Accredited SINTA 2 Ranking
Decree of the Director General of Higher Education, Research, and Technology, No. 158/E/KPT/2021
Validity period from Volume 5 Number 2 of 2021 to Volume 10 Number 1 of 2026

Published online at: **http://jurnal.iaii.or.id**

JURNAL RESTI
(Rekayasa Sistem dan Teknologi Informasi)
Vol. 9 No. 2 (2025) 343 - 356 e-ISSN: 2580-0760

Large Language Model-Based Extraction of Logic Rules from Technical Standards for Automatic Compliance Checking

Rizky Nugroho[1]*, Adila Krisnadhi[2], Ari Saptawijaya[3]
[1,2,3]Faculty of Computer Science, Universitas Indonesia, Depok, Indonesia
[1]rizky.prasetya21@ui.ac.id, [2]adila@ui.ac.id, [3]saptawijaya@ui.ac.id

Abstract

In this research, we design logic rules as a representation of technical standards documents related to ship design, which will be used in automatic compliance checking. We present a novel design of logic rules based on a general pattern of technical standards' clauses that can be produced automatically from text using a large language model (LLM). We also present a method to extract said logic rules from text. First, we design data structures to represent the technical standards and logic rules used to process the data. Second, the representation of technical standards is produced manually and tested to ensure that it can give the same conclusion as human judgment regarding compliance. Third, a variation of prompting methods, namely pipeline method and few-shot prompting, is given to LLM to instruct it to extract logic rules from text following the design. Evaluation against the logic rules produced shows that the pipeline method gives an accuracy score of 0.57, a precision of 0.49, and a recall of 0.62. On the other hand, logic rules extracted using few-shot prompting have an accuracy score of 0.33, precision of 0.43, and recall of 0.5. These results show that LLM is able to extract a logic rule representation of technical standards. Furthermore, the representation resulting from the prompting technique that utilizes the pipeline method has a better performance compared to the representation resulting from few-shot prompting.

Keywords: automatic compliance checking; logic rules; technical standards; large language model; prompting

How to Cite: R. Nugroho, A. Krisnadhi, and A. Saptawijaya, "Large Language Model-Based Extraction of Logic Rules from Technical Standards for Automatic Compliance Checking ", *J. RESTI (Rekayasa Sist. Teknol. Inf.)* , vol. 9, no. 2, pp. 343 - 356, Apr. 2025.
DOI: https://doi.org/10.29207/resti.v9i2.6285

1. Introduction

In the context of shipping, classification is a process of verifying whether a vessel (ships or other floating structures) complies with technical standards throughout the vessel's life [1]. This verification is done by an~~organizatio~~ independent governing body called a *classification society*~~.~~. Such a classification society usually works at a national level, such as Biro Klasifikasi Indonesia (BKI) in Indonesia, or sometimes it extends beyond national borders, such as the American Bureau of Shipping in the US.~~.~~ A classification society checks whether the design documents of a vessel, as well as the corresponding (physically built) vessel itself, against recognized technical standards through document examinations and physical surveys.

However, the compliance checking is still done manually. Typically, a (human) reviewer reads the technical standards to obtain the design requirements. Next, the data regarding the vessel whose compliance is to be checked is gathered from the design documents. Finally, the conclusion about compliance is made by checking the data against the requirements of technical standards. This manual process has many weaknesses, mainly because it is prone to error and its low efficiency [2].

Compliance checking is a common task in engineering, construction, and architecture. As a result, there have been a number of research works done concerning the automation of the process. Given an object whose compliance with the standards is to be checked, several approaches utilize a knowledge graph to capture the object's data and SPARQL queries [3], sometimes extended with custom-built rules, to represent the requirements~~.~~ [2], [4]-[9]. Others such as Xue and Zhang [10] use logic programming rules to represent technical standards, while Ren et al. [11] and Liu et al. [12] employ deep learning models such as BERT [13] and Bi-LSTM to learn the pattern of the technical

Received: 11-01-2025 | Accepted: 12-04-2025 | Published Online: 17-04-2025

standards and compare them with the data to determine the compliance.

The aforementioned approaches possess several shortcomings. They either obtain representation through a manual process, depend on specific (natural language) sentence patterns, or need a large, labeled dataset for training deep learning models. A manual process, such as that employed in [2], [4], [6], [7], [9] requires intensive participation by the corresponding domain experts, e.g., ship designers, structural engineers, etc. Unfortunately, such domain experts are not easily available. Meanwhile, methods that employ rule-based extraction, which rely on specific sentence patterns, such as those proposed in [5], [8], risks the possibility of unaccounted patterns that may appear when the standards are updated in the future. Finally, a labeled dataset containing annotated text of technical standards [11], [12] may not be available for the domain in which the compliance checking is done, for example, electrical installations or structural design of ships.

In this paper, we propose an approach that can avoid those shortcomings. Our approach makes use of recently developed generative LLMs, like GPT and its variants [14], Llama [15], and PaLM [16] to translate the requirements from the standards into a set of logic programming rules, which can then be run against data, represented as logic facts, obtained from the object for whom the compliance is to be checked. Specifically, we focus on compliance checking of vessels and their design documents with respect to the standards provided by BKI.

Our approach leverages the ability of LLMs to extract structured data from natural language texts through prompt engineering, as shown in [17]-[19] to obtain a logic rules representation of technical standards. Employing LLM in our approach means we are able to automate the generation of logic rules representation without relying on intensive participation of domain experts. In addition, the utilization of LLM through prompt engineering lets us avoid the need for annotated datasets. Lastly, we define a general pattern of sentences found in the technical standard's text to guide the LLM in the translation process. We ensure the use of a pattern general enough to be applied to any text in the technical standards, which means it will be flexible enough to account for any other sentence pattern that may be introduced in the future.

In this research, logic programming rules are used as the representation of technical standards. We argue that a logic programming rule is the most appropriate representation for technical standards since it is inherently able to represent conditional sentences that are prevalent in technical standards. On the other hand, other representations, such as a knowledge graph, need a more complex form, such as shown in [5], [6].

Overall, the contribution of this work is threefold. First, we design logic programming rules that can be used to represent technical standards in the ship construction domain. Second, we develop prompts to extract the logic rules as designed from texts using LLM. To the best of our knowledge, there is no prior research that employs LLM to extract logic rules from texts. Last, we work on new documents that are technical standards published by BKI.

This paper is organized as follows. After the introduction, we discuss our research methods in section 2, starting with a brief explanation of the compliance checking process and logic programming basics. Afterwards, we will explain our proposed method, which consists of the design of the logic rules, the prompt engineering we do to generate them, and the evaluation scenarios. Then, we present the results and discussion in Section 3. Lastly, we present the conclusion of our research in Section 4.

2. Research Methods

We first briefly explain the basic definitions relevant to the compliance checking process for vessel design and construction, as well as basic concepts from logic programming. Following these, we describe how the requirements from the technical standards can be represented as logic rules and then a prompt engineering approach to actually produce such logic rules from the requirements in the technical standards, which are expressed in complex natural language sentences.

2.1 Compliance Checking Process

Generally, compliance checking involves three components: requirements from technical standards, the objects being assessed, complete with their property, and the reviewer (human). The reviewer will read the technical standards and find the requirements. He will then gather the object's properties from the design documents. Afterwards, he will compare the property with the requirement to determine whether the object complies with the requirement. In some cases, the reviewer may also conclude that the requirement is not applicable to the object. We will use the term *compliance decision* to refer to such a conclusion throughout this paper.

We present an example of the compliance checking process using the following clause: "Emergency switchboard shall be installed above the uppermost continuous deck and behind the collision bulkhead." This clause contains one sentence in which a requirement must be fulfilled by an object. In this clause, the object is "emergency switchboard" and it must satisfy the requirement that is, "be installed above the uppermost continuous deck and behind the collision bulkhead." A reviewer will then consult the design document to check the position where the emergency switchboard is installed. If the position is simultaneously above the uppermost continuous deck and behind the collision bulkhead, then the clause is complied. Inversely, if the position does not satisfy the requirement, then the clause is not complied with.

In automatic compliance checking, the machine will do all the compliance checking processes from gathering the requirements to making the conclusion, as shown in Figure 1. Within the context of this research, requirements inside clauses are represented by logic rules, compliance decisions are obtained by executing the logic rules, and the extraction of requirements is done using LLM. To increase efficiency, the machine should also do the data extraction from documents related to compliance checking objects. However, for this research, data extraction is done manually. We will leave it for future research to include the extraction of data automatically.

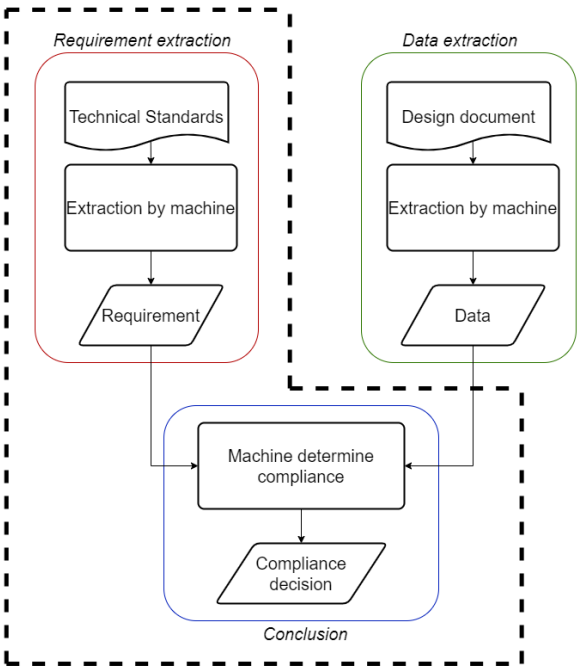

Figure 1. Illustration of automatic compliance checking. The area inside the dashed border is the scope of this research

2.2. Logic Programming Basics

In logic programming, we start with disjoint sets $\mathcal{P}$ of *predicates*, $\mathcal{F}$ of *function* symbols (including *constant* symbols), and $\mathcal{V}$ of *variables*. A *term* is recursively defined as either a variable $v \in \mathcal{V}$, or a *constant* $c \in \mathcal{F}$, or an expression of the form $f(a_1, \ldots, a_n)$ where $f \in \mathcal{F}$ is a function symbol of arity $n \geq 1$ and each a_i are terms. Constant symbols are also called *atoms* in Prolog. An *atomic formula* is of the form $p(t_1, \ldots, t_n)$ where p is a predicate of arity $n \geq 0$, and each t_i are terms. A *rule* is a statement of the following form in Equation 1

$$H \leftarrow \mathrm{B}_1, \ldots, \mathrm{B}_\mathrm{m} \quad (1)$$

$H, B_1, \ldots, B_m$ are all atomic formulas or negations of atomic formulas expressed in the form of $not\ p(t_1, \ldots, t_n)$. The rule in Equation 1 represents a logical implication and can be read as $if\ B_1, \ldots, B_m\ then\ H$.

The atomic formula H is called the *head* of the rule, while the set $\{B_1, \ldots, B_m\}$ is called the *body* of the rule. If $m = 0$, we say that the rule is a *fact*. A *logic program* is then defined as a set of such rules. In the context of such a logic program, one may pose a query, which is just a conjunctive set of atomic formulas $F_1, \ldots, F_k$.

A *list* in logic programming is represented as either the atom '[ ]' representing an empty list or a compound term with functor '.' and two arguments representing the *head* and *tail* of the list. The tail of the list is itself a list, thus, a list can be represented in Equation 2.

$$.\left(t_1, .\left(t_2, \ldots .\left(t_n, [\,]\right)\right)\right) \quad (2)$$

each t_i is a term, and n is the number of elements in a list. For readability, special notation using square brackets can be used, so (2) can be represented as $[t_1, t_2, \ldots, t_n]$. *Tuple* and *triple* are terms in the form of $(t_1, \ldots, t_n)$ where n is 2 and 3 for tuple and triple, respectively.

Negation in the logic programming language Prolog is implemented in the form of *negation as failure*. In this type of negation, $not\ p$ is evaluated by trying to prove p. If p is proven, then the negation $not\ p$ will fail. Since Prolog operates under a *closed world assumption*, whether p is proven or not relies on whether the facts related to p exist.

In Prolog, the left arrow (←) is written using a semicolon followed immediately by a dash, i.e., the characters ':-' (without the quotes). An example of a simple logic program is given in Equation 3.

```
edge(a, b).
edge(b, c).
edge(c, d).

path(A, B) :- edge(A, B).
path(A, B) :- edge(A, X), path(X, B).
```
(3)

The semantics of logic rules can be understood as a universally quantified logical implication. For example, in the rules given in equation (3), we define `path (A,B)` as a predicate that represents the existence of a path between points `A` and `B`. The rules then can be read as "path between A and B exists *if* there is an edge between A and B." It also checks for the path that passes through other nodes before reaching `B` with the recursive call in `path(A, B) :- edge(A, X), path(X, B).` This rule is then read as "path between A and B exists *if* there is an edge between A and X and there is a path between X and B,"

The conclusion of whether a path exists between two points, say `a` and `d`, is obtained by running a query as shown in Equation 4.

```
?- path(a, d).
```
(4)

in which the existence of the fact `edge(a, d)` is checked. When it is not found, the *alternative rule* is evaluated to recursively check whether a path connecting a to d through other nodes exists. For further details on the semantics of logic programs, please refer to [20].

2.3. Overview of the Proposed Method

Technical standards in our context typically consist of *clauses* that must be satisfied by a vessel or its design documents. Each clause is expressed in the form of one or more natural language sentences that convey the meaning of obligation or necessity. For the purpose of automating compliance checking, our aim is to represent such a clause as a set of logic programming rules. Meanwhile, data about a vessel or its design documents are represented as a set of logic programming facts. The compliance checking is then realized by running a query (expressing a compliance check for a particular object) against the logic programming rules (expressing the requirements from the standard and the database of facts corresponding to all objects whose compliance is to be checked).

Since there are numerous requirements in technical standards, it is generally infeasible to write logic programs to represent all those requirements. Thus, the second part of our proposed method consists of the use of an LLM to generate a logic program that approximately represents the requirements in the standards. Specifically, we employ prompt engineering over GPT-4o to obtain logic rules. The overall scheme of our proposed method is shown in Figure 2.

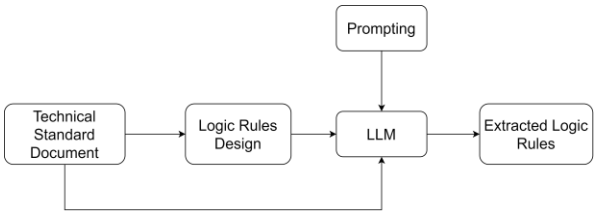

Figure 2. Overall scheme of the proposed method.

2.4. Expressing Data as Facts

One of the parts of compliance checking is a database of facts corresponding to objects whose compliance is to be checked. These facts consist of the representation of objects and their properties. In addition, they also represent the relation between objects.

First, we define ObjectID as a representation of the object. ObjectID is defined as a combination of number and initials of the object. Second, we define facts that represent the properties of objects. The category of the object is represented as a fact named hasCategory. This fact has two arguments, ObjectID and Category of the object. Third, the facts that represent relationships between objects are defined. A predicate named objectRelation is used for this purpose. It may have arity of 3 to represent S – V – O structure or arity of 2 to represent S – V structure. Last, we define a fact to represent information about objects' numerical value. The predicate named hasValue is defined with an arity of 2 and arguments which consist of ObjectID and Number. The summary of the facts is presented in Table 1.

Table 1. Facts

Object category
hasCategory(ObjectID, Category)
Object relation
objectRelation(ObjectID1, Relation, ObjectID2)
objectRelation(ObjectID, Relation)
Object value
hasValue(ObjectID, Number)

To better illustrate how data is represented as logic programming facts, we choose one clause as an example, that is "If internal combustion engines and boiler plants operating on heavy fuel, provision is to be made to ensure that internal combustion engines and boiler plants can be operated temporarily on fuel which does not need to be preheated". This clause gives requirements regarding a ship's machinery installation, specifically the combustion engine and boiler plant. We present some data related to those objects and their representation as facts in Table 2.

Table 2. Example of Facts

Data
Internal combustion engine operates on heavy fuel
Boiler plant operates on heavy fuel
Facts
hasCategory(id1_ice, internal_combustion_engine).
hasCategory(id1_bp, boiler_plant).
hasCategory(id1_hf, heavy_fuel).
objectRelation(id1_ice, operates_on, id1_hf).
objectRelation(id1_bp, operates_on, id1_hf).

2.5. Expressing Clauses as Logic Rules

Logic rules representation is produced based on the clauses. Our approach differs from the method explained in [10] in that we use a general pattern that is applicable to all clauses to minimize variation in logic rules. We observe the clauses used in technical standards to find the general pattern. Based on that consideration, we defined two components of a sentence, *prerequisite* and *main requirement*. *Prerequisite* is a requirement that has to be satisfied so that the clause is applicable to a given object. *Main requirement* is a requirement that has to be satisfied by the object so that the clause is deemed as complied. We use our earlier example to illustrate this concept in Figure 3.

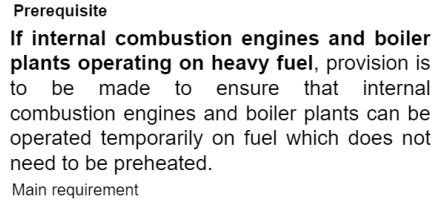

Figure 3. Illustration of prerequisite (in bold) and main requirement in a text

In that clause, the prerequisite part means that the main requirement only applies to internal combustion engines and boiler plants that satisfy the condition of "operating on heavy fuel". If there exist internal

combustion engines and boiler plants that operate on heavy fuel, the main requirement has to be satisfied so that internal combustion engines and boiler plants are deemed to comply with the said clause, i.e. they must be able to be temporarily operated on fuel which does not need to be preheated.

Inspired by the approach proposed by Lee et al. [21] and Ramanauskaitė et al. [22] we break each part even further into *atomic statements*. We defined a more general structure of this atomic statement to be a simple statement that follows S – P – O (Subject – Predicate – Object) akin to knowledge graph triples. With this definition, structures such as S – V – O (Subject – Verb – Object), S – V (Subject – Verb), or S – V – C (Subject – Verb – Complement) can be accommodated, which will be explained later.

Following the pattern in the clause, the main requirement and prerequisite may share an object in S or O position. Alternatively, they may have their separate sets of objects. Figure 4 illustrates this pattern.

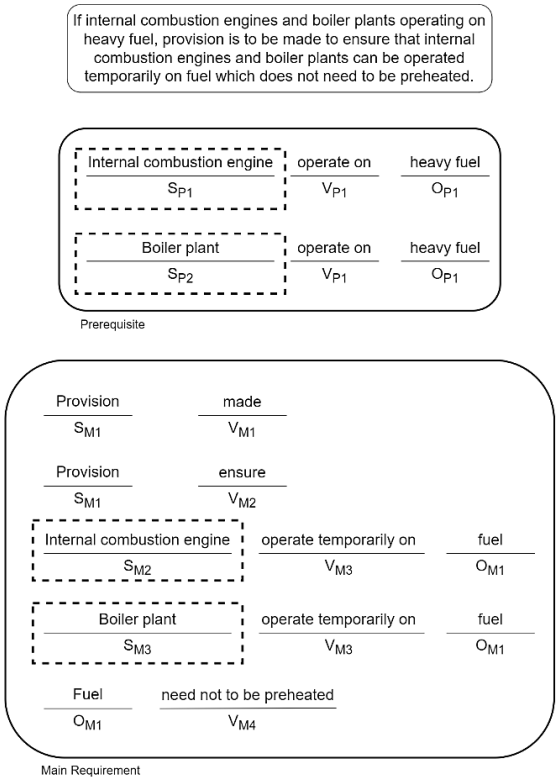

Figure 4.Illustration of atomic statements. Dashed thick border indicates objects in S position shared by main requirement and prerequisite. P index is used to indicate words belonging to prerequisite, while words with M index belongs to main requirement. The same words are assigned to the same index

Another pattern that we consider is the logical relations between objects and statements, marked by the use of connectives. There are three types of connectives used in the clauses, namely *and*, *or*, *and/or*. The explanation for each relation is given below:

And: this relation means that all the objects in the statement must exist, or all the statements must be true for the clause to be evaluated to comply.

Or: this relation conveys the meaning of exclusive or (XOR) in logics, i.e. when used to connect objects, only one object needs to exist for the clause to be evaluated as comply. Similarly, when used as a connector between statements, only one statement needs to be true. The objects and statements connected by *or* cannot be simultaneously true.

And/or: this relation between objects means that one or more objects need to exist and the statements involving the objects that exist must all evaluate to true for the overall clause to be evaluated as comply.

The following examples illustrate the relations further:

– Generator *and* main switchboard must be installed in main engine room

This clause means that both generator and main switchboard must be provided and both of them must be installed in main engine room.

– Generator must be installed in main engine room *or* particular auxiliary machinery room

This clause means that generator must be installed in either main engine room or particular auxiliary machinery room. It cannot be installed in both rooms.

– Emergency switchboard must be installed in the room containing emergency generator *and/or* emergency battery

This clause means that emergency generator, emergency battery or both of them must be provided. Whichever is provided must be installed in the same room with emergency switchboard.

After identifying the patterns in the clause, we define the logic rules to represent the clause based on the identified pattern. First, we take a look at the general structure of the clause which contains prerequisite and main requirement. We define the compliance value of a clause as true if the requirements in prerequisite and main requirement are fulfilled. We also define the compliance value to be true if the prerequisite is not fulfilled. The reason for this choice is twofold. The first one is that since there is no other conclusion in logic other than true or false, one of the two needs to be chosen to indicate that the clause is not applicable. The second is that when the prerequisite is not fulfilled, it is generally considered still complied by reviewer, thus the compliance value will be defined as true when the prerequisite is not fulfilled. Based on the aforementioned considerations, the main rule is defined in Table 3.

Table 3. Main rule

Main rule
satisfiesRegulation(RuleID) :- applicableRule(RuleID, PrereqObjects), !, satisfiesMainRequirement(RuleID, PrereqObjects). satisfiesRegulation(RuleID).

In this rule, a RuleID is assigned to each clause and satisfiesRegulation takes RuleID as an argument. If a clause with RuleID of r1 is complied then satisfiesRegulation(r1) will evaluate to true. Furthermore, satisfiesRegulation will evaluates to true when both predicates applicableRule and satisfiesMainRequirement evaluates to true. When applicableRule evaluates to false, alternative rule of satisfiesRegulation will be evaluated which will be true, mimicking the definition previously mentioned.

Next, the rules to check applicability and fulfillment of main requirement is defined. These rules have the predicates applicableRule and satisfiesMainRequirement as their head, respectively. The rules are presented in Table 4.

Table 4. Rule for applicability and fulfillment

Rule to check applicability
applicableRule(RuleID, PrereqObject) :- processPrerequisite(RuleID, PrerequisiteStatement, Acc, PrereqObjects). Rule to check main requirement fulfillment satisfiesMainRequirement(RuleID, PrereqObjects):- resolveRequirement(RuleID, MainRequirementStatement, PrereqObjects, LinkedObject).

Both applicableRule and satisfiesMainRequirement have arity of 2 which consists of the RuleID and a variable called PrereqObjects. This variable will be unified in applicableRule with objects that satisfy the requirements. The same variable is also used as argument in satisfiesMainRequirement to represent objects shared between prerequisite and main requirement. Other variables used for this representation are Acc and LinkedObject. Acc is used to collect objects in prerequisite that satisfy the requirements. After all the objects are checked, it will be unified with PrereqObjects. LinkedObject is in the form of a list of objects in prerequisite that will also be used in main requirements, hence the name. Only objects that are both present in PrereqObjects and LinkedObject will be used to check the fulfillment of main requirement. In a condition where there is no common object between main requirement and prerequisite, LinkedObject will be an empty list.

The atomic statements that formed the prerequisite and main requirement are represented in variables. They are named PrerequisiteStatement and MainRequirementStatement respectively. Those variables are List of tuples and triples as explained in Table 5.

Main requirement statements
[*(Obj1, Relation, Obj2)*, ...]
Linked objects
[LObj1, LObj2, ..., LObjn]

POobj is prerequisite object in the form of list of tuples. The list of tuples is in turn in the form of [(Fx, Cat)]. Fx is a variable, where x is a running number and Cat is an atom that indicates the category of the object.

The triple *(Obj1, Relation, Obj2)* is a representation of atomic statement in triple form. As explained before, the atomic statement follows the structure of S – P – O. Consequently, the predicate (P) is not always a verb and the object (O) may not be a noun.

Obj1 is the first object of the statement. It fills the position of Subject (S) in the S – P – O triple. Only list of tuple in the form of [(Fx, Cat)] is allowed as Obj1. If the same object is referred multiple times in the text, they will have same variable.

Relation between object in the statement fills the position of Predicate (P) in the S – P – O triple. As mentioned before, Relation is not always a verb and modals in the predicate is omitted. We define the Relation to be one of the several possible form. The first one is an atom that represents the relation between objects e.g. installedIn, equippedWith, protects. Next, the Relation can be in the form of mathematical comparison symbols e.g. more than (>), equals to (==), not equal to (<>). Last, the Relation can take the form of special relation named comp.

Table 5. Data structure (prerequisite object is in **bold**, statements are in *italic*, and objects are marked with underline)

Prerequisite statements
[(**POobj**, [*(Obj1, Relation, Obj2)*, ...]), ...]

Mathematical comparison symbols are used to represent comparison between objects and numbers. On the other hand, the special relation comp is used to represent comparison between objects. Comparison between objects is handled using a special rule called comparison. The rule is defined in Table 6.

Table 6. Comparison rule

Comparison rule
comparison(RuleID, Value1, Value2) :- Value1 ? Num ◊ Value2 Note: Num denotes a number ? denotes comparison (<, >, >=, <=, ==, <>) ◊ denotes mathematical operator (addition, multiplication, divsion)

Obj2 is the second object of the statement. This component fills the position of Object (O) in the S – P – O triple. Several types of data objects are allowed for Obj2. The first one is list of tuple in the form of [(Fx, Cat)]. Special atom named no_object is also allowed, which is used when the original statement in the clause

follows S – V structure. Lastly, numbers are also allowed to be in the position of Obj2.

LObj is the list-linked object. It is used to indicate shared objects between the prerequisite and the main requirement. Only tuple in the form of (Fx, Cat) is allowed as LObj. Objects in the linked object list will have the same variable as an object in the main requirement statement if they are of the same category.

Using the design shown in Table 3 through Table 5, we show another example of logic rules representation. We use our running example of the clause to demonstrate how a clause is represented in logic rules. We assign the RuleID of r1 for this representation. This example can be found in Table 7.

Table 7. Rule for applicability and fulfilment

Clause
If internal combustion engines and boiler plants operating on heavy fuel, provision is to be made to ensure that internal combustion engines and boiler plants can be operated temporarily on fuel which does not need to be preheated
Representation
satisfiesRegulation(r1) :- applicableRule(r1, PrereqObjects), !, satisfiesMainRequirement(r1, PrereqObjects). satisfiesRegulation(r1). applicableRule(r1, PrereqObject) :- processPrerequisite(r1, [([(P1, internal_combustion_engine)], [([(P1, internal_combustion_engine)], operates_on, [(P2, heavy_fuel)])]), ([(P1, boiler_plant)], [([(P1, boiler_plant)], operates_on, [(P2, heavy_fuel)])])], Acc, PrereqObjects). satisfiesMainRequirement(r1, PrereqObjects):- resolveRequirement(r1, [([(F1, provision)], made, no_object), ([(F1, provision)], ensure, no_object), ([(F2, internal_combustion_engine)], operates_temporarily_on, [(F2, fuel)]), ([(F3, boiler_plant)], operates_temporarily_on, [(F2, fuel)]), ([(F2, fuel)], need_not_to_be_preheated, no_object)], PrereqObjects, [(F2, internal_combustion_engine), (F3, boiler_plant)]).

After defining rules related to the prerequisite and main requirement, as well as the facts, we define the way to represent logical relations in the clause. *And* relation is represented as triples in the same List in the prerequisite or main requirement statement. *Or* relation is represented using alternative rules, so multiple applicableRule or satisfiesMainRequirement predicates are used to represent it. On the other hand, *And/Or* relation is represented as multiple (Fx, Cat) tuples in the Obj1 or Obj2 position. The examples of such representation are presented in Tables 8 through 10.

The rules to check the applicability and fulfillment of the main requirement have processPrerequisite and resolveRequirement as the head, respectively. In general, those rules involve recursively evaluating the list of triples which represent atomic statements. To find suitable objects, hasCategory facts are searched to find objects whose Category matched with the value of Cat in [(Fx, Cat)] tuple. After objects with matched category are found, next the ObjectID is unified with Fx. Moving on to the relation between objects, objectRelation facts are then searched to determine whether there is a relation between objects that matched the one stated in the triples. When comparison is involved, the value of the object is first obtained by searching through the hasValue facts and then the value is used to evaluate the predicate comparison.

For processPrerequisite, the PObj part of the tuple will be appended to the variable Acc if the list of triples is evaluated to true. It will then consequently unified with PrereqObjects. In resolveRequirement the elements of PrereqObjects will be unified with elements of LinkedObject whose category is the same and then used to evaluate the list of triples recursively.

Table 8. And relation example

Original text
Control panel and switchboard must be installed in control room
Representation
satisfiesMainRequirement(r1, PrereqObjects):- resolveRequirement(r1, [([(F1, control_panel)], installed_in, [(F2, control_room)]), ([(F1, switchboard)], installed_in, [(F2, control_room)])], PrereqObjects, LinkedObj).

Table 9. Or relation example

Original text
Generator must be installed in engine room or machinery room
Representation
satisfiesMainRequirement(r1, PrereqObjects):- resolveRequirement(r1, [([(F1, generator)], installed_in, [(F2, engine_room)]),], PrereqObjects, LinkedObj). satisfiesMainRequirement(r1, PrereqObjects):- resolveRequirement(r1, [([(F1, generator)], installed_in, [(F2, machinery_room)]),], PrereqObjects, LinkedObj).

Table 10. And/Or relation example

Original text
Generator and/or battery must be provided
Representation
satisfiesMainRequirement(r1, PrereqObjects):- resolveRequirement(r1, [([(F1, generator), (F2, battery)], provided, no_object)], PrereqObjects, LinkedObj).

2.6. Prompt Engineering to Generate Logic Rules

After designing the logic rules to represent clauses of technical standards, we define the prompt that is given to LLM to obtain such representation from the text. We take inspiration from research conducted by Carta [17],

which uses zero-shot prompting in a pipeline manner. We modify the pipeline and incorporate few-shot prompting [23] and chain-of-thought [24] to analyze the sentence structure in a clause and obtain the logic rules. For the remainder of the paper, this method will be called the *pipeline method*.

Considering the ability of LLM to generate code to solve the problem given in the prompt, we explored another prompting method to obtain the logic rules representation. In this method, we prompt the LLM to obtain logic rules directly from text. To steer it to use the logic rules we have previously defined, we use few-shot prompting and give it several examples. We call this method a *code generation* method.

The pipeline method generally consists of the analysis of sentence structure and extracting its component to obtain logic rules. The LLM is given several tasks in a pipeline to obtain logic rules from text. The tasks given to LLM can be seen in Figure 5.

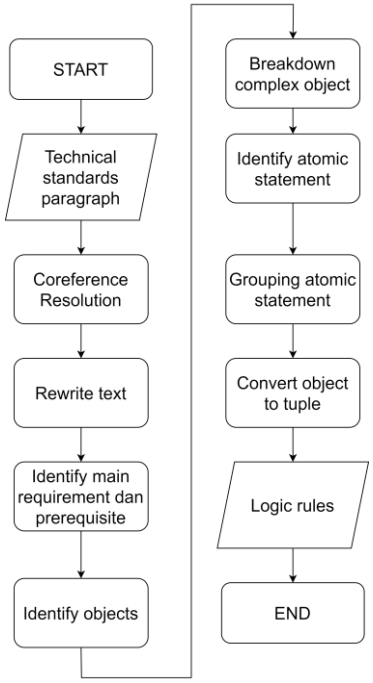

Figure 5. LLM Task in Pipeline Manner

First, the LLM is tasked to rewrite the text of the technical standard. It is done by prompting LLM to do coreference resolution to the text so that all synonyms and pronouns are changed into the object they are referring to. In addition, given the text of the technical standards and its context, LLM is also tasked to rewrite the text to incorporate the context.

Next task is identification of main requirement and prerequisite. Identification is done while considering whether the prerequisite is explicit and whether expression of permission is present. The prompt given to LLM can be summarized with the flowchart shown in Figure 6.

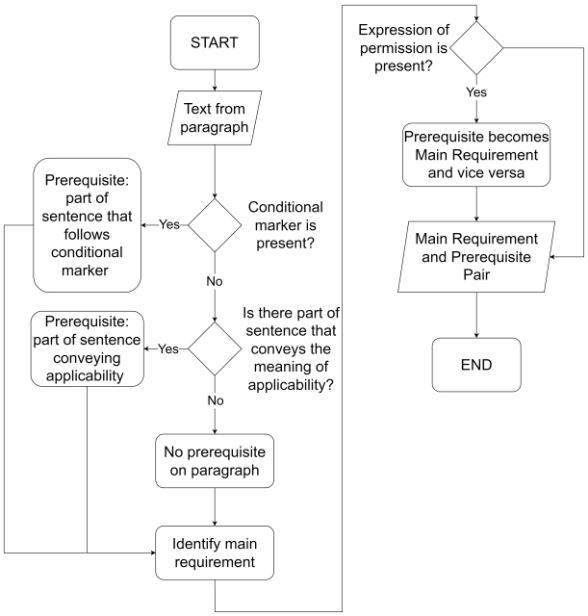

Figure 6. Flowchart of Main Requirement and Prerequisite Identification Process

After main requirement and prerequisite are identified, LLM is tasked to identify the objects of each of those parts. The objects are not necessarily in grammatical Object (O) position, but they are always a noun. Coordinating conjunction between objects is left as it is. LLM is also given prompt to break down object in the form of participial phrase or noun phrase to S – P – O triples. In addition, the object that implies the meaning of possession is also broken down to S – P – O triples. For example, the phrase "rated current of motor" or "motor's rated current" is converted into "motor – has – rated current".

Next, a prompt is given to LLM to extract atomic statements from prerequisite and main requirement. Generally, it is done by breaking down the sentences into S – P – O triples where the S and O must be singular noun and the P must conform to the singular form of S and O. Some considerations have to be taken, particularly regarding comparison. Predicates that contain comparison between object and number must be converted into mathematical comparison symbol, for example phrases like "more than" and "not less than" is converted into ">" and ">=". On the other hand, comparison between objects is converted so that the comparison uses special relation comp and then the actual comparison is separated from the text. For example, the comparison in the sentence "The protection is not to be less than 150% the rated current of the motor" is converted into "protection – comp – rated current" while the actual comparison "protection – >= – 150% * rated current" is placed separately from the rest of the text. This extraction can be summarized in the flowchart shown in Figure 7.

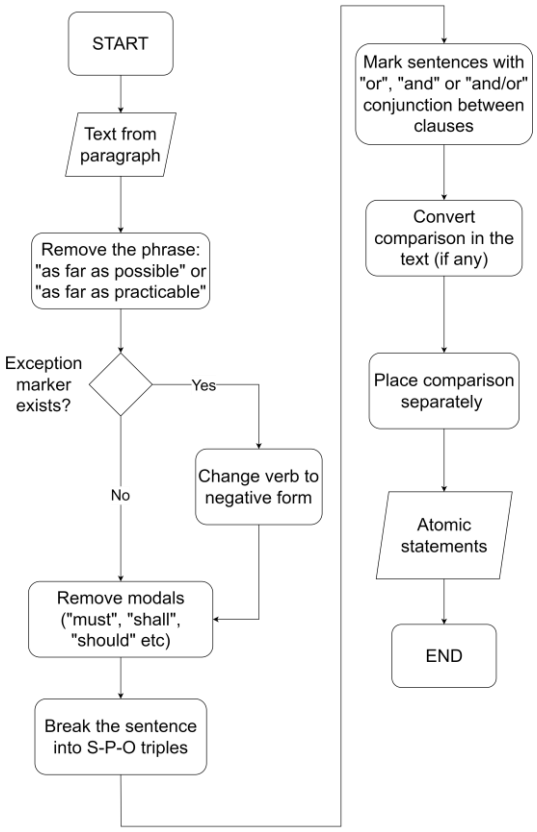

Figure 7. Flowchart of Atomic Statements Extraction Process

Another consideration is related to the logical relation in the text. To handle this logical relation, the atomic statements are first grouped based on the part of the text they originated. Indications are also given to show the logical relation applied to them. For example, the text "Electrical installation must be provided with a device to protect against residual current or a device to monitor ground fault" is extracted into the following atomic statements:

(or_1) electrical installation – provided with – device

(or_1) device – protect against – residual current

(or_2) electrical installation – provided with – device

(or_2) device – monitor – ground fault

The last part of the pipeline deals with conditioning the atomic statements so that it will be easier to convert into logic rules as designed. First, the identified atomic statements are collected in JavaScript Object Notation (JSON) format. The structure of the JSON is presented in Table 11.

The value of type is a string that indicates the logic relation in the original text i.e. "AND", "OR", or "AND/OR". The objects related to the atomic statements are marked with the key objects in the form of array of strings such as ["motor", "generator"], or ["panel"], while the atomic statements are arranged as array of strings in the form of ["object1 – relation – object2"] and marked with the key statements. The key calculation marks the comparison between object in the form of "Value1 ? Num ◊ Value2" as explained in Table 6.

Next, the objects are converted into [(Fx, Cat)] tuples. The object will be used as the category, while the variable is assigned by LLM. We define the naming of convention of the variables where Px is used as variables related to prerequisite and Fy is used as variables related to main requirement while x and y are running numbers. As previously explained, objects that are referred multiple times in the text will have the same variable.

Table 11. JSON structure

```
JSON
{
  "prerequisite": {
    "type": type,
      "pair": [
        {
          "objects": objects,
          "statements": statements
        }
      ],
    "calculation": calc
  },
  "main_requirement": {
    "type": type,
      "pair": [
        {
          "objects": objects,
          "statements": statements
        }
      ],
    "calculation": calc
  }
}
```

The completed JSON is then processed using a Python script to be converted to proper logic rules. A postprocessing process is also applied to the resulting logic rules to ensure that the rules can be executed without error. The postprocessing contains converting dashes (-) to underscores (_) and changing capital letters to lowercase.

The code generation method makes use of the LLM ability to understand and generate code [25] [26]. For this method, we utilize few-shot prompting to directly convert text to logic rules. We use few-shot prompting to make sure that the logic rules produced by LLM follow the rules and data structures we have defined previously.

We prepare several pairs of text and the resulting logic rules as examples for the LLM. The pairs are selected to present LLM a variety of clause structures and their corresponding logic rules representation. The examples contain clauses with and without prerequisites, a variety of logical relations and clauses containing comparison, whether it is between objects or between objects and number.

2.7. Data and Evaluation Scenarios

We use 30 clauses from Rules for Electrical Installations (Pt.1, Vol.IV) [27] published by BKI. The selected clause contains no table, figure, equation or definition. We assign RuleID of r1 through r30 for the clauses. Some characteristics of the clauses are presented in Table 12.

Table 12. Characteristics of clauses used

Criteria	# of clause	Clause
With prerequisite	20	r1, r2, r3, r4, r6, r9, r10, r11, r13, r14, r15, r18, r20, r21, r22, r23, r24, r25, r26, r27, r29
No prerequisite	10	r5, r7, r8, r12, r16, r17, r19, r21, r28, r30
With “and” relation	26	r2, r3, r4, r5, r6, r7, r8, r10, r11, r12, r13, r14, r15, r16, r17, r19, r20, r21, r22, r23, r24, r25, r27, r28, r29, r30
With “or” relation	5	r1, r2, r3, r10, r27
With “and/or” relation	3	r9, r10, r26
With comparison	6	r5, r7, r8, r12, r28, r30

The facts used to evaluate the logic are taken from real design documents of three (3) ships of the type of tug boat. The designs of these ships have already been judged as complying with the technical standards by the reviewer. We then obtain the facts following the previously explained format. We also use synthetic data to present facts which are then judged by reviewers as not complying and not applicable. We label the data from the three real ships as originating from ship 1 through 3, and the synthetic data is given labels ship 4 and ship 5. Some examples of the facts obtained are presented in Table 13.

The prompt and the logic rules are run on a laptop with an Intel Core i7 8 CPU @2.9GHz processor and 16GB RAM. We use GPT4o LLM, which is accessed using an application programming interface (API) provided by Openai with a temperature setting of zero (0). The code used to access the API is written in Python using the LangChain library. The logic rules are written in Prolog while the execution is done using SWI-Prolog.

Table 13. Examples of facts

Data
Motor has power rating of 4000 W
Motor is protected against short circuit
Motor is protected against overload

Table 13. Examples of facts - continued

Facts
hasCategory(id94a_mot, motor).
hasCategory(id95a_pwr, power_rating).
objectRelation(id94a_mot, has, id95a_pwr).
objectRelation(id94a_mot, protected_against, id72_shc).
objectRelation(id94a_mot, protected_against, id73_ovl).
hasValue(id95a_pwr, 4000).

Two types of evaluation are done in this research. The first is done to the design itself to evaluate how well the logic rules that we design are able to represent the technical standards clauses. The second one is done according to the logic rules obtained using LLM to evaluate how well the output of LLM is able to represent the technical standards clause.

We argue that the logic rules representation obtained using LLM cannot be evaluated using its similarity with the logic rules obtained manually. The clause of technical standards can be represented as logic rules in many ways; thus, if the logic rules are not similar, we cannot dismiss them outright as false representations. Hence, for the evaluation, we measure the *faithfulness* of the representation. We define faithful representation as one which will give the same compliance decision as the reviewer's judgement given the same facts. In both evaluations, various facts are provided. We seek judgment from reviewers on whether the given fact will produce a compliance decision of true, false or not applicable. We then compare it with the compliance decision as produced by the logic rules.

To do the first evaluation, we make the logic rules representation manually. For the second evaluation, the prompt from each method is run 5 times. The resulting logic rules are then run against the provided facts.

Since there is no metric to measure faithfulness, we approximate it by posing the evaluation as an evaluation of multi-class classification. We treat each compliance decision as a class and then define the true value as the compliance decision based on reviewer judgement, and the predicted value as the compliance decision produced by the logic rules. Afterwards, we calculate the accuracy score as well as macro-averaged precision and recall.

3. Results and Discussions

3.1 Results

As explained before, first, we make a logic rules representation of the technical standards manually. We execute it against the facts provided and record the result as shown in Table 14. This table groups the clauses in comply, not comply, and not applicable groups based on ground truth and results from executing logic rules. We find that the representations produce exactly the same compliance decision as reviewer judgement. We also present some examples of the manually obtained logic rules in Table 15.

Logic rules obtained by using LLM are executed with the same provided facts. We present a sample of results of logic rules execution in Table 16, while the overall result measured as accuracy, precision and recall score is presented in Table 17. It can be seen that logic rules obtained using the pipeline method have better accuracy, precision and recall than those obtained using the code generation method. We can say then, between the two methods utilized, logic rules obtained using the

pipeline method can represent the technical standards more faithfully.

Table 14. Result of executing the manually obtained logic rules

Ship	Compliance	Ground Truth	Result from logic rules
Ship 1	Comply	r1 - r6, r8-r30	r1 - r6, r8-r30
	Not comply	r7	r7
	N/A	-	-
Ship 2	Comply	r1 - r7, r9-r30	r1 - r7, r9-r30
	Not comply	-	-
	N/A	r8	r8
Ship 3	Comply	r1 - r7, r9-r30	r1 - r7, r9-r30
	Not comply	-	-
	N/A	r8	r8
Ship 4	Comply	r3, r4, r6, r7, r10, r13, r15, r16, r17, r18, r24, r26, r27	r3, r4, r6, r7, r10, r13, r15, r16, r17, r18, r24, r26, r27
	Not comply	r1, r2, r5, r9, r11, r12, r14, r19, r20-23, r25, r28-r30	r1, r2, r5, r9, r11, r12, r14, r19, r20-23, r25, r28-r30
	N/A	r8	r8
Ship 5	Comply	r1-r4, r6, r9, r10, r11, r13-r15, r18, r20, r22-r25, r26, r27, r29	r1-r4, r6, r9, r10, r11, r13-r15, r18, r20, r22-r25, r26, r27, r29
	Not comply	-	-
	N/A	r5, r7, r8, r12, r16, r17, r19, r21, r28, r30	r5, r7, r8, r12, r16, r17, r19, r21, r28, r30

Table 15. Examples of manually obtained logic rules

Original text
Main and emergency switchboards shall be fitted with insulation handrails or handles.
Rules

applicableRule(r27, PrereqObjects):-
  processPrerequisite(r27, [], [], PrereqObjects).
satisfiesMainRequirement(r27, PrereqObjects):-
  resolveRequirement(r27, [
  ([(F1, main_switchboard)], fitted_with, [(F2, insulation_handrail)]),
  ([(F1, emergency_switchboard)], fitted_with, [(F2, insulation_handrail)])
  ], PrereqObjects, []).
satisfiesMainRequirement(r27, PrereqObjects):-
  resolveRequirement(r27, [
  ([(F1, main_switchboard)], fitted_with, [(F2, insulation_handle)]),
  ([(F1, emergency_switchboard)], fitted_with, [(F2, insulation_handle)])
  ], PrereqObjects, []).

Original text
Motors with a power rating of more than 1kW shall be individually protected against overloads and short-circuits
Rules

applicableRule(r28, PrereqObjects):-
  processPrerequisite(r28, [
  ([(P1, motor)],[
  ([(P1, motor)], has, [(P2, power_rating)]),
  ([(P2, power_rating)], >=, 1000)
  ])
  ], [], PrereqObjects),
  checkNotEmpty(PrereqObjects).

Table 15. Examples of manually obtained logic rules - continued

Rules

satisfiesMainRequirement(r28, PrereqObjects):-
  resolveRequirement(r28, [
  ([(F1, motor)], protected_against, [(F2, overload)]),
  ([(F1, motor)], protected_against, [(F2, short_circuit)])
  ], PrereqObjects, [(F1, motor)]).

Table 16. Result of executing the logic rules obtained from LLM

Ship	Compliance	Ground Truth	Result from logic rules	
			Pipeline	Code Generation
Ship 1	Comply	r1-r6, r8-r30	r1, r3, r4, r6, r9, r11-r15, r18-r25	r4, r5, r18, r20, r22, r23, r25, r28
	Not comply	r7	r2, r5, r7, r8, r10, r16, r17, r26, r27, r30	r1, r3, r6, r8-r11, r13-r15, r26, r27
	N/A	-	r28, r29	r2, r7, r12, r16, r17, r19, r21, r24, r29, r30
Ship 2	Comply	r1-r7, r9-r30	r1, r3, r4, r6, r9, r11-r15, r18-r25	r4, r5, r18, r20, r22, r23, r25, r28
	Not comply	-	r2, r5, r7, r8, r10, r16, r17, r26, r27, r30	r1, r3, r6, r8-r11, r13-r15, r26, r27
	N/A	r8	r28, r29	r2, r7, r12, r16, r17, r19, r21, r24, r29, r30
Ship 3	Comply	r1 - r7, r9-r30	r1, r3, r4, r6, r9, r11-r15, r18-r25	r4, r5, r18, r20, r22, r23, r25, r28
	Not comply	-	r2, r5, r7, r8, r10, r16, r17, r26, r27, r30	r1, r3, r6, r8-r11, r13-r15, r26, r27
	N/A	r8	r28, r29	r2, r7, r12, r16, r17, r19, r21, r24, r29, r30

Table 16. Result of executing the logic rules obtained from LLM - continued

Ship	Compliance	Ground Truth	Result from logic rules	
			Pipeline	Code Generation
Ship 4	Comply	r3, r4, r6, r7, r10, r13, r15, r16, r17, r18, r24, r26, r27	r3, r4, r6, r13, r14, r15, r18, r24	r4, r18
	Not comply	r1, r2, r5, r9, r11, r12, r14, r19, r20-23, r25, r28-r30	r1, r2, r5, r7-r12, r16, r17, r19-r23, r26, r27	r1, r3, r5, r6, r8-r11, r13-r15, r20, r22, r23, r26-r28

Ship	Compliance	Ground Truth	Result from logic rules	
			Pipeline	Code Generation
	N/A	r8	r25, r28-r30	r2, r7, r12, r16, r17, r19, r21, r24, r25, r29, r30
Ship 5	Comply	r1-r4, r6, r9, r10, r11, r13-r15, r18, r20, r22-r25, r26, r27, r29	r1, r3, r4, r6, r9, r11, r13, r14, r15, r18, r20, r22-r25	r4, r5, r18, r20, r22, r23, r25
	Not comply	-	r2, r5, r10, r16, r17, r19, r26, r27	r1, r3, r6, r9-r11, r13-r15, r26, r27
	N/A	r5, r7, r8, r12, r16, r17, r19, r21, r28, r30	r7, r8, r12, r21, r28-r30	r2, r7, r8, r12, r16, r17, r19, r21, r24, r28-r30

Table 17. Evaluation of logic rules obtained using LLM

Method	Accuracy	Precision	Recall
Pipeline	0.57	0.49	0.62
Code Generation	0.33	0.43	0.5

3.2 Discussions

Evaluation of logic rules obtained manually shows that the logic rules designed in this research have already been able to represent the 30 clauses used. Since the pattern used as the basis of logic rules design is general enough, we argue that this design can be applied to other clauses in the technical standards. The exception is on the clause that contains tables, figures and equations. We will try to address these limitations in future works.

To analyze the performance of the representation obtained using LLM, we compare the logic rules outputted by LLM and the logic rules made manually. As explained before, we avoid evaluating the similarity of the logic rules since different logic rules do not necessarily mean that the rules are incorrect. Instead, we define several fault categories that may be present in the logic rules obtained by LLM.

All the occurrence of the fault is then presented in the bar chart in Figure 8. For each of the 5 runs of each method, we count all the occurrences of the following fault. The first fault is an incorrect prerequisite that is defined as a condition in which LLM produce a prerequisite where there is none or vice versa. The second fault, incorrect object representation, is defined as a condition where objects are not broken down into simpler objects or combined with another object. Incorrect number of statements is the third fault where LLM leave out some atomic statements or even makes some that do not exist in the original text. Fault number four, incorrect logical relation, is when the representation of the logical relation does not match what is written in the text, e.g. making an or-relation representation when the relation in the text is an and-relation and so on. Next, incorrect comparison is the condition where the comparison in the text is not represented using the format given. Last, mis-ordered relation is the condition where the order of object in a relation is incorrect. This kind fault happened, for example, when the text "rated power of motor" is represented as "rated power – has – motor."

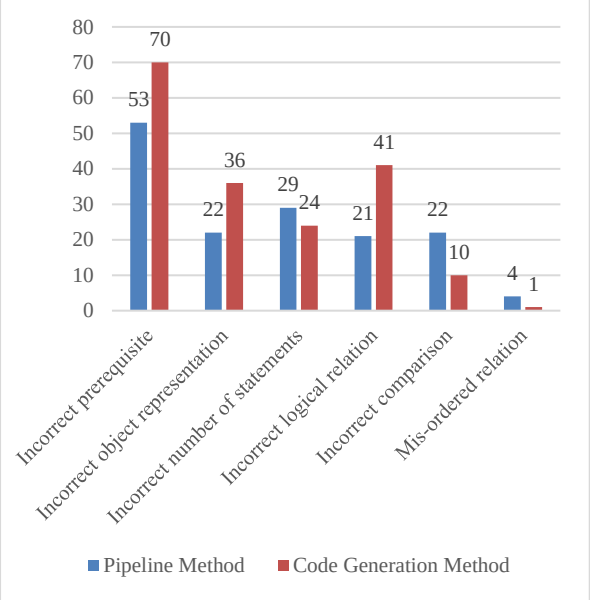

Figure 8. Flowchart of Atomic Statements Extraction Process

Examples of the occurrence of errors in the rules are given in Table 18. In that table, we present the original text of the clause, the representation that is obtained manually, and the representation generated from LLM (one for each method). We can see that the rules generated using the pipeline method cannot extract all atomic statements, hence the occurrence of the "incorrect number of statements" fault. On the other hand, the one generated from the code generation method has an object that is a combination of two conditions, estuary trading and navigation close to ports, hence the occurrence of the "incorrect representation of object" fault.

We also note the occurrence of faults in each clause. From that data, we found that there are several clauses represented perfectly. i.e. possess zero occurrence of error. We present the summary of such clauses in Table 19.

It can be seen in Table 14 that in the pipeline method, there are 7 clauses without the occurrence of error. On the other hand, in the code generation method, there are 3 clauses without the occurrence of error. This supports the results in Table 17, where logic rules produced by the pipeline method have better accuracy, precision and recall scores.

We made some observations regarding the error that occurred in the logic rules. The first one is related to the most frequently occurring error, that is, Fault 1 or incorrect prerequisite. From Figure 8, we can see that

this type of error occurred 53 times in logic rules produced by the pipeline method and 70 times in the ones produced by code generation. We argue that this error is the most prevalent because many of the prerequisites in the clause are implicit. The implicit nature of the prerequisites makes them indistinguishable from other parts of the clause. Based on this fact, the prompt given to LLM tends to rely on LLM's internal knowledge to identify the prerequisite. This leads to many occurrences of this category of error, and in turn, also affects the overall precision and recall score.

Table 18. Comparison of logic rules

Original text
The power demand has to be determined for the following operating conditions: — navigation at sea — estuary trading and navigation close to port — emergency power supply
Rules constructed manually
applicableRule(r14, PrereqObjects):- processPrerequisite(r14, [], [], PrereqObjects). satisfiesMainRequirement(r14, PrereqObjects):- resolveRequirement(r14, [([(F1, power_demand)], determined_for, [(F2, navigation_at_sea)]), ([(F1, power_demand)], determined_for, [(F3, estuary_trading)]), ([(F1, power_demand)], determined_for, [(F2, navigation_close_to_port)]), ([(F1, power_demand)], determined_for, [(F2, emergency_power_supply)])], PrereqObjects, []).
Rules generated by pipeline method
applicableRule(r14, PrereqObjects):- processPrerequisite(r14, [], [], PrereqObjects). satisfiesMainRequirement(r14, PrereqObjects):- resolveRequirement(r14, [([(F1, power_demand)], has_to_be, [(F2, determined)])], PrereqObjects, []).
Rules generated by code generation method
applicableRule(r14, PrereqObjects):- processPrerequisite(r14, [], [], PrereqObjects). satisfiesMainRequirement(r14, PrereqObjects):- resolveRequirement(r14, [([(F1, power_demand)], determined_for, [(F2, operating_condition)]), ([(F2, operating_condition)], includes, [(F3, navigation_at_sea)]), ([(F2, operating_condition)], includes, [(F4, estuary_trading_navigation_close_to_port)]), ([(F2, operating_condition)], includes, [(F5, emergency_power_supply)])], PrereqObjects, []).

Table 19. Evaluation of logic rules obtained using LLM

Method	# of rules with zero occurrence of error	Clauses
Pipeline	7	r1, r9, r11, r12, r18, r20, r22
Code Generation	3	r5, r22, r28

Another significant observation is the occurrence of the "incorrect logical relation" fault. This type of fault occurs more frequently, twice the occurrence in the logic rules obtained using code generation method. This fact indicates that pipeline method is more capable of producing representation of logical relation than code generation method. We argue that this happens because the prompt to obtain logical relation representation in the pipeline method is more detailed. It may result in more data that is seen by LLM and thus making it easier to predict logical relation representation.

We note interesting facts regarding the occurrence of incorrect comparison faults. Contrary to other types of faults, this type occurs less frequently in the logic rules obtained from the code generation method. We argue that this happens because of the training data of GPT itself, which includes code written in various programming languages. When presented with a task related to programming, GPT will use patterns in the programming languages it has learned. Those programming languages possess similarity with Prolog in their comparison-related syntax. Thus, posing the task of obtaining logic rules representation as a programming task will produce better results when comparison is involved.

4. Conclusions

This research presents the design of logic rules that can be used to represent technical standards as an effort to realize automatic compliance checking. Two prompting methods that can be used to obtain logic rules from text using LLM are also presented. The evaluation suggests that the design has already been able to represent 30 clauses of technical standards and is potentially able to represent other clauses as well. Evaluation done to the logic rules obtained using LLM suggests that the pipeline method can extract a more faithful representation. Future works should explore the method of extracting facts from design documents to realize an end-to-end system of automatic compliance checking. It should also explore the effect of various prompting techniques and LLMs in improving the faithfulness of the representation. Last, future works should explore representation that can accommodate tables, equations and images.

References

[1] "Rules for Classification: Ships - Part 1 General Regulations - Chapter 1 General Regulations," Jul. 2023

[2] L. Jiang, J. Shi, and C. Wang, "Multi-ontology fusion and rule development to facilitate automated code compliance checking using BIM and rule-based reasoning," *Advanced Engineering Informatics*, vol. 51, Jan. 2022, doi: 10.1016/j.aei.2021.101449.

[3] W3C SPARQL Working Group, "SPARQL 1.1 Overview." Accessed: Jun. 08, 2024. [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

[4] P. Patlakas, I. Christovasilis, L. Riparbelli, F. K. Cheung, and E. Vakaj, "Semantic web-based automated compliance checking with integration of Finite Element analysis,"

Advanced Engineering Informatics, vol. 61, Aug. 2024, doi: 10.1016/j.aei.2024.102448.

[5] X. Zhu, H. Li, and T. Su, "Autonomous complex knowledge mining and graph representation through natural language processing and transfer learning," *Autom Constr*, vol. 155, Nov. 2023, doi: 10.1016/j.autcon.2023.105074.

[6] M. Yang *et al.*, "Semi-automatic representation of design code based on knowledge graph for automated compliance checking," *Comput Ind*, vol. 150, Sep. 2023, doi: 10.1016/j.compind.2023.103945.

[7] I. Fitkau and T. Hartmann, "An ontology-based approach of automatic compliance checking for structural fire safety requirements," *Advanced Engineering Informatics*, vol. 59, Jan. 2024, doi: 10.1016/j.aei.2023.102314.

[8] D. Guo, E. Onstein, and A. D. La Rosa, "A Semantic Approach for Automated Rule Compliance Checking in Construction Industry," *IEEE Access*, vol. 9, pp. 129648–129660, 2021, doi: 10.1109/ACCESS.2021.3108226.

[9] X. Zhao, L. Huang, Z. Sun, X. Fan, and M. Zhang, "Compliance Checking on Topological Spatial Relationships of Building Elements Based on Building Information Models and Ontology," *Sustainability (Switzerland)*, vol. 15, no. 14, Jul. 2023, doi: 10.3390/su151410901.

[10] X. Xue and J. Zhang, "Regulatory information transformation ruleset expansion to support automated building code compliance checking," *Autom Constr*, vol. 138, Jun. 2022, doi: 10.1016/j.autcon.2022.104230.

[11] Q. Ren *et al.*, "Automatic quality compliance checking in concrete dam construction: Integrating rule syntax parsing and semantic distance," *Advanced Engineering Informatics*, vol. 60, Apr. 2024, doi: 10.1016/j.aei.2024.102409.

[12] S. Liu, B. Zhao, R. Guo, G. Meng, F. Zhang, and M. Zhang, "Have you been properly notified? automatic compliance analysis of privacy policy text with GDPR article 13," in *The Web Conference 2021 - Proceedings of the World Wide Web Conference, WWW 2021*, Association for Computing Machinery, Inc, Apr. 2021, pp. 2154–2164. doi: 10.1145/3442381.3450022.

[13] J. Devlin, M.-W. Chang, K. Lee, K. T. Google, and A. I. Language, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of NAACL-HLT 2019*, 2019, pp. 4171–4186. doi: 10.18653/v1/N19-1423.

[14] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training," 2018, Accessed: Jan. 06, 2025. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[15] H. Touvron *et al.*, "LLaMA: Open and Efficient Foundation Language Models," 2023, doi: https://doi.org/10.48550/arXiv.2302.13971.

[16] A. Chowdhery *et al.*, "PaLM: Scaling Language Modeling with Pathways," Apr. 2022, doi: https://doi.org/10.48550/arXiv.2204.02311.

[17] S. Carta, A. Giuliani, L. Piano, A. S. Podda, L. Pompianu, and S. G. Tiddia, "Iterative Zero-Shot LLM Prompting for Knowledge Graph Construction," *ArXiv*, Jul. 2023, doi: https://doi.org/10.48550/arXiv.2307.01128.

[18] J. H. Caufield *et al.*, "Structured prompt interrogation and recursive extraction of semantics (SPIRES): A method for populating knowledge bases using zero-shot learning," *Bioinformatics*, Feb. 2024, doi: 10.1093/bioinformatics/btae104.

[19] Z. Bi *et al.*, "CodeKGC: Code Language Model for Generative Knowledge Graph Construction," *ACM Transactions on Asian and Low-Resource Language Information Processing*, Feb. 2024, doi: 10.1145/3641850.

[20] I. Bratko, *Prolog Programming for Artificial Intelligence*, 4th ed. Harlow: Addison Wesley, 2012.

[21] J. K. Lee, K. Cho, H. Choi, S. Choi, S. Kim, and S. H. Cha, "High-level implementable methods for automated building code compliance checking," *Developments in the Built Environment*, vol. 15, Oct. 2023, doi: 10.1016/j.dibe.2023.100174.

[22] S. Ramanauskaitė, A. Shein, A. Čenys, and J. Rastenis, "Security Ontology Structure for Formalization of Security Document Knowledge," *Electronics (Switzerland)*, vol. 11, no. 7, Apr. 2022, doi: 10.3390/electronics11071103.

[23] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," in *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[24] J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2022.

[25] L. Zhong, Z. Wang, and J. Shang, "Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step," *Findings of the Association for Computational Linguistics: ACL 2024*, Aug. 2024, doi: 10.18653/v1/2024.findings-acl.49.

[26] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation," Dec. 2023, doi: https://doi.org/10.48550/arXiv.2312.13010.

[27] Biro Klasifikasi Indonesia, "Rules for Electrical Installations Consolidated Edition," Part 1, Vol. IV, 2024 [Online]. Available: www.bki.co.id

