ballot to be deposited into the case. In case of timeout, prosecution actions should be initiated [i.e., by notifying officials].*".

Self-Referencing Modifications. Self-referencing modifications, as in "*Let Article X of Bill Y be modified to mandate that from now on such and such*". *Self* here means that they modify the same normative systems that contains them. This should not be confused with any type of software compile-time or runtime configuration. In such cases the specification is still fixed and contemplates the different possible behaviors.[2]

Deontic-Conditional Validity. Deontic operators whose activating condition is the validity of another deontic operator. A typical example is a conditional over an obligation as in "*if at the time of the execution the agent were obligated to ... then she ...*". Software specifications might impose obligations based on the runtime operating conditions of the software, but they do not specify behaviour that is conditional to the *runtime requirements*, if that term makes any sense at all.

We found that the common denominator of the last group is considering the deontic operators as first-class operators, allowing for operators that take operators as parameters, check if they are active, and so on. However, we believe that if we consider only the legal documents that do not use such classes of predicates we can a) cover an important and varied amount of regulations that are common in the real world, and b) resort to the tools and technologies that can be used to analyze software specifications. The first group of predicates, we postulate, can be accommodated in such setting if treated properly.

We believe that this is good news for the deontic community, as it means that decades of effort in software-analysis tool building, optimizations and expressivity improvements can be leveraged, and there is no need to start from scratch and climb again the road from handling toy examples to real-size ones.

3 The FL Language

Our starting point is that many contracts and regulations can be formally analyzed with tools originally aimed for software specifications. This allows for making the most out of existing tools.

FL, introduced in [13], is built on the following premises:

- It aims at finding *coherence problems*, defined in a very pragmatic way: behaviours can not be permitted and forbidden, or obligated and forbidden, can not be plain mandatory or mandatory with CTDs, CTD reparations cannot be forbidden, etc. The complete list of covered topics is presented in Section 4.3.
- The input language is divided into a *background theory* and a set of rules. While the rules are LTL formulae with additional deontic operators aimed at capturing normative propositions, the background theory provides some simple constructs to describe the class of models over which the rules predicate.

[2] Although there are some prototype dynamic specification languages with self-referencing capabilities, they are still far from being used in the current state of the practice.

- Models are linear[3], and each one describes a possible *legal* behaviour. That is, behaviours that do not comply with the normative rules are discarded.
- If something is obligatory, then it must hold in every legal model at every possible state, and thus $O(\varphi)$ is interpreted as $\Box\varphi$. Prohibition of something is obligation to the contrary ($F(\varphi) \equiv O(\neg\varphi)$). The diamond operator works in the usual way, and $\Diamond\varphi$ looks ahead in the model for some state where φ holds.
- Contrary-To-Duty obligations are supported as $O_\rho(\varphi)$, translated as $\Box(\neg\varphi \to \rho)$. $F_\rho(\varphi)$ is interpreted as $O_\rho(\neg\varphi)$. It is worth noticing that our encoding skips out most of the deontic logic paradoxes (see [14, Sect. 6] for details).
- Although based on translating to LTL, the input syntax is preserved to perform analysis at the meta-logical level.

Permission is thought of as absence of prohibition, but treated not as an operator that modifies the set of legal behaviours, but rather as a predicate that the legal models must satisfy. Otherwise, it is considered that the normative system under analysis (NSUA) has a *coherence problem*: it states that something is permitted when it actually is not. If the user flags the permission as an exception to a prohibition, as in F(kill) and P(self-defense killing), the internal representation of the affected prohibition is changed to reflect that. In the example, to F(kill unless self-defense).[4]

The main component of the background theory is the *action*. An action can be happening or not at any moment. In `FL` an action is interpreted as a *digital signal* that can be on or off for an arbitrary number of consecutive states. Actions can represent proper actions by the implicit agents (e.g., **action** `DriveCar`) or non-controllable, external events (e.g., **action** `CarCrash`). There is no explicit notion of a role performing an action, so if they are needed the subject must be encoded in the action. We plan to add this feature in the future.

Some requirements seem to be easy to express, like being licensed in order to drive a car. It would seem that it suffices to forbid the `DriveCar` action if there is no prior `GetLicense` action. But the easiness is only apparent, as individuals can not only get their license, but also lose it. If the `LoseLicense` action is also considered, establishing whether an individual is licensed or not by means of a pure formula amounts to "parenthesis counting", and that can be very challenging to write or plain impossible depending on the particular logic used. `FL` incorporates the notion of *intervals*, similar to the *fluents* of [12]. An interval is delimited by beginning and ending actions, in a such a manner that there is no nesting nor closing of an already closed interval. In the automata, a propositional variable is set to true or false, indicating whether the interval is open (see Section 4.1 for details). With intervals, the driving requirement can be posed as

interval `licensed` **delimited by actions** `GetLicense-LoseLicense`

and then simply stating $F(\neg \mathit{is\_licensed} \wedge \texttt{DriveCar})$.

Intervals can also be used to bound the occurrence of other actions:

interval `school_time` **delimited by actions** `CourseBegin-CourseEnds`
action `TakeExam` **occurs only in scope** `school_time`

There is also the view of obligation not as something that must always hold, but rather as something that must be done, usually within some bound of time, sometimes called *non-persistent obligation*. We can deal with such expressions in two ways. Either with the $O^E(\varphi)$

[3]It does not mean the branching alternatives are not present, each possible alternative is present in one of the considered models.

[4]This can be done automatically for simple cases and requires manual intervention in others.

operator that expresses an *eventual* obligation, one that ceases to exists as soon as it is fulfilled, or, if time bounds are provided (e.g., "*You ought to return the borrowed books within school time*"), by using intervals inside obligations: $O(\Diamond_{\texttt{school\_time}}\texttt{ReturnBook})$. Other interactions between obligations and deadlines are shown in [14].

As the background theory is translated into an automata, we can accommodate there (bounded) *counting*, allowing for expressions like $O(\texttt{bbc} > 0 \rightarrow \Diamond\texttt{bbc} = 0)$, where the integer counter `bbc`, *borrowed books count*, is handled by the automata incrementing it and decrementing it with every `BorrowBook` or `ReturnBook`, respectively. Such formula properly states that every borrowed book must be returned, whereas the simple $O(\texttt{BorrowBook} \rightarrow \Diamond\texttt{ReturnBook})$ is satisfied by borrowing many and returning just one.

Also interesting are persistent obligations with one-time-each reparations, where violating the obligation once, or even performing the reparation should not free the subject from the obligation. An example of that is obligation to not cross red traffic lights, subject to a fee for each violation. Such CTD obligations are usually hard to express. For instance, $F_{\texttt{PayFine}}(\texttt{RedCrossing})$, says that red crossing is forbidden, except that a fine is payed immediately. If a diamond is added, as in $F_{\Diamond\texttt{PayFine}}(\texttt{RedCrossing})$, then many violations can be canceled with one payment.

In `FL` the formula can be easily stated with the aid of a `fines` counter that increments with `BeFined` and decrements with `PayFine`, and the following formulae: $F_{\texttt{BeFined}}(\texttt{RedCrossing})$ (red crossing is forbidden, under the penalty of being fined), $O(\texttt{fines} > 0 \rightarrow \Diamond\texttt{PayFine})$ (it is mandatory to eventually pay the fines).

Section 4 shows formal semantics, how the background theory is translated into automata, the handling of formulae and how the coherence checks are defined and performed.

4 Semantics & Inner Workings

4.1 Background Theory

`FL`'s background theory is translated into a Büchi automata network with additional code that controls state transitions and handles state variables. Each run of the automata defines a standard LTL model over which the rules formalized in the next section predicate. The tool can use SPIN [20], DiVinE [3] or NuSMV [9] as backends for model checking, but the encoding presented here will use an agnostic dialect.

Time modeling is discrete, considered as succession of states, some of which have a proper name to refer to timestamps of interest for the NSUA. In `FL` an action is thought of as a *digital signal* that can be on or off for an arbitrary number of consecutive states. Thus, each action is represented by an enumerated variable that covers the `HAPPENING` and `NOT_HAPPENING` states. This is an easy way to model time density, as the net effect is that any event can happen while others are taking place.

One single automaton is responsible for controlling all the variables. It has a single state called `running` and non-deterministically guarded self-transitions. Let's exemplify with the action `BorrowBook` (e.g., from the library).

```
declare enum borrow_book = NOT_HAPPENING

running -> running
        guard borrow_book = NOT_HAPPENING -> set borrow_book = HAPPENING;
        guard borrow_book = HAPPENING -> set borrow_book = NOT_HAPPENING;
```

The automaton is therefore defined as follows:

The automaton can switch the value of `borrow_book` or leave it as it is, changing other variables. At the automaton level only one variable changes at a time, resembling the one input assumption of SCR [4]. As said before, at the normative level many actions can be taking place at the same time.

The encoding shown so far allows to easily refer to whenever actions are happening or not, but sometimes it is required to express that an action has happened completely (i.e., it has finished taking place) or just happened (i.e., has finished taking place in the current state). For instance "*account the loan after borrowing*", needs to refer to a moment when the `BorrowBook` action is not happening after having happened. To facilitate this, another state called `just_happened`, of a type sometimes refered to as *urgent* or *committed*, is included in the automaton. The semantics of such type is that whenever an execution reaches one of these states, of all the available options, the automaton must execute a transition that leaves a committed state:

```
running -> running
        guard borrow_book = NOT_HAPPENING -> set borrow_book = HAPPENING;
running -> just_happened
        guard borrow_book = HAPPENING -> set borrow_book = JUST_HAPPENED;
just_happened -> running
        guard borrow_book = JUST_HAPPENED -> set borrow_book = HAPPENING;
```

With this new possible value for the state variable actions can be not happening for an arbitrary number of states, switch to `HAPPENING`, also for an arbitrary number of states, but before switching to `NOT_HAPPENING` again they must spend one state as `JUST_HAPPENED`. Finished actions are easy to pose with this new state: in $O(\texttt{BorrowBook} \rightarrow \Diamond\texttt{AccountLoan})$. From the formula perspective, the terms `BorrowBook` and `AccountLoan` are translated to propositional variables whose value is `borrow_book`=`JUST_HAPPENED` and `account_loan` = `JUST_HAPPENED` respectively.

If actions have output values, as in:

action `BorrowBook` **output values** { available, in_house_reading_only, not_available }

another enumerated variable `borrow_book_output` is added to the automaton and the encoding turns into:

```
...
running -> just_happened
        guard borrow_book = HAPPENING -> set borrow_book = JUST_HAPPENED,
                                         borrow_book_output = AVAILABLE;
        guard borrow_book = HAPPENING -> set borrow_book = JUST_HAPPENED,
                                         borrow_book_output = IN_HOUSE_READING_ONLY;
        guard borrow_book = HAPPENING -> set borrow_book = JUST_HAPPENED,
                                         borrow_book_output = NOT_AVAILABLE;
...
```

So the output is set non-deterministically to any of the possible values and it is retained until the next setting. Similarly, if the action has extra guards, they are added to the `guard` of the transition.

As we mentioned before, intervals are another important feature of the language. Let's suppose books can only be borrowed during the academic year:

interval `academic_year` **delimited by actions** `BeginYear-EndYear`
action `BorrowBook` **occurs only inside** `academic_year`

A boolean variable, `academic_year_opened` is added to the automaton, and is set by the transitions that represent the respective actions. Also, it is added as a guard, so no `BeginYear` happens inside an academic year and no `EndYear` happens outside one. Similarly, it is added as a guard for `BorrowBook`.

Expressivity-wise, counters are a very powerful feature: they are basically a non-negative integer variable with actions that either increment, decrement or reset their value. The implementation is straightforward: an integer variable is added to the automaton, and it is manipulated in the respective transitions.

Temporal actions are a way to implement timers. The **temporal actions** $ta_1, \ldots, ta_n$ clause declares a sequence of time points that follow the specified order and let an arbitrary number of actions happen between them. The implementation uses another synchronizing automaton with one state representing each ta_i and others representing the time intervals after ta_i and before ta_{i+1}.

4.2 FL's Syntax and Semantics

We present here a formal definition for the rule-stating part of `FL`.[5] Although allowed in the input language, general terms like `bcc` > 0, `action.value`, etc. are abstracted away as propositional symbols in the presented syntax. There is no need to model them explicitly since they can be thought of as encoded in propositional values that later the model handles in the proper way. The same happens with actions `a`, which are abstracted as the proposition `a=JUST_HAPPENED`.

Syntax. Let PROPS be a countable infinite set of symbols, INTERVALS $\subseteq$ (PROPS $\times$ PROPS) a set of intervals, and FORMS the set of `FL` formulae in the signature $\langle$PROPS, INTERVALS$\rangle$ defined as

$$\begin{aligned} \text{INNER\_FORMS} &::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \Diamond\varphi \mid \Diamond_{\mathtt{i}}\varphi \\ \text{FORMS} &::= O(\varphi) \mid F(\varphi) \mid O_\rho(\varphi) \mid F_\rho(\varphi) \mid O^E(\varphi) \mid P(\varphi), \end{aligned}$$

where $p \in$ PROPS, $\varphi, \rho, \varphi_1, \varphi_2 \in$ INNER_FORMS and $\mathtt{i} \in$ INTERVALS. We usually work with (finite) sets of FORMS when specifying a NSUA, so conjunction between formulae in FORMS does not need to be formally defined. We will usually write one formula below another, implicitly defining a conjunction between them. Some operators could have been defined in terms of others, but we intentionally define all of them at this level since our tool considers them differently for coherence analysis (see Section 4.3 for more details).

[5] This reduced presentation does not allow for nesting of deontic operators, a restriction only introduced to save space in this article. For the same reason we ommit the repared version of the O^E operator.

Semantics. We give FL semantics by providing a translation Tr from FL into classic LTL[6], as both work over the same class of models. Let $\mathscr{F}$ be a set of FL formulae whose intended meaning is defining the set of legal models for the NSUA. Recall that LTL models are linear infinite structures that represent possible runs on the automata defined by the background theory. We first split permissions from the rest and define the Tr domain as $\mathscr{F}_{\bar{P}} = \{\varphi \mid \varphi \in \mathscr{F}$ such that φ is not of the form $P(\psi)\}$. Tr acts as the identity for the INNER_FORMS constructions not explicitly specified and it is asumed that the target LTL signature has the implicitly defined propositions involved in the translation (like i_opened for each interval i).

$$\begin{array}{rclcrcl} Tr(\Diamond_{\mathtt{i}}\varphi) & = & \mathtt{i}\_opened \rightarrow (\mathtt{i}\_opened\ U\ Tr(\varphi)) & & Tr(O(\varphi)) & = & \Box Tr(\varphi) \\ Tr(F(\varphi)) & = & \Box\neg Tr(\varphi) & & Tr(O_\rho(\varphi)) & = & \Box(\neg Tr(\varphi) \rightarrow Tr(\rho)) \\ Tr(F_\rho(\varphi)) & = & \Box(Tr(\varphi) \rightarrow Tr(\rho)) & & Tr(O^E(\varphi)) & = & \Diamond Tr(\varphi) \end{array}$$

Tr can be lifted to take a set of FL formulae and return the set of their translations.

Let's now consider an automaton A defined by the background theory and the class of models $\mathscr{C}_A$ that represents all possible runs on A. Let $\mathscr{F}$ be the set of FL formulae that encode the NSUA. The class of legal models defined by $\mathscr{F}$ over $\mathscr{C}_A$ is defined as

$$\mathscr{C}_A^{\mathscr{F}} = \{\mathscr{M} \in \mathscr{C}_A \mid \mathscr{M} \models Tr(\mathscr{F}_{\bar{P}})\}.$$

That is, every legal model must satisfy the obligations and prohibitions specified by $\mathscr{F}$.

But what about permissions? Permissions are actually a *check* that must be performed on $\mathscr{C}_A^{\mathscr{F}}$ to ensure coherence. The condition that $\mathscr{C}_A^{\mathscr{F}}$ must fulfill is the following: for every φ of the form $P(\psi)$ in $\mathscr{F}$ there must be a model $\mathscr{M}$ in $\mathscr{C}_A^{\mathscr{F}}$ such that $\mathscr{M} \models Tr(\psi)$. I.e., if something is permitted then the rest of the NSUA does not prevent it from happening.

We are going to expand the concept of coherence in the next section by analyzing other cases of interest.

4.3 Analyzing Coherence

A difficult topic in deontic logic is the concept of *coherence* of a normative system: whenever the set of rules is "contradictory" in any sense. As stated in the literature (e.g., [19]), the problem cannot be simply reduced to logical consistency. We take a pragmatic approach where a normative system is not coherent if it has any of a list of problems.

While some of them are straightforward to check, others require more sophistication. Let's see an example of each class. To fix notation, $\varphi\#\psi$ means that φ is incompatible with ψ and the following equivalences hold: $O(\varphi) = O_\bot(\varphi)$, $F_\rho(\varphi) = O_\rho(\neg\varphi)$.

To check that there are no forbidden obligations (i.e., that there is no pair of rules $O(\varphi)$ and $O(\psi)$ such that $\varphi\#\psi$), we need to check that there is at least one possible legal behaviour that satisfies both the background theory and the complete set of formulae. To do that, all the rules r_i are conjuncted into $\Phi = \bigwedge Tr(r_i)$, and both the automata and $\neg\Phi$ are fed to the model checker. If $\neg\Phi$ is not satisfiable the model checker will output a counter example trace, τ. τ satisfies the negation of $\neg\Phi$ so it is the legal behaviour we were looking for. Should $\neg\Phi$ be satisfiable, that means that Φ is not, so a backtracking-type of procedure should be started to find the "guilty" rules. How to improve this procedure is an active area of research.

[6] That is, the basic version of LTL with the standard boolean connectives plus the *until* operator (from which the diamond is defined).

We are also interested in checking that there are no "contradicting obligations": rules $O_\rho(\varphi)$ and $O_{\rho'}(\psi)$ with $\varphi\#\psi$ even if it is not the case that $\rho\#\rho'$. That is, incompatible obligations, but with compatible reparations. If that were the case, there would be a legal behaviour: doing the reparations ρ and ρ', yet it makes no sense that the primary obligations φ and ψ are impossible to comply with.

To check for that, we build Φ' as $\bigwedge Tr(O(\varphi_i))$ for all the rules $r_i = O_{\rho_i}(\varphi_i)$. Then the automata and $\neg\Phi'$ are fed to the model checker as before. If $\neg\Phi'$ is satisfiable there is no way to comply with all the obligations, leaving aside the possible reparations. If $\neg\Phi'$ is not, then, as before, the τ' counter example trace is a possible way of complying with all the primary obligations.

It should be noted that this last check is one of the analysis were preservation of input syntax is important and the translation of repared obligations is not done directly.

The following conditions also violate coherence:

- *Forbidden reparations*, such as $O_\rho(\varphi)$ and $F(\rho)$. In that case the reparation is only nominal, as it is indeed forbidden. $O_\rho(\psi)$ and $O(\psi)$ with $\rho\#\psi$ is another case of the same problem.
- *Obligations with conflicting reparations.* If $\rho\#\rho'$ and $O_\rho(\varphi)$ and $O_{\rho'}(\varphi)$ is found then there is a contradiction in how the obligation φ could be repaired. Note that this does not mean that there is no legal behaviour, as respecting φ is always allowed.
- *Impossible permissions.* Whatever was said to be permitted should be possible as was explained in Sections 3 and 4.2.
- *Unrealizable background theory.* The background theory should not generate an empty set of traces, which would mean it is, by itself, logically inconsistent.

5 Case Study

To show `FL` at work we analyze some excerpts from a university regulation. This case study focuses on conflicts that can arise from students being able to be also teachers. Although fictional, the inspiration is real. The case study features the use of actions, intervals, counters, obligations, prohibitions, permissions and many forms of coherence analysis.

The analyzed excerpt is the following:

1. Chapter 1, Students.
   (a) Every individual that has enrolled for a career and has not yet graduated from it is considered to be a student.
   (b) Students should respect each other. Major disciplinary faults are punished by forbidding the entering to university premises for one year after the fault.
   (c) Students have the following rights: ..., participate in research activities, ...
2. Chapter 2, Teachers.
   (a) There are three teaching categories: c1) Undergraduate Teaching Assistant (aka UTA), c2) Teaching Assistant and c3) Professor.
   (b) To become a teacher, aspirants must apply when the selection opens. The selection will be made based on the following criteria for each category: [omitted, not relevant to the case study]
   (c) To apply for the UTA category, aspirants must be students at the time of the selection.[7]

[7] The UTA category position lasts one year in the real case. This particular spelling of the norm was chosen because it is desired that only students fill this position, yet allow them to keep the job if they graduate after the selection.

(d) Teachers must perform their duties, starting 30 days after the selection ends.
(e) Working from home is allowed, but teachers must spend at least one day a week in the premises of the university.

3. Chapter 3, Research.
   (a) Research activities can only be pursued by members of approved research groups.
   (b) Research groups are conformed by accredited professors or teaching assistants.
4. Chapter 4, University Library.
   (a) Every borrowed book should be returned by the end of the month.[8]
   (b) Students and teachers are subject to a fine for not returning books in time.
   (c) As students are generally on a budget, their fine should be low.
   (d) Teachers should be an example of conduct, thus their fine should be strictly higher that the students' one.

Let's model the student's chapter first. To avoid clutter some abbreviations will be used, such as not declaring actions that are used to bound intervals if they do not take any extra parameter, as it is the case for declaring what a student is.

interval `student` **delimited by actions** `Enroll-Graduate`

Regarding discipline, they should not commit disciplinary faults or be banned from entering the premises for one year. To model that we will define two intervals: one that spans from the fault to one year after, and another that accounts for being inside the building.

interval `ban` **delimited by actions** `CommittFault-OneYearPassed`
interval `inside_building` **delimited by actions** `Enter-Exit`

And stipulate the prohibition:

$$F_{\Diamond_{\texttt{ban}}\neg \textit{is\_inside\_building}}(\texttt{CommitFault}) \tag{1}$$

meaning that the fault should not occur but if it does, during the ban period the student cannot be inside the building (*is_inside_building* is a boolean variable made true between the bounding actions of the interval).

Article 1c permits students to do research, in what can be thought of as a case of antithetical permission [26]: a permission set to invalidate future prohibitions.

$$P(\textit{is\_student} \wedge \texttt{DoesResearch}) \tag{2}$$

Now let's focus on the teachers' selection process. There is the selection interval and its possible outcomes: being elected in any of the categories, or not being elected at all.

action `ElectWinners` **output values** { teacher_c1, teacher_c2, teacher_c3, no_teacher }
interval `selection` **delimited by actions** `OpenSelection-ElectWinners`
action `Apply` **only occurs in scope** `selection`

For simplicity let's only model the requirements for the c1 category (UTAs) of still being a student:

$$O(\Diamond_{\texttt{selection}}(\texttt{Apply} \rightarrow \textit{is\_student})) \tag{3}$$

Teachers also have duties, that must start 30 days after the election:

[8] The more realistic requirement of returning within days is also possible but more involved, thus the simpler version is preferred for space reasons.

interval `grace_period` **delimited by actions** `ElectWinners-30DaysAfter`
interval `on_duty` **delimited by actions** `30DaysAfter-+inf`
interval `week` **delimited by actions** `StartWeek-EndWeek` **occurs only in scope** `on_duty` **repeatedly**

It is mandatory to have a weekly visit:

$$O(is\_teacher \rightarrow \Diamond_{\texttt{week}}\texttt{Enter}) \tag{4}$$

with *is_teacher* defined as[9]

macro *is_teacher* = `Apply` ∧
(`ElectWinners`.*teacher_c1* ∨ `ElectWinners`.*teacher_c2* ∨ `ElectWinners`.*teacher_c3*)

Regarding Chapter 3, the restriction of research activities to members of research groups is:

$$F(\neg\texttt{JoinResearchGroup} \wedge \texttt{DoesResearch}) \tag{5}$$

while the requirements of being TA or professor to belong to a research group is:

$$F(\neg(\texttt{ElectWinners}.teacher\_c2 \vee \texttt{ElectWinners}.teacher\_c3) \wedge \texttt{JoinResearchGroup}) \tag{6}$$

Then there is the borrowing of books, similar to both students and teachers, that requires the `bbc` (borrowed books count) counter and signaling months (it should be noted that the exact duration of a month is not important and it is thus abstracted away):

counter `bbc` **increases with action** `BorrowBook` **decreases with action** `ReturnBook`
interval `month` **delimited by actions** `MonthBegin-MonthEnd` **repeatedly**

Although articles 4c and 4d are mainly motivational, there is one prescriptive consequence, even at the level of abstraction we are using – fines should be different: `StudentFine` # `TeacherFine`.

Article 4b is a CTD to 4a, so we encode them as:

$$O_{\texttt{StudentFine}}(is\_student \rightarrow \Diamond_{\texttt{month}}(\texttt{bbc} > 0 \rightarrow \Diamond_{\texttt{month}}\texttt{bbc} = 0)) \tag{7}$$

$$O_{\texttt{TeacherFine}}(is\_teacher \rightarrow \Diamond_{\texttt{month}}(\texttt{bbc} > 0 \rightarrow \Diamond_{\texttt{month}}\texttt{bbc} = 0)) \tag{8}$$

When analyzed by FormaLex three coherence problems are pointed out. First, the reparation for not returning borrowed books is troublesome for teachers that are also students, and the tool signals a case of *conflicting reparations*, as there are traces where the implicit agent is indeed a teacher and a student. Looking at the NSUA nothing impedes students to become teachers, quite the contrary, as there is a UTA category.

What type of fine should a UTA be subject to? Whatever conclusion can be obtained by looking at the motivational articles 4c and 4d is disputable, as it is both true that UTAs should be an example of conduct because they are teachers, and that they are also on a budget, because UTAs' salaries are symbolic. As the fining would probably be decided by a library's clerk, there is high risk of different solutions applied to identical cases. It would be much better if this case could be decided by the norm-givers, and that is the sense of the warning.

The second problem is more involved and related to the weekly visit rule. The tool detects that the reparation of the rule that forbids discipline faults (1) contradicts the obligation to the weekly visit (4). Indeed, there is the possibility of a student committing a fault, thus being banned to enter the premises, applying for a teaching position, then being elected as UTA and not being able to comply with his weekly visit duty.

A possible solution is to forbid the application of punished students as in:

[9]Note that although the name of the `Apply` action is in the present tense, because of the translation, it is easier read if it thought of as written in the past tense. This reflects the fact that the intended semantics is to check if the action has occurred at some point in the past. The same happens with formulae 5, 6 and 9.

action `Apply` **only occurs in scope** `selection` **requires that** ¬`CommitFault`

However, the problem persists, as the student can now apply (not yet being faulty), commit the fault, then being selected, to the same effect. A better solution would be to restrict the `ElectWinners` action so only non faulty students can become teachers:

$$F(\texttt{CommitFault} \wedge \texttt{ElectWinners}.\mathit{teacher\_c1}) \tag{9}$$

This formula states that it is forbidden to commit the fault and to be elected UTA. Observe that if the fault is commited before becoming a teacher, then this formula forbids the election, which is the primary intention of it. On the other hand, if the fault is commited after becoming UTA, then again the reparation of the rule that forbids discipline faults (1) contradicts the obligation to the weekly visit (4). This situation is also noticed by the tool and notified to the user as a warning. If the user considers that it should be possible to repair the fault even under these conditions, then she must introduce appropriate modifications in the NSUA to envisage this situation. If, on the other hand, she thinks that it is correct to tighten the rules for teachers so they should not have a way to repair their faults, then the warning can be ignored.

The third problem is the collision of allowing research only to TAs and professors (rules 5 and 6) with the permission for students by rule 2. The fix is either the removal of the permission or the inclusion of at least UTAs into research groups.

There is another interesting aspect of this NSUA. Let's assume somebody proposes a different writing for article 2c, that is supposedly more faithful to the spirit of not letting graduates fill the UTA category. She proposes defining what a graduate is:

interval `graduate` **delimited by actions** `Graduate-+inf`

and plain forbidding the application of graduates. When queried about the validity of this new writing:

? $F(\Diamond_{\texttt{selection}}(\texttt{Apply} \wedge \mathit{is\_graduate}))$

the tool responds that the prohibition does not hold as there is a trace where a student graduates and then enrolls again (say, for a different career) before applying. That perfectly satisfies the requirement that during the selection process applications must be done by students (3), as the implicit agent is *also* a student. It means that the phrasing of the rule (3) does not comply with the intended normative effects: restrict the UTA category to non-graduates; so the proposed alternate writing is actually the correct one. This "bug" is extracted from a real university's regulation.

An interesting remark is that although the lack of roles in `FL` can be seen as a drawback, it turned useful in this case. Should roles had been available, probably many of the above mentioned bugs would have been concealed, including the real one.

6 Related Work

The idea of using a temporal logic for deontic purposes is not new. It can be traced to [1, 22, 16, 8, 11, 7, 23], among others who use some type of temporal-deontic logic. However, as far as we know they provide neither tool support nor a translation into a standard tool, and thus are not directly comparable to our work which is heavily tool-biased.

[18] deals with automated conflict detection in norms by using a tool that supports ontologies and translates normative propositions into a Prolog program, but the analysis is restricted to logical contradiction. More similar to our approach of analyzing contracts and regulations for

coherence problems are BCL [15] and $\mathscr{CL}$ [24]. BCL is a contract specification language that is meant for monitoring, allows to build executable versions, can detect conflicts among rules off-line and provides features like clause normalization. However, it lacks support for temporal reasoning and background theories.

$\mathscr{CL}$ is another logical language based on dynamic logic that treats deontic operators as first-class citizens. It is based on an ad-hoc tool and it neither uses background theories, nor discusses how to deal with some limitations of expressivity: for instance, in dynamic logic approaches it is easy to say that if a book is borrowed (b), a book should be returned (r) as [b]O(r), but such rule matches the borrowing of multiple books and the returning of just one; the correct version is pretty involved or plain impossible depending on the particular logic.

7 Conclusions

Starting from the premise that normative systems are very similar to software specifications we propose a language and related tool set to analyze the former with tools designed for the latter. Besides the similarities, the decision is based on avoiding to build tools from scratch: model checkers have decades of effort in tool building, optimizations and expressivity improvements.

In this article we analyzed a feature-rich, real-life inspired, case study. Although fictional, we believe it shows the power of both the tool and its underlying definition of coherence to spot defects that are not self-evident. Our next step is dealing with a 100% real case study to investigate the payoff of having to logically encode the NSUA vs. the severity of the defects found.

References

[1] Thomas Ågotnes, Wiebe van der Hoek, Juan A. Rodrıguez-Aguilar, Carles Sierra & Michael Wooldridge (2009): *A Temporal Logic of Normative Systems. Towards Mathematical Philosophy* 28, pp. 69–106, doi:10.1007/978-1-4020-9084-4_5.

[2] C.E. Alchourrón & E. Bulygin (1981): *The expressive conception of norms. New studies in deontic logic* 152, pp. 95–124.

[3] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai & P. Šimeček (2006): *DiVinE – A Tool for Distributed Verification (Tool Paper)*. In: *Computer Aided Verification*, *LNCS* 4144/2006, Springer Berlin / Heidelberg, pp. 278–281, doi:10.1007/11817963_26.

[4] R. Bharadwaj & C. Heitmeyer (2002): *Developing high assurance avionics systems with the SCR requirements method*. In: *Digital Avionics Systems Conferences, 2000. Proceedings. DASC. The 19th*, 1, IEEE, doi:10.1109/DASC.2000.886888.

[5] Guido Boella & Leendert van der Torre (2003): *Permissions and obligations in hierarchical normative systems*. In: *ICAIL '03: Proceedings of the 9th international conference on Artificial intelligence and law*, ACM, New York, NY, USA, pp. 109–118, doi:10.1145/1047788.1047818.

[6] Guido Boella, Leendert W. N. van der Torre & Harko Verhagen, editors (2007): *Normative Multi-agent Systems, 18.03. - 23.03.2007. Dagstuhl Seminar Proceedings* 07122, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[7] Jan Broersen, Frank Dignum, Virginia Dignum & John-Jules Ch. Meyer (2004): *Designing a Deontic Logic of Deadlines*. In Alessio Lomuscio & Donald Nute, editors: *DEON*, *Lecture Notes in Computer Science* 3065, Springer, pp. 43–56, doi:10.1007/978-3-540-25927-5_5.

[8] Julien Brunel, Jean-Paul Bodeveix & Mamoun Filali (2006): *A State/Event Temporal Deontic Logic*. In Lou Goble & John-Jules Ch. Meyer, editors: *DEON*, *Lecture Notes in Computer Science* 4048, Springer, pp. 85–100, doi:10.1007/11786849_9.

[9] A. Cimatti, E. Clarke, F. Giunchiglia & M. Roveri (2000): *NuSMV: a new symbolic model checker*. *International Journal on Software Tools for Technology Transfer (STTT)* 2(4), pp. 410–425, doi:10.1007/s100090050046.

[10] D. Corapi, M. De Vos, J. Padget, A. Russo & K. Satoh (2010): *Norm Refinement and Design through Inductive Learning*. In: *11th International Workshop on Coordination, Organization, Institutions and Norms in Multi-Agent Systems*, pp. 33–48, doi:10.1007/978-3-642-21268-0_5.

[11] Tim French, John Christopher McCabe-Dansted & Mark Reynolds (2010): *Axioms for Obligation and Robustness with Temporal Logic*. In Guido Governatori & Giovanni Sartor, editors: *DEON*, *Lecture Notes in Computer Science* 6181, Springer, pp. 66–83, doi:10.1007/978-3-642-14183-6.

[12] D. Giannakopoulou & J. Magee (2003): *Fluent model checking for event-based systems*. *ACM SIGSOFT Software Engineering Notes* 28(5), p. 266, doi:10.1145/940103.940106.

[13] Daniel Gorín, Sergio Mera & Fernando Schapachnik (2010): *Model Checking Legal Documents*. In: *Proceedings of the 2010 conference on Legal Knowledge and Information Systems: JURIX 2010*, pp. 111–115, doi:10.3233/978-1-60750-682-9-151.

[14] Daniel Gorín, Sergio Mera & Fernando Schapachnik (2011): FormaLex – *A Software Tool for Legal Drafting*. Technical Report GMS-2011, FCEyN, Universidad de Buenos Aires. Available at `http://publicaciones.dc.uba.ar/Publications/2011/GMS11`.

[15] G. Governatori & D.H. Pham (2009): *Dr-contract: An architecture for e-contracts in defeasible logic*. *International Journal of Business Process Integration and Management* 4(3), pp. 187–199, doi:10.1504/IJBPIM.2009.030985.

[16] Guido Governatori, Antonino Rotolo & Giovanni Sartor (2005): *Temporalised Normative Positions in Defeasible Logic*. In: *ICAIL*, ACM, pp. 25–34, doi:10.1145/1165485.1165490.

[17] Nienke den Haan (1997): *Tools for automated legislative drafting*. In: *ICAIL '97: Proceedings of the 6th international conference on Artificial intelligence and law*, ACM, New York, NY, USA, p. 252, doi:10.1145/261618.261662.

[18] S. Hagiwara & S. Tojo (2006): *Discordance Detection in Regional Ordinance: Ontology-based Validation*. In: *Proceeding of the 2006 conference on Legal Knowledge and Information Systems: JURIX 2006: The Nineteenth Annual Conference*, IOS Press, pp. 111–120.

[19] Jörg Hansen, Gabriella Pigozzi & Leendert W. N. van der Torre (2007): *Ten Philosophical Problems in Deontic Logic*. In Boella et al. [6].

[20] Holzmann (2003): *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, doi:10.1109/32.588521.

[21] David Makinson & Leendert W. N. van der Torre (2007): *What is Input/Output Logic? Input/Output Logic, Constraints, Permissions*. In Boella et al. [6], pp. 383–408.

[22] J.J.C. Meyer, R.J. Wieringa & F.P.M. Dighum (1998): *The role of deontic logic in the specification of information systems*. In: *Logics for Databases and Information Systems*, Kluwer, pp. 71–116.

[23] G. Piolle (2010): *A Dyadic Operator for the Gradation of Desirability*. *Deontic Logic in Computer Science* 6181, pp. 33–49, doi:10.1007/978-3-642-14183-6_5.

[24] Cristian Prisacariu & Gerardo Schneider (2009): *$\mathscr{CL}$: An Action-Based Logic for Reasoning about Contracts*. In: *WoLLIC '09: Proceedings of the 16th International Workshop on Logic, Language, Information and Computation*, Springer-Verlag, Berlin, Heidelberg, pp. 335–349, doi:10.1007/978-3-642-02261-6_27.

[25] Ellis Solaiman, Carlos Molina-jimenez & Santosh Shrivastava (2003): *Model Checking Correctness Properties of Electronic Contracts.* In: *Proceedings of the International conference on Service Oriented Computing (ICSOC03)*, Springer-Verlag, pp. 303–318, doi:10.1007/978-3-540-24593-3_21.

[26] Audun Stolpe (2010): *A theory of permission based on the notion of derogation. J. Applied Logic* 8(1), pp. 97–113, doi:10.1016/j.jal.2010.01.001.