Towards Incorporating Normative Requirements In Autonomous Systems Using Datalog

Mahrokh Mirani[1]

[1]GSSI (Gran Sasso Science Institute), L'Aquila, Italy

Abstract

The increasing integration of AI components in autonomous systems, particularly multi-robot systems, poses new regulatory challenges under the EU AI Act. In this work, we address the formalization and analysis of normative (SLEEC) requirements—those capturing social, legal, ethical, empathetic, and cultural constraints—essential for ensuring compliance with such regulations. We focus on the problem of obligation inference, which refers to selecting the necessary obligations, while considering a set of normative rules. We demonstrate our approach through an extension of NASA's FRET tool, as a proof of concept. This extension enables the user to specify, formalize, and translate SLEEC rules into both LTL and Datalog for runtime monitoring and decision-making.

1. Introduction

Normative requirements are non-functional requirements that specify social, ethical or legal constraints of a system. As the AI systems become more pervasive and interact with humans on a daily basis, they need to satisfy critical properties, respecting more human-centred values. Such properties are usually extracted by experts in fields other than formal verification, e.g., lawyers, psychologists, philosophers, etc. This highlights the elicitation and formalization of these properties as a primary concern.

The AI Act is the recent regulation adopted in the European Union countries to ensure safer and more secure use of artificial intelligence technologies. The European Union's regulation on AI classifies systems into four groups according to the risks they pose to users. These four levels are: 1. *Unacceptable risk* (e.g., social scoring systems), 2. *High risk* (e.g., medical devices), 3. *Limited risk* (e.g., deepfakes), and 4. *Minimal or no risk*. Each of these groups is treated with the necessary regulations to provide safety. Most of the AI Act targets systems with high risk (the second group). It includes many systems, including those used as safety components or for profiling individuals. These systems should provide enough evidence to prove compliance with requirements and remain compliant during all stages of the deployment of the system. However, it is not explained how this compliance should be checked by the providers of these systems. This raises the need for developing new testing methods and adapting the existing ones to focus on the specific needs of this domain.

Multi-robot systems often incorporate AI components for tasks such as autonomous decision-making, bringing them under the scope of the AI Act regulations. This brings up the problem of analyzing these systems and checking their compliance with the requirements. This problem can be broken into various sub-problems, e.g., requirement elicitation, formalization, consistency checking, validation and verification. Bennaceur et. al. in [1] count the main activities in the requirement engineering process as elicitation, modelling and analysis, assurance, and management and evolution. Each of these steps has its respective challenges and open issues in the field of robotic systems. For example, for the elicitation process, FRET (Formal Requirements Elicitation Tool) is a tool proposed to fill the gaps between requirements in natural language and formal specification of them by introducing a structured specification language close to English. For the analysis of the requirements, many tools and techniques are proposed to monitor, test, and in some cases formally verify requirements in a robotic system. However, there is no integrated solution for the specification and analysis of normative

ICLP DC 2025: 21st Doctoral Consortium on Logic Programming, September 2025, Rende, Italy.

requirements in robotic areas.

The autonomous systems that incorporate AI components usually have to follow dynamic decision-making procedures, and in many cases, this process has to be done in real time. This makes the efficiency of the decision-making process of high importance. In this work, we also try to tackle the efficient decision-making problem by incorporating optimized logical reasoning techniques. In section 2, we will give a background on normative requirements and tools and methods used for our purpose. In Section 3, the main approach to the problem is explained, and finally, in the section 4, challenges and future directions are discussed.

2. Background

In this section, we will explain the current state of the art for SLEEC requirements and previous works on how to perform logical reasoning with them. We will also give a short introduction of Datalog, which we have used for efficient obligation inference.

2.1. Normative Requirement Specification

Specifying and formalizing the normative requirements for AI can be challenging due to the ambiguities of these requirements and Townsend et. al. [2] introduce a procedure to elicit SLEEC properties from high-level principles considering the capabilities and limitations of the robot agent. Their proposed approach consists of five steps as follows:

1. Identifying Norms and Normative Principles
2. Mapping Principles to Agent Capabilities
3. Identifying SLEEC Concerns
4. Identifying and Resolving SLEEC Conflicts
5. Labelling, Identifying Impact, and Re-assessing Complex Rules

In the first step, the goal is to extract a list of general norms and principles regarding the context of the robot and its objectives. In the next step, the robot's capabilities such as input or output devices are taken into account. In this step, these capabilities are mapped with SLEEC principles and how they can affect the principles. This step may also result in recognizing the need for supporting additional capabilities. In the third step, SLEEC concerns are derived regarding the system processes. Especially, the legal concerns that might be related to the system at hand should be listed in this step, alongside other cultural, social, empathetic, and ethical concerns. During the fourth step, the conflicting norms are extracted and stakeholders need to decide in which cases which norm is superior and based on that which action should be taken. In the last step, the SLEEC rule should be labelled based on the norms that it considers in each case and be reevaluated across other SLEEC concerns. In the case of further conflicts, the last three steps should be repeated as many times as needed until there are no more exceptions to be added.

The resulting SLEEC rule consists of a general case with a recursive set of exceptions that are expressed with the structure "unless ... in which case ...". This structure is based on [2] who believe that moral principles have exceptions that can be captured by "unless" and these cases don't follow the normal form. The resulting SLEEC rule is in the following form, and consecutive works also follow the same pattern:

$$\begin{gathered}\text{IF } C_0 \text{ THEN DO } O_0,\\ \text{UNLESS } C_1 \text{ IN WHICH CASE DO } O_1,\\ \text{UNLESS } C_2 \text{ IN WHICH CASE DO } O_2,\\ \dots\\ \text{UNLESS } C_n \text{ IN WHICH CASE DO } O_n.\end{gathered} \tag{1}$$

2.2. Obligation Inference

The problem of obligation inference is to compute the output response based on a set of inputs and a set of rules, in a way that no rule is violated. To better understand the problem, here is the formal definition of it:

Definition 2.1. *The problem of* obligation inference *is defined as follows: Given a set of FRETish-SLEEC rules, an observation set $A_1(\vec{c_1}), \dots, A_n(\vec{c_n})$, and an obligation $B(\vec{c_0})$, is $B(\vec{c_0})$ entailed?*

Considering the SLEEC format introduced in section 2.1, the goal is to find efficient ways to perform the obligation inference. In this format, $\{C_0, C_1, \dots, C_n\}$ is the set of observations and $\{O_0, O_1, \dots, O_n\}$ is the set of obligations to compute.

There are two different parses for this pattern, and different works in the literature use them arbitrarily. The difference is shown in table 1. Ultimately, they are equivalent from a computational perspective. however, it is left to the authors to decide between them according to simplicity and comprehensibility for the user. To avoid confusion, we use the first one for the rest of this paper, but everything can be extended to include the other parsing too.

IF C_0 THEN ((...(DO O_0),	IF C_0 THEN (DO O_0,
UNLESS C_1 IN WHICH CASE DO O_1),	UNLESS C_1 IN WHICH CASE (DO O_1,
UNLESS C_2 IN WHICH CASE DO O_2),	UNLESS C_2 IN WHICH CASE (DO O_2,
...	...
UNLESS C_n IN WHICH CASE DO O_n).	UNLESS C_n IN WHICH CASE (DO O_n)...)).
(giving highest priority to O_n)	(giving highest priority to O_0)

Table 1
two different parsings for SLEEC rule format

Troquard et. al. [3] offer a transformation of these SLEEC rules into classical logic. Here we explain briefly how this is done. In this compilation, each line of the formula is broken into a logical clause and satisfaction of the conjunction of these clauses results in satisfaction of the formula. For each line, the status of all of the previous lines needs to be checked. One obligation will be conducted only if the next unless does not happen. So obligation O_i depends on the satisfaction of all the previous observations $(C_0, C_1, \dots, C_i)$ and also dissatisfaction of C_{i+1}. Therefore, the transformation for the formula 1 is the following:

$$\begin{aligned} \phi \quad = \quad & \left[\bigwedge_{0 \le i \le n-1} \left(\left(\bigwedge_{0 \le j \le i} (C_j) \wedge \neg C_{i+1} \right) \rightarrow O_i \right) \right] \\ & \wedge \left[\left(\bigwedge_{0 \le j \le n} (C_j) \right) \rightarrow O_n \right] \end{aligned} \tag{2}$$

2.3. Datalog

Datalog is a declarative programming language for logical inference. It is different from Prolog in that it is specifically developed to perform inference on sets of data. Soufflé is a high-performance Datalog engine designed especially for program analysis, and has proved to show good performance on large datasets [4].

A program in the Soufflé implementation of Datalog must contain the declaration of all relations used in the program. It also needs to specify explicitly which relations constitute the inputs and outputs. The body of the rules are defined in the program as conjunctions (shown by character ',') and disjunctions (shown by character ';') between relations. For example, consider the case that a relation A for a number is true if that number satisfies both relations B and C. We have relations B, C and want to derive A. This can be defined using the following program:

```
.decl A(x: number) .
.decl B(x: number) .
.decl C(x: number) .
A(x) :- B(x), C(x) .
.input B
.input C
.output A
```

A relation without inputs can be used as a constant.

3. Current Work

As explained before, the efficiency in the decision-making process by AI is of high importance. We try to address this problem by transforming the SLEEC rules into Datalog logic programs and then employing Soufflé for logical inference.

In this section, the explanations are developed through an example. Consider an assistant robot helping to take care of a patient in a nursing home. It is a well-known example in the literature. This robot has to respect some rights of the user. Stakeholders reach a SLEEC rule after discussing the priority of various rights of the user. The resulting rule is as follows:

> If the user asks to open the window, then open the window,
> Unless the user is underdressed, in which case do not open the window and inform the user of the reason behind this inaction,
> unless the user is highly distressed, in which case open the window.

FRET (Formal Requirement Elicitation Tool) is a tool developed by NASA for the elicitation, formalization and understanding of requirements. It is widely used in industry for formalizing requirements and generating future- and past-time LTL formulas. It also offers realizability checking and interactive simulation of the requirements. We added an extension to this tool to formulate SLEEC rules and generate future-time LTL formulae. In addition, this extension provides the user with the Datalog code for obligation inference and runtime monitoring of the requirements.

In this extension of FRET, the SLEEC rule for the assistant robot can be specified as:

> If ask_open(u,w) then open(w), unless underdressed(u) in which case not_open(w) & inform(u), unless highly_distressed(u) in which case open(w)

Based on the compilation explained in section 2.2, the following logical rule is obtained:

$$\begin{gathered} ask\_open(u,w) \wedge \neg underdressed(u) \rightarrow open(w) \wedge \\ ask\_open(u,w) \wedge underdressed(u) \wedge \neg highly\_distressed(u) \rightarrow inform(u) \wedge not\_open(w) \wedge \\ ask\_open(u,w) \wedge underdressed(u) \wedge highly\_distressed(u) \rightarrow open(w) \end{gathered} \tag{3}$$

Figure 1 shows the specification of this rule in our Fret extension, and on the right panel, the LTL formula and the Datalog program for obligation inference and runtime monitoring are generated. In the next subsections, these programs are explained.

3.1. Obligation Inference

To generate the Datalog code for the obligation inference, one Datalog rule is generated for every obligation. The formula 2 is broken into *n+1* implications, and if the conditions of an obligation are satisfied, then this obligation gets the value "true" in the outputs. To evaluate these conditions, observations are treated as input data for the Datalog program. This data should be supplied by the system, for example, via sensors.

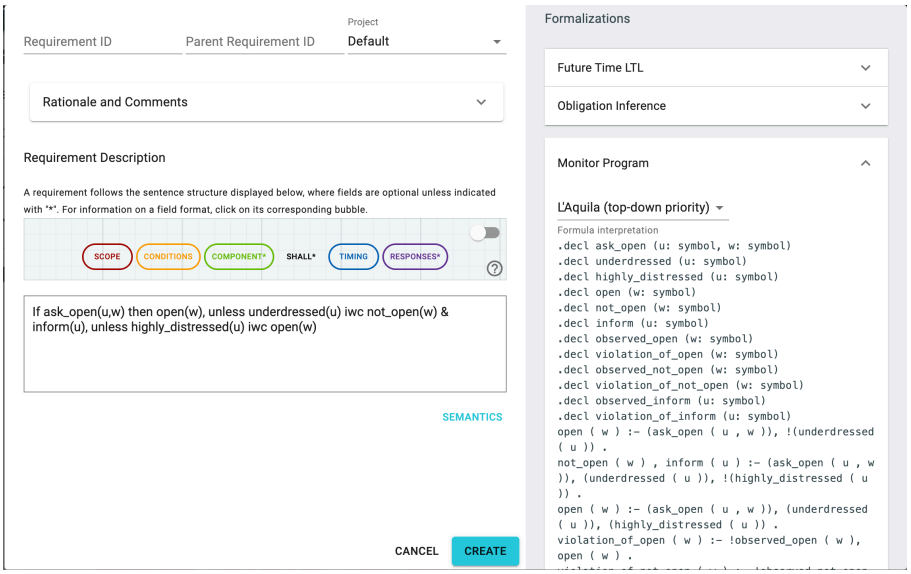

Figure 1: FRET requirement creation window with assistant robot rule specification

The rules for deriving obligations are expressed as logical operations over predicates. As customary, constants are represented as predicates by introducing an auxiliary variable. For instance, if a rule states that support should be called when the temperature exceeds 30 degrees, it would be expressed in our extension as follows:

```
IF temperature(t:number) & t > 30 THEN callSupport()
```

The Datalog rule to compute this obligation will be generated as follows:

```
callSupport() :- temperature(t), t > 30.
```

3.2. Runtime Monitoring

The goal of the runtime monitor is to observe the system while running and raise a warning whenever a violation of the rules happens. For this purpose, the runtime monitoring works in two steps. In the first step, the same as the previous section, obligation inference is done. This time, in addition to input data, the observed actions are another group of inputs. The second step compares the observed actions and the inferred obligations, and if an inferred obligation does not happen, a violation has occurred.

Formally, a new boolean predicate is introduced for each obligation, and its value is set to "true", if and only if that obligation is inferred to be done, but it is not observed to be done in reality. These predicates are introduced as the outputs of the program.

For example, if a rule contains an obligation called *callSupport*(), then an input *observed_callSupport*() and an output called *violation_of_callSupport*() are defined, and the following rule is added to the monitoring program:

```
violation_of_callSupport() :- callSupport() , ! observed_callSupport() .
```

4. Conclusions and Future Directions

We try to tackle the problem of obligation inference and runtime monitoring specific to normative requirements. We use Datalog as an inference engine to decide on obligations in an AI system. This approach shows a good performance considering the data complexity and rule complexity. So far, we have limited these rules to only include predicates as both observations and obligations. In a real-world scenario, many requirements demand a notion of time. For example, some obligations need to be

carried out within a certain amount of time, but in our work, we have forced the obligation predicate to take effect immediately. This issue can be solved by defining timed predicates and enforcing the timing constraints internally to the system. However, explicit times can be more practical in some contexts and can put an extra focus on timing during the requirement engineering process.

It is also needed to define sequences of actions with a specific ordering. A very common example of such an obligation is known as patrolling, which means a sequence of locations that a robot should visit in order. For that purpose, it is also needed to have a temporal logic like LTL to be able to monitor a program.

Declaration on Generative AI

The author affirms that no generative artificial intelligence (AI) tools were employed in the conception, analysis, writing, or revision of this manuscript. All content and interpretations presented herein are the sole work of the author.

References

[1] A. Bennaceur, T. T. Tun, Y. Yu, B. Nuseibeh, Requirements Engineering, Springer International Publishing, Cham, 2019, pp. 51–92. URL: https://doi.org/10.1007/978-3-030-00262-6_2. doi:10.1007/978-3-030-00262-6_2.

[2] B. Townsend, C. Paterson, T. T. Arvind, G. Nemirovsky, R. Calinescu, A. Cavalcanti, I. Habli, A. Thomas, From pluralistic normative principles to autonomous-agent rules, Minds and Machines 32 (2022) 683–715. URL: https://doi.org/10.1007/s11023-022-09614-w. doi:10.1007/s11023-022-09614-w.

[3] N. Troquard, M. De Sanctis, P. Inverardi, P. Pelliccione, G. L. Scoccia, Social, legal, ethical, empathetic, and cultural rules: Compilation and reasoning, Proceedings of the AAAI Conference on Artificial Intelligence 38 (2024) 22385–22392. URL: https://ojs.aaai.org/index.php/AAAI/article/view/30245. doi:10.1609/aaai.v38i20.30245.

[4] H. Jordan, B. Scholz, P. Subotić, Soufflé: On synthesis of program analyzers, in: Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28, Springer, 2016, pp. 422–430.