Graph-Based Rule Representation for Automated Design Checking

Su Zhang1, Ke-Yin Chen1, Jia-Rui Lin1,2, Peng Pan1,2

1. Department of Civil Engineering, Tsinghua University, Beijing, 100084, China

2. Key Laboratory of Digital Construction and Digital Twin, Ministry of Housing and Urban-Rural Development, Beijing, 100084, China

*Corresponding author, E-mail: lin611@tsinghua.edu.cn (J.R. Lin), panpeng@tsinghua.edu.cn (P. Pan)

Abstract: Automated rule checking aims to automate design checking via computer-processible rules and has been extensively studied for years. Existing methods for rule interpretation are often limited in their ability to represent implicit complex computational logic, and the integration between complex logic representation methods and rule interpretation outcomes remains insufficient. In this paper, we propose a graph-based method for representing knowledge rules for automated design checking. The graph representing procedure consists of two main steps. Firstly, semantic mapping is employed, whereby the rules are represented as a semantic combination of three corresponding graph branches. Secondly, atomic function mapping is utilized, which adds atomic function nodes to the graph from the level of computable logic to form a complete graph representation, thereby forming a complete graph representation. We then provide a method for automatically generating programming code from the graphs, which, combined with the development of atomic functions, can be used for checking. This approach demonstrates considerable potential in terms of its interpretability and comprehensibility, and it provides novel ideas for the development of fully automated rule checking systems.

Keywords: Automated rule checking; Graphic representation; Atomic function; Code generation.

1. Introduction

The entire lifecycle of a building, encompassing design, construction, operation, and maintenance, must adhere to various codes, regulations, and standards to ensure safety, sustainability, and comfort[1][2]. Traditional building design reviews are primarily conducted through manual verification, which can easily lead to ambiguities during the process. Additionally, manual reviews are often inefficient and prone to errors[3][4][5].To improve the efficiency and reliability of rule reviews, Automated Rule Checking (ARC) has been extensively studied over the past few decades. ARC is a practical tool that can automatically interpret building codes and use them to check models or designs[6], typically involving four stages: rule interpretation, building model preparation, rule extraction, and reporting checking results[6]. In architectural and structural engineering, national and industry standards are usually expressed in natural language[8]. The rule interpretation stage is intended to provide interpretations of these statements that will enable the computer to understand them and facilitate the automatic recognition and checking of design models or drawing entities[7]. This stage is recognized as the most critical and complex within ARC[9]. The complexity arises from both the semantic expression of the regulations and the practical challenges of compliance checking. Semantically, the expression of building regulations can be quite complex. A single rule may incorporate multiple attributes, comparative relations, and quantitative values[10][11]. Furthermore, from an implementation perspective, acquiring the necessary data for checking is highly challenging. While some requirements target explicit data that can be directly retrieved from models, others rely on implicit domain knowledge, necessitating complex logical computations to derive the required attributes. It is also necessary to interpret such complex logic, including quantitative, topological, and geometric relations[12][13]. At the same time, the development process of the ARC system requires the participation of domain experts at various stages, such as architects, programmers, and model developers. These place demands on the comprehensibility, flexibility, and applicability of the rule interpretation results[14].

Several automated review tools have already been implemented in the industry. Notable examples include the CORENET electronic drawing checking module introduced in Singapore in 2002[7][6], the Solibri offered by a Finnish software

company[15]. However, the automated review process relies on hard-coded methods, making the review procedure opaque, inflexible, and costly. These methods focus heavily on automating the review process but do not address integration with users and are commonly referred to as "black box" approaches[16][17].

This necessitates more automated and adaptable techniques for the interpretation of rules. The mainstream approaches now include semi-automatic rule interpretation methods and Natural Language Processing (NLP)-based automatic rule interpretation methods. Some of the typical semi-automated methods include labelling-based methods[18][19][20], direct ontology-based construction methods[21][22][23], and conceptual graph-based representation methods[14]. While these methods significantly enhance the flexibility and transparency of the rule interpretation process, they are only capable of processing text at a coarse granularity level and continue to necessitate human involvement for tasks such as semantic tagging[17][24]. The NLP methods use deep learning to extract and classify text that supports automated rule interpretation[25][26], and further process text into specific logical forms that can be utilized for ARC[11][27][28][29]. The rapid development of Large Language Models (LLMs) has significantly reduced the cost of fully automatic rule interpretation. These models, through extensive pre-training on large and diverse corpora, have demonstrated strong capabilities in natural language understanding and generation. They show great potential and advantages in the task of extracting structured information from regulatory texts in automatic BIM compliance checks[30][31][32]. However, most NLP-based methods are only able to achieve model review for some simple statements, mainly first-order logic and struggle to describe the clauses with implicit properties that demand complex computation logics[24][33][34]. Since they are limited by the explicit terms that can be recognized in the rule language, they are relatively weak in the face of implicit complex logic[8]. For example, the cause "Adjacent nursing units in hospital designs shall be separated by fire partition walls with a minimum fire resistance rating of 2.00 hours (from Chinese building code) " involves implicit topology and geometry information that should be derived independently. The expression of complex computational logic needs further exploration.

Some methods have been proposed specifically for representing complex computable logic, mainly: Domain-Specific Languages (DSLs) and high-level functional databases. The first approach involves creating a domain-specific

programming language for built environment specification, defining computational modules within Building Information Modeling (BIM) models. These languages would define the computational modules that are present in the Building Information Modeling (BIM) model. Examples of such languages include BERA[35], QL4BIM[36], BimSPARQL[37]. The latter approach aims to extract computational logic units with commonality in the review process, write special functions to form a function library and realize the expression of complex logic by calling these public functions[38]. However, the rule interpretation process still relies on a lot of expert work to pre-define the DSL or to select suitable atomic functions[8][7]. Furthermore, the DSLs cannot be extended without changing the syntax, and the atomic function cannot be easily extended to a wide range of rule requirements from different sub-domains and different countries[14][39].

In general, seldom do the above methods simultaneously ensure good interpretability, comprehensibility and automaticity, i.e., good representation of semantic structures and computable logic, easy to comprehend, and at the same time capable of automation. To address the above problems, this work proposes a framework for representing rules based on knowledge graphs for rule checking. Atomic functions are applied to facilitate the simplicity and interpretability of knowledge graphs. This approach is expected to combine some automated methods to automate the process, such as automated NLP methods, or automated function identification methods[8][38].

The remainder of this paper is as follows: Section 2 reviews the related studies and highlights the potential research gaps. Section 3 illustrates the methodology for constructing the knowledge graph base. The code generation methodology is also included in this section. Section 0 gives some typical examples along with small-scale feasibility verification. Finally, Section 5 and Section 6 offer the discussion and conclusion of this research.

2. Overview of related work

2.1. Rule interpretation methods

The earliest known instance of rule interpretation can be traced as far back as 1966, when Fenves employed the use of decision tables to represent review logic in building

codes as a more user-friendly framework[38]. Subsequently, SASE systems were developed using decision trees and predicate-based logical structures[6]. These early systems represented the initial forays into the processing of the semantic structure of rules, yet they necessitated a considerable degree of human involvement.

To better represent the semantic structure of rules, Lau et.al proposed a framework for extracting similarities between clauses and using eXtensible Markup Language (XML) to represent features[40]. The RASE method divides the content of clauses into four parts: Requirement, Applicability, Selection and Exception[17]. These texts could be further converted into Semantic Web Rule Language (SWRL) format for automatic review of industry foundation classes (IFC) format model[19][20]. Another type of approach is using ontological representation for rule extraction. The typical approach is to construct domain ontologies using semantic web technologies, which serve to define the main concepts and relationships involved[21][22]. Bouzidi et al. then employed a semantic web approach of Resource Description Framework (RDF) graphs to translate and store model data[22].

With the rapid evolution of NLP in recent years, effective techniques for automatic text parsing are undergoing maturation[7]. Studies have demonstrated that domain-specific texts are more conducive to automated NLP than general ontologies of general knowledge and many NLP studies combine approaches based on rules and ontology[10][41]. Zhang and El-Gohary put forth an automated rule-interpretation approach that is capable of transforming extracted information instances into logical clauses through regular expression-based mapping rules and conflict resolution rules[3][10][26]. Xu and Cai proposed an ontology-based and rule-based information extraction method for automatically interpreting utility provisions into deontic logic (DL) clauses[27]. Zhou et al. proposed a semantic annotation method, which can automatically convert input tokens into rule checking trees[27]. Based on this, Zheng et al. proposed an unsupervised learning-based semantic alignment method and a knowledge-based conflict resolution method to enhance the precision[23]. However, NLP here appears to be lacking when confronted with natural language that contains a large amount of implicit domain knowledge[14]. It still requires specific modules to process this domain knowledge and perform complex logical computations[42][43].

Researchers are also concerned with the formalization of rules and logic

representation to enhance their usability for communication purposes. Preidel et al. propose the Visual Code Checking Language, using object nodes for data types, operation nodes for algorithms, and directed edges with interfaces to represent inputs or outputs[43]. Solihin et al. use conceptual graphs to represent rules, which allow them to be decomposed into atomic rules, removing the ambiguities that often plague architectural specifications[15]. They likewise used the DSL to validate the reliability of the representation[43]. Several visual programming language approaches have also been developed, such as BERA, where code logic is displayed through connected graphs during the programming process[44]. However, for these graphic-based methods, complex graphical relationships are required to represent complex implicit, and the entire implementation process requires a certain level of comprehension cost. Some parts of the representation often rely on manuals and do not guarantee complete automation.

2.2. Computational logic representation methods

In the stage of rule interpretation, building codes frequently contain implicit domain knowledge, including mathematical, topological, and geometric operations[7][8][39]. This demands that computers execute complex calculating logic based on model data for effective review.

DSLs are designed to develop programming languages, enhancing comprehension for non-specialists while articulating complex computational logic[7]. Examples such as BERA, QL4BIM and BimSPARQL. The BERA language offers both static and dynamic forms, which are used to obtain static data and dynamic geometric attributes of BIM models[35]. The QL4BIM language includes functions for object and geometric attribute computation and has ventured into visual programming language approaches[35]. BimSPARQL is focused on query capabilities, facilitating the examination of the three-dimensional geometric data of models[36]. However, these approaches lack applicability and the language itself cannot be extended without modifying the syntax. Furthermore, the execution environment of the language constrains its capacity to accommodate complex logic.

Databases of high-level functions express complex computational logic by designing various functions suited for BIM inspections. These databases encompass a range of advanced logical functions to examine geometric, topological, and other

relationships. Uhm et al. analyzed Korean building codes, categorized nouns and then created specialized functions for complex logic[11]. For Chinese building codes, Zheng et al. have tried to define public computational units as atomic functions, including existence checking, quantity checking, and geometric checking[8]. However, using high-level function databases usually demand a strong command of the tools. And it sometimes has limitations across different domains or national standards. Both DSLs and function databases remain heavily reliant on manual effort, which is time-consuming and hinders true automation. This dependency exists because accurately mapping regulatory clauses to predefined DSLs or corresponding functions requires a level of expertise possessed only by domain specialists and has high learning costs. Thus, there are still certain difficulties in rule interpretation[11][45].

2.3. Research gaps

Although lots of efforts have been made to interpret rules and to independently represent complex logic, there are still several key limitations that persist. The first concerns interpretability. Present rule interpretation methods are usually insufficient for interpreting implicit complex logic. The existing complex logic representation approaches now have not been effectively integrated with semantic interpretations to form a complete interpretation of clauses. The second concerns comprehensibility. Constrained by their format of presentation, current rule interpretation methods lacked intelligibility among the actors engaged in the whole development process of rule checking systems. The existing graphics-based methods still suffer from other shortcomings and could be further improved. The third concerns automaticity. Several existing methods still rely on experts to interpret rules or represent complex logic. There is a lack of automated frameworks or ideas.

To address the above problems, this paper proposes a method for representing textual rules by knowledge graphs for structural review. Atomic functions are used to represent computational logic. We further provide a method to automatically generate executable Python code for BIM model reviews, based on each graph structure. The integration of knowledge graphs and atomic functions realizes the combination of interpretation at the level of semantic and computational logic, thereby ensuring interpretability. The graph structure of the interpreting result ensures overall expressiveness, while the use of atomic functions reduces the graphical expression of

complex logic and makes the entire graph more concise. By combining some automated rule interpretation methods in advance and using some techniques in atomic function identification, this framework allows for a degree of automation.

3. Methodology

As previously mentioned, existing rule representations encounter challenges in simultaneously achieving high levels of interpretability, expressiveness and automaticity. To address this issue, this paper proposes a knowledge graph-based rule representation method to enhance these aspects and to provide a relatively complete framework for integrating computational logic representation with semantic interpretation and maintaining comprehensibility. The whole work could be divided into three stages: the graph representation of rules, the generation of executable code for model reviews, and the validation of the proposed methods. The workflow of this study is illustrated in Figure 1.

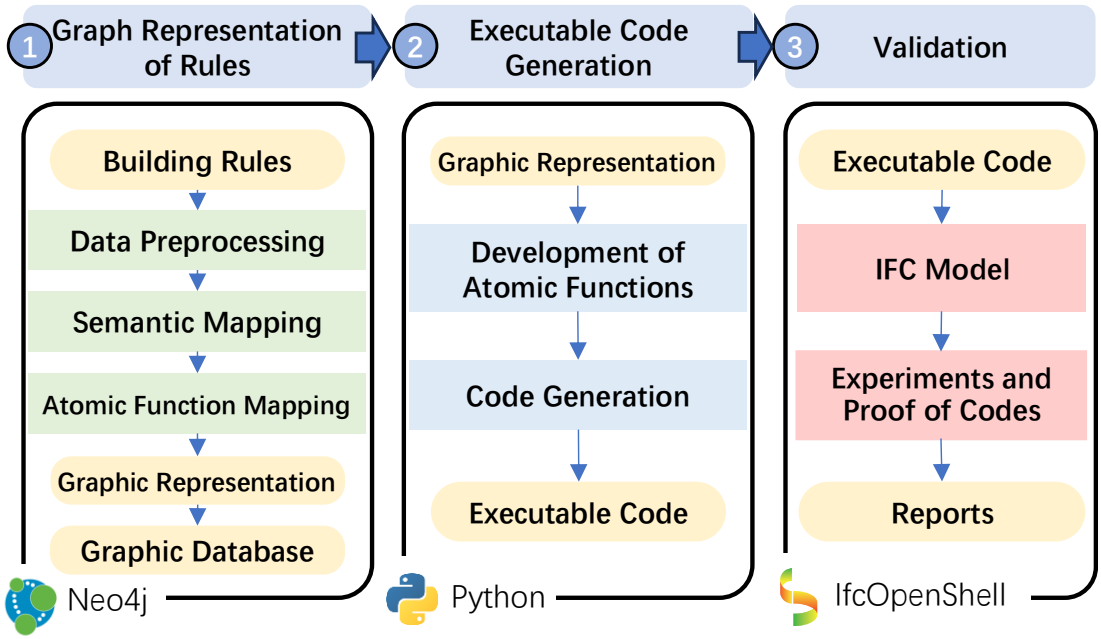


Figure 1 The workflow of this study

The graph-based rule representing methods introduce an integration structural format for interpreting clauses at the semantic level and computational logic level. It could begin with the semantic labeling results from NLP[27], and then convert the text

form clauses into graphic form. We use atomic functions to uniformly represent computational logic, thereby effectively addressing complex computational logic while concomitantly reducing the complexity of the graph. The process involves three main steps: First, collecting and preprocessing regulatory texts by segmenting, converting, and filtering them for subsequent analysis (Section 3.1.1). Second, performing semantic mapping on the preprocessed clauses, where we map a textual clause into three parts, core concepts, filters, and requirements. Each part is then mapped into more detailed components that are subsequently represented in the form of graphs as the results of semantic-level interpretation (Section 3.1.2). Thirdly, to combine computational logic, we further matched each semantic part with an atomic function as a computable method for this part in the checking process, resulting in a complete graphic-based representation of clauses. The results of the rule-by-rule representation are stored in the graph database through Neo4j (Section 3.1.3).

To further facilitate automated checking based on the graphical format of clauses, we developed a method for generating executable Python code from the clause graph representation, applicable to reviewing IFC format BIM files. We defined suitable atomic functions aligning with the knowledge graph, serving as the computational logic (Section 3.2.1). We then provided a logic framework for generating executable code based on the constructed knowledge graph (Section 3.2.2).

We then offered several typical clauses as examples to demonstrate how our method can be applied to represent these clauses as knowledge graphs and generate executable Python code (Section 0). These examples highlight the potential of our method in handling complex logic effectively. To illustrate the effectiveness of the proposed method and the correctness of the generated executable code, we employ our method for some selected clauses. A graph database is constructed, and the corresponding Python code is then generated. The approach is verified with a provided BIM model (Section 4.2). We summarize our work and make some outlooks (Section 5, Section 6).

3.1. Graph representation of rules

3.1.1. Data acquisition and preprocessing

Fire protection requirements for building structures are crucial considerations in

architectural design. This work uses the "Chinese Code for Fire Protection Design of Buildings (GB 50016-2014) "[46] to analyze semantic modeling methods for regulatory texts. The selected clauses from these chapters contain complex requirements, such as fire resistance ratings, fire separation distances, evacuation distances, and floor plans, which are suitable for semantic analysis.

The preprocessing of rule texts includes sentence segmentation, conversion of tabular clauses into textual clauses, and filtering out non-computable clauses[8][11]. The sentence segmentation step decomposes long, multi-requirement clauses into short, atomic units, each containing a single design requirement. Each resulting clause must include all necessary elements for automated checking: the object to be checked, the applicable constraints, and the specific requirement. For tabular clauses, they are first transformed into textual format expressed in natural language. Typically, the text preceding a table often defines the unified or similar objects to be checked, while the table contains specific requirements under different conditions. They are combined to form a complete, structurally sound sentence. Non-computable clauses, which require additional criteria for computers to assess as either ‘satisfied’ or ‘failed’[38][11], are filtered out. These clauses can be classified into three types. The first type is definitional clauses, which primarily introduce conceptual definitions or categorical descriptions without specifying measurable requirements. The second type consists of qualitative clauses that rely on subjective judgment due to the use of vague or non-quantifiable language, necessitating expert evaluation. The third type includes clauses that make external references to other standards or documents without providing self-contained criteria.

3.1.2. Semantic mapping and graph representation

To provide a more comprehensive representation of the semantic structure of the clauses, a process called semantic mapping was performed. Specifically, the content of each textual clause is mapped into three semantic parts based on its function: core concepts, constraint filters, and requirements conditions. To facilitate the comprehension of the interpreting results by the various participants in the ARC development process, the semantic structures are further represented in the form of graphs. The principle of mapping can be outlined as follows, and Table 2 contains some examples of such mapping.

(1) Core Concepts. The core concept corresponds to the main instance object in a textual clause, serving as the explicit subject of the clause's review and usually appearing as a noun. Its semantic function is to act as the primary focus of the clause's statement and is central to the review process. The object in each core concept is represented as an independent node. We will add a target value for each core concept. The relationship is defined as the "core concept" between the object node and the target value node, indicating that this subgraph corresponds to a core concept identified in the semantic model, as shown in Figure 2.

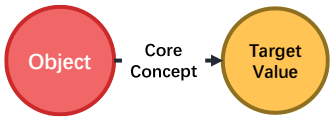


Figure 2 Core concept subgraph

(2) For Constraint Filters. Constraints filters correspond to the limitations placed on the core concept. The semantic function of the filters is to narrow down the scope of the core concept within the clause. A Constraint filter is mapped into four components: the filter object, filter attribute, comparison operator, and target value. The core logic of a constraint filter is to determine whether the relationship between the filter object's attribute and the target value satisfies the conditions defined by the comparison operator, thereby filtering the core concept. The relationship is defined as the "Constraint Filter" between the object node and the attribute node. The represented subgraph is shown in Figure 3.

Figure 3 Constraint Filter subgraph

(3) For Requirement Conditions. Requirements establish the core requirements for reviewing a textual clause. The primary purpose of the review is to identify core concepts in the model that do not meet these requirements. A requirement condition is also mapped into four components: requirement object, requirement attribute, comparison operator, and target value. The core logic of a requirement involves assessing whether the requirement attribute of the requirement object meets the criteria specified by the comparison operator, thereby serving as the basis for determining the

compliance of the clause. The overall structure of the subgraph is shown in Figure 4.

Figure 4 Requirement condition subgraph

To enable ARC, a critical step is to establish a robust mapping between target values and the corresponding entities in the IFC model. This mapping is addressed in two primary scenarios. The first scenario involves a direct mapping, where the target value can be explicitly described by a defined IFC object or its properties. For core concepts, the target is set to a specific IFC class if it fully encapsulates the concept, such as "IfcBuilding" for buildings and "IfcWall" for walls. Similarly, for attributes in filter and requirement checks, if the value can be retrieved directly from a standard property set, the target is set to that property.

The second scenario handles indirect designation, where direct IFC mapping is insufficient. For core concepts, this involves introducing a custom property to relevant objects for functional identification. For example, for spaces such as conference rooms or cinemas, we create custom properties like "Conference Room" or "Cinema" under attribute label "Function". For filters and requirements, this applies when target values must be derived through logical computations. To support these checks, we define different data types for the target values: integer, floating-point, and Boolean. Integer and floating-point data are used for quantifiable scenarios, such as quantity, distance, ratio, and position, while Boolean data is used for judgment-based situations, such as topological assessments. This data type allocation can also match the return value of atomic functions, which we will discuss in detail in the next section.

Before proceeding with rule execution, it is important to emphasize the necessity of ensuring that the BIM model includes all required data for the checks. In practice, this could be done through an MVD (Model View Definition) or IDS (Information Delivery Specification) validation process. MVD ensures that the appropriate data sets, properties, and relationships are present in the model by defining specific views of the data, tailored to the needs of the application or rule set. Similarly, IDS sets out the information delivery requirements, ensuring that data is not only present but also of the right quality and format for rule execution. These validation processes guarantee that

the mapped target values exist and can be retrieved for checking.

3.1.3. Atomic function mapping

To facilitate the utilization of graph representation by computers for ARC, previous studies have expanded the graph by adding new edges and nodes. However, this approach frequently results in the graph becoming more complex and therefore losing some expressiveness. In this study, we employ atomic functions to represent computable logic. Atomic functions constitute a common unit of calculation in the review process that can integrate computable methods. For instance, the atomic function "getSpace" retrieves all IfcSpace entities that possess a specific custom property value; the function "hasElement" determines all IfcElement entities that belong to a given input object. We need to add an atomic function node to each semantic subgraph for specific usage. The following discussion will address the functionality of atomic functions and the way they are matched to each semantic part.

(1) Core Concepts. The atomic function here refers to a method that retrieves all objects in the model corresponding to a specified target value. For example, with the core concept "plant", once mapped to the target value "IfcBuilding", the corresponding atomic function "getBuilding" is assigned. This function allows the extraction of all model objects with the IFC type "IfcBuilding" from the set of building entities.

Graphically, an atomic function node is added to the original core concept subgraph to illustrate the purpose described above. This subgraph is illustrated in Figure 5.

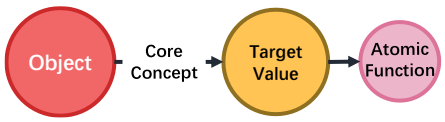


Figure 5 Core concept subgraph with atomic function

(2) Constraint Filter. The role of an atomic function here is to retrieve the filter attributes of the filter object, thereby preparing for further evaluation. For example, for the filter condition "liquids with a flash point less than 28 degrees", we can match the atomic function "getProperty" to obtain the flash point attribute from the filter object. In the development of the inspection code, the core concept is filtered by comparing this filter attribute with the target value. This subgraph is illustrated in Figure 6.

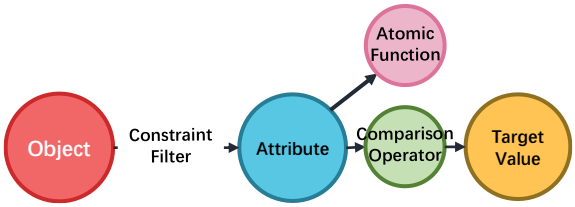


Figure 6 Constraint filter subgraph with atomic function

(3) Requirement Condition Subgraph. The atomic function's role here is to retrieve the requirement attribute of the requirement object. For instance, for the requirement condition "the sum of the areas of doors, windows, and openings should not exceed 5% of the exterior wall area", the corresponding atomic function would be "getWindowWallRatio" to obtain the required hollow wall area ratio. This subgraph is illustrated in Figure 7.

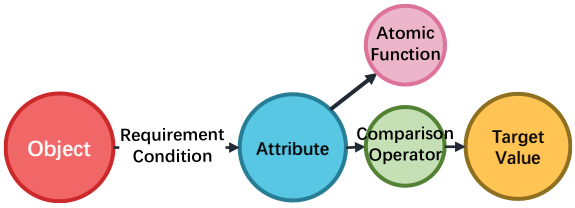


Figure 7 Requirement condition subgraph with atomic function

Ultimately, by integrating the subgraphs and consolidating common object nodes, we obtain the total graph representation corresponding to the textual clause, as shown in Figure 8. Each graph structure comprises subgraphs corresponding to the three distinct semantic functions: core concepts, constraint filters, and requirement conditions. This final graph comprehensively represents the semantic logic of the textual clause and includes the computational methods for automatic rule checking of the clause. We offer some typical examples of the graphic representation of textual clauses in Section 0.

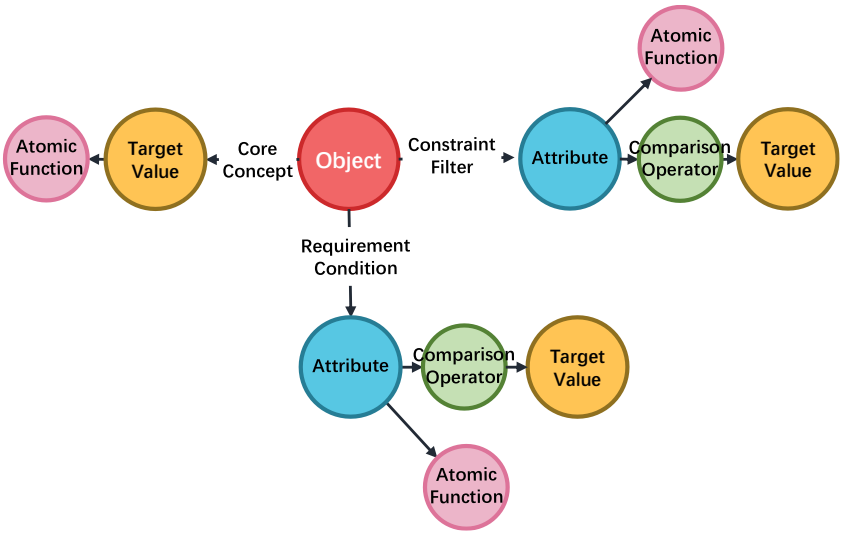


Figure 8 Standard graph representation of a textual clause

In summary, a textual clause is mapped into three semantic functional parts: core concepts, constraint filters, and requirement conditions. For each of these categories, a graphic representation method was proposed to convert each part into a subgraph as the interpretation results at the semantic level. For computational logic, atomic functions are introduced by adding nodes to the subgraph. Each clause is then mapped to a connected graph, with each semantic functional part (core concept, constraint filter, requirement condition) corresponding to a subgraph representing a connected branch. The graph will contain the semantical interpretation results and the computational methods for corresponding rule checking. The iteration through each rule can result in a graphical database of the interpretation results.

3.2. Executable code generation based on knowledge graphs

3.2.1. Development of atomic functions

In the previous section, we defined atomic functions for implementing common computational logic. These functions, which may be applied across multiple regulatory clauses during the review process, should be initially defined and developed so that they can be readily called for subsequent usage.

This study employs the definition methodology of atomic functions as outlined in [8]. The naming convention adheres to the camel case, where each function begins with a verb followed by a noun or adjective that captures its purpose. The verbs are typically

categorized into three types: "get", "is", and "has". Functions prefixed with "get" are designed to retrieve objects or target values, returning data types such as containers, strings, or numerical values. Functions beginning with "is" are used to determine whether an object belongs to a specific category, yielding a Boolean result. Lastly, functions starting with "has" return specific objects contained within another object, either as a list or a Boolean.

From a semantic structure perspective, atomic functions related to core concepts are predominantly prefixed with "get", mainly employed to retrieve instance objects from the model. In the context of our study, which focuses on the IFC format BIM model, the core concepts can usually be obtained directly by retrieving the "IfcProduct" category they belong to. For instance, "IfcBuilding" can be used for buildings, "IfcWall" for walls, and the corresponding functions, "getBuilding" and "getElement". Generally, the core concepts of review covered in the clauses are divided into three categories: building, space, and element. Buildings correspond to physical buildings such as 'IfcBuilding' and 'IfcBuilding' in the IFC category, spaces correspond to 'IfcSpace' in the IFC category, and elements correspond to 'IfcElement' in the IFC category. An atomic function has been proposed for the retrieval of each category: "getBuilding", "getSpace", "getElement".

The functions called constraint filters and requirement conditions need to fulfill some specific functionality. These atomic functions are divided into two categories: low-order and high-order functions. Low-order functions handle fundamental, specific tasks, primarily concerning the direct retrieval of attribute information from regulatory clauses or the model, such as the intended use of a building or the material properties of components. High-order functions, on the other hand, build upon these lower-order functions, enabling more abstract and sophisticated operations, such as spatial distance calculations, connectivity analysis, and mechanistic feature extraction. Based on the clauses selected through our data processing and filtering, we ultimately extracted 27 atomic functions according to their functionality for use, as shown in Table 1. For the specific development process, the open-source library IfcOpenShell was used to manipulate IFC format.

Table 1 Atomic Functions for getting attribute

Functionality	Atomic Functions
Attribute Acquisition	getProperty, getFloorArea, getElementWidth, getSpaceWidth, getSpaceLength, getElementLength, getElementLocation, getElementHeight
Geometric Calculation	getElementDistance, getBuildingHeight, getDistance, getWindowWallRatio, getSpaceDistance, getLinearDistance
Topology Check	isAdjacent, isAccessible, isOpened, isConnectedTo, isExternal, isGroupArranged, isFacedDirectly, isOpenDirection
Membership Judgment	hasElement, hasSpace, hasExternal, hasBuilding
Quantity calculation	getNumberOfElement

3.2.2. From Graphic Clauses to Automated Code

The branching structure of the graph encapsulates the semantic information of the clause, where each computational step is mapped to a corresponding atomic function. Leveraging these atomic functions, this section presents a method for automatic code generation. The process translates the graphic representation of clauses into executable code by integrating the respective atomic functions to facilitate automated rule checking.

This method is designed to generate executable code for checking compliance with individual regulatory clauses. The scope of generatable code is defined by the library of pre-developed atomic functions. Specifically, our approach can automatically construct the logical flow (e.g., sequencing, conditional filtering, and evaluation) for checking rules that can be decomposed into the following three operational steps, which are common in building codes and standards: (1) identifying relevant objects, (2) applying constraint filters, and (3) evaluating requirement conditions. Any clause whose logic can be mapped to this three-step pattern and whose operational primitives (e.g., retrieving objects, comparing attributes) are covered by our atomic function library can be automatically translated into review code.

The specific process for generating review code generally involves the following four steps and the code logic framework is shown in Figure 9. Some typical examples are offered in Section 0.

(1) Capture core concepts. The core concept subgraph is processed to identify all relevant core concepts from the model. These concepts, which match the specified target values, serve as the preliminary review objects. Specifically, the corresponding atomic function is called to form a pre-check set of all instance objects from the model that satisfy the target value.

(2) Constraint Filtering. The constraint filter subgraph is processed to refine the pre-check set. The atomic function is called to get the attributes of each instance in the pre-check set, and the instances that meet the constraints are left to form a check set by comparing them with the target values of the constraints.

(3) Requirement Evaluation. The requirement condition subgraph is evaluated against the core review objects in the check set. Atomic functions are called to acquire the attributes of each object in the check set, which are subsequently compared to the target value. Any object whose attributes do not match the target value is flagged and output as invalid.

Finally, by logically connecting the results from the preceding steps, the complete review code is generated.

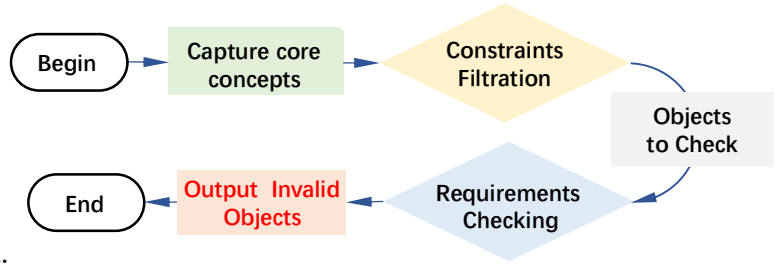


Figure 9 Framework from graphic clauses to executable code

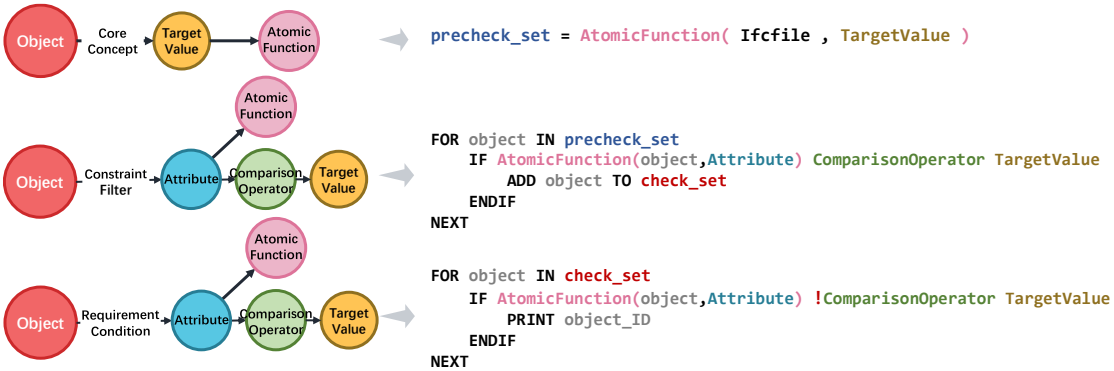


Figure 10 Pseudo-code generation process

By leveraging atomic functions, the generated code can execute complex computational logic to obtain corresponding attributes, based on the capabilities of the function library. Furthermore, to address the complexity arising from intricate clause semantics, we have developed specialized code generation strategies for several common scenarios. These include clauses containing multiple parallel review objects, multiple parallel requirement conditions, and multiple objects with inheritance relationships. We have also explored handling more complex situations, particularly those where filters or requirements involve relationships among multiple objects. Detailed methods for handling these scenarios are demonstrated in the subsequent case study section.

4. Case Study

In this section, we select several representative rules as subjects for applying the method. On one hand, we utilize the knowledge graph-based representation to depict each regulatory clause as its corresponding knowledge graph. On the other hand, we generate Python code based on these knowledge graph representations, which can be used to review the respective clauses. Through the selected case studies, we demonstrate the flexibility of our proposed method in both the knowledge graph representation and the review code generation. This method effectively represents and reviews clauses with complex logical structures. Graph-based representation and code generation

4.1. Graph-based representation and code generation

Four illustrative examples have been selected for the demonstration. These typical

examples encompass essentially the full range of situations we can handle. In fact, most of the clauses are stipulations for a specific object under certain filters, which have been categorized as standard, as illustrated in the first example. The second example can be regarded as a special case of the first one, in which parallel concepts, filters, or requirements are included. Our third and fourth examples are typical of the two broadest cases. The third example pertains to the situation where the core concepts are affiliated, necessitating object filtering based on this affiliation. The final example deals with filters or requirements involving relationships with other objects, which are generally challenging to address. Our approach can represent this logic thanks to the flexibility of graph representations.

To illustrate these examples, the graphical representation of each clause and the automatically generated Python code are provided. We also provide a tabular form of semantic mapping and atomic mapping results in Table 2 for better comprehension.

4.1.1. Case-1 rule (standard knowledge graph representation and code generation method) example

We call a clause standard if it contains one core concept that is subject to several filters and requires checking its requirements. Here, we take the clause "A factory with a fire hazard category of Level D and a fire resistance rating of Level 4 must not exceed one floor" as a standard clause example. Its core concept is "factory"; the constraint filters include "fire hazard category of Level D" and "fire resistance rating of Level 4"; the requirement condition is "the number of floors of the factory must not exceed one". The tabular form of semantic and atomic function mapping results is shown in Table 2. The final knowledge graph representation is illustrated in Figure 11.

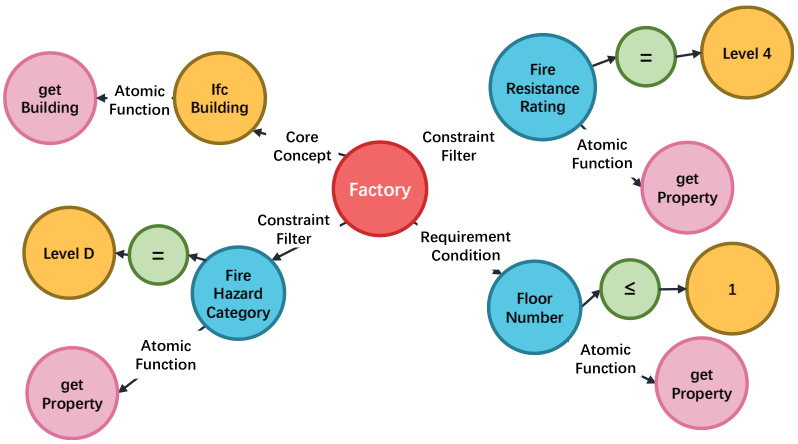


Figure 11 Graphic Representation of case1 clause

The general process for code generation is as follows:

(1) Capture core concepts. The "getBuilding" function retrieves a list of instances where the building is the core object, and the "getElement" function retrieves a list of instances where the element is a wall. The specific code for this step is:

```
list_IfcPlant = getBuilding ( ifc_file , "IfcBuilding" )
```

Figure 12 Case 1 Code Snippet 1

(2) Constraint Filtering. Each object in the list of building instances is processed using the function "getProperty" to filter for buildings with a fire resistance rating of level one. The specific code for this step is:

```
list_IfcPlant = [ obj for obj in list_IfcPlant if getProperty( obj ,
"fire hazard category" ) == "Level D" ]
list_IfcPlant = [ obj for obj in list_IfcPlant if getProperty( obj ,
"fire resistance rating" ) == "Level 4" ]
```

Figure 13 Case 1 Code Snippet 2

(3) Requirement Evaluation. Each object in the list of wall instances obtained in the previous step is processed using the function "getProperty" to determine if its fire resistance limit is at least 3 hours. IDs of wall objects not meeting the requirements are output.

```
for obj in list_IfcPlant:
    if not getProperty ( obj , "Floor Number" ) <= 1:
            print( "Object: %s  NOT MEET: \"A factory with a fire hazard
category of Level D and a fire resistance rating of Level 4 must not
exceed one floor.\"" ) % ( obj.GlobalId )
```

Figure 14 Case 1 Code Snippet 3

(4) Integration of Code. Finally, concatenate the snippet code described above to obtain the complete code for the clause.

```
# "A factory with a fire hazard category of Level D and a fire
resistance rating of Level 4 must not exceed one floor"
ifc_file = ifcopenshell.open( ifc_file_path )
list_IfcPlant = getBuilding ( ifc_file , "IfcPlant" )
list_IfcPlant = [ obj for obj in list_IfcPlant if getProperty ( obj ,
"fire hazard category" ) == "Level D" ]
list_IfcPlant = [ obj for obj in list_IfcPlant if getProperty ( obj ,
"fire resistance rating" ) == "Level 4" ]
for obj in list_IfcPlant:
    if not getProperty ( obj , "Storey Number" ) <= 1:
            print( "Object: %s  NOT MEET: \"A factory with a fire
hazard category of Level D and a fire resistance rating of Level 4
must not exceed one floor.\"" ) % ( obj.GlobalId )
```

Figure 15 Case 1 Code

4.1.2. Case-2 rule (multiple parallel semantic objects) example

In textual clauses, there are sometimes parallel semantic objects. When encountering this situation, each semantic part should be clearly defined. Specifically, each core concept should be a specific noun, and each filter or requirement condition should be a precise statement without logical conjunctions such as "and" or "or". Therefore, to address these parallel cases, we decompose the regulatory clause into two independent clauses, ensuring that each clause does not contain parallel semantic components. There might be several types of situations.

(1) Multiple Parallel Core Concepts. For example, in the clause "The fire resistance limit of the walls and the anteroom must be no less than 2 hours". Here, "stairwell" and "anteroom" are parallel core concepts that both require examination. We decompose it into:

- The fire resistance limit of the walls must be no less than 2 hours.

- The fire resistance limit of the anteroom must be no less than 2 hours

(2) Multiple Parallel Constraint filters. For example, in the clause "For multi-story factories with a fire resistance rating of Level 1 or Level 2, the straight-line distance from any point inside the factory to the nearest safety exit must not exceed 25 meters", the constraint filter on the core concept "factory" includes two possible target values for the "fire resistance rating" (Level 1 or Level 2), meaning factories with either of these ratings need to be examined. We decompose it into:

- For multi-story factories with a fire resistance rating of Level 1, the straight-line distance from any point inside the factory to the nearest safety exit must not exceed 25 meters.

- For multi-story factories with a fire resistance rating of Level 2, the straight-line distance from any point inside the factory to the nearest safety exit must not exceed 25 meters.

(3) Multiple Parallel Requirement Conditions. For example, in the clause "In residential underground garages, the net height of the driveway must be no less than 2.2 meters, and the net height of the parking spaces must be no less than 2 meters", there are two parallel requirements for the garage: "the net height of the driveway must be no less than 2.2 meters" and "the net height of the parking spaces must be no less than 2 meters", both of which need to be examined. We decompose it into:

- In residential underground garages, the net height of the driveway must be no less than 2.2 meters.

- In residential underground garages, the net height of the parking spaces must be no less than 2 meters.

Following this, we can process each decomposed clause using the methods we proposed in Case 1.

4.1.3. Case-3 rule (multiple core concepts with hierarchical relationships) example

In some cases, a normative clause may contain multiple core concepts that are organized into a hierarchical structure, where the core concepts exhibit a subordinate relationship.

The clause "For buildings with a fire resistance rating of Level 1, the fire resistance

of the firewall must not be less than 3 hours" involves two instance objects: "building" and "firewall". In such cases, we handle these two core concepts as a hierarchical structure with containment relationships. Here, "building" acts as the parent object encompassing the child object "firewall", with both being core concepts, as shown in Table 2. The corresponding knowledge graph can be found in Figure 16.

In the subgraph, multiple core concepts with hierarchical relationships are represented as two object nodes. The parent object node points to the child object node, indicating that the parent object contains the child object.

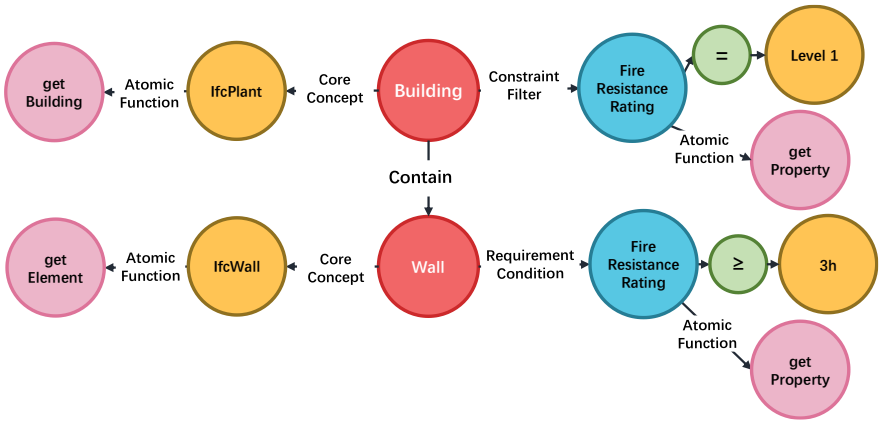


Figure 16 Graphic Representation of case3 clause

In the code generation process, it is imperative to incorporate an additional segment of code to assess hierarchical relationships. The fundamental logic of this code involves employing a membership judging function on the parent object to extract the list of corresponding child objects. The checking objects are then derived from the intersection of this child object list and the preliminary filtered object list obtained in the preceding step. For instance, in this example, each plant object in the list should be subjected to the "hasElement" function to retrieve a comprehensive list of all component child objects. This list is subsequently intersected with the previously obtained list of wall objects to ascertain the final core examination object list. The specific code is as follows:

```
for parentobj in list_IfcPlant:
        subobjlist += [ subobj for subobj in hasElement ( parentobj ) ]
list_IfcWall = [subobj for subobj in subobjlist if subobj in list_IfcWall]
```

Figure 17 Case 3 Code Snippet 1

The final code is as follows.

```
# "For buildings with a fire resistance rating of Level 1, the fire resistance
of the firewall must not be less than 3 hours"
ifc_file = ifcopenshell.open( ifc_file_path )
list_IfcPlant = getBuilding ( ifc_file , "IfcPlant" )
list_IfcPlant = [ obj for obj in list_IfcPlant if getProperty ( obj, "fire
resistance rating" ) == " Level 1" ]
list_IfcWall = getElement ( ifc_file , "IfcWall" )
subobjlist = []
for parentobj in list_IfcPlant:
        subobjlist += [ subobj for subobj in hasElement ( parentobj ) ]
list_IfcWall = [subobj for subobj in subobjlist if subobj in list_IfcWall]
for obj in list_IfcWall:
        if not getProperty ( obj , "fire resistance rating" ) >= 3:
                print( "Object: %s  NOT MEET: \"For buildings with a fire
resistance rating of Level 1, the fire resistance of the firewall must not be
less than 3 hours\"" ) % ( obj.GlobalId )
```

Figure 18 Case 3 Code

4.1.4. Case-4 rule (attribute involves multiple objects) example

In standard semantic logic expressions, the instances in clauses usually appear as core concepts, with the filters and requirements being limitations on the attributes of these core concepts themselves. However, there are instances where filters or requirements involve other objects beyond the core concept, necessitating more complex semantic modeling methods. One such scenario is when the attributes of filters and requirements involve additional instances.

Consider the regulatory clause: "The staircase in the building should ideally extend to the roof". This clause involves two instance objects: "staircase" and "roof". Here, "staircase" is considered the core object. The requirement is to determine whether the staircase is connected to the roof, with the attribute being "connected or not".

To determine the "connected or not" attribute, inputs from both instance objects are required: the core concept "staircase" and the "roof". Each instance object has its branch, detailing the IFC category and the atomic functions used to obtain that instance object. The atomic function "isAccessibleWith" is used to output a Boolean value indicating whether the connection exists. The graphical representation of this is illustrated in Figure 19.

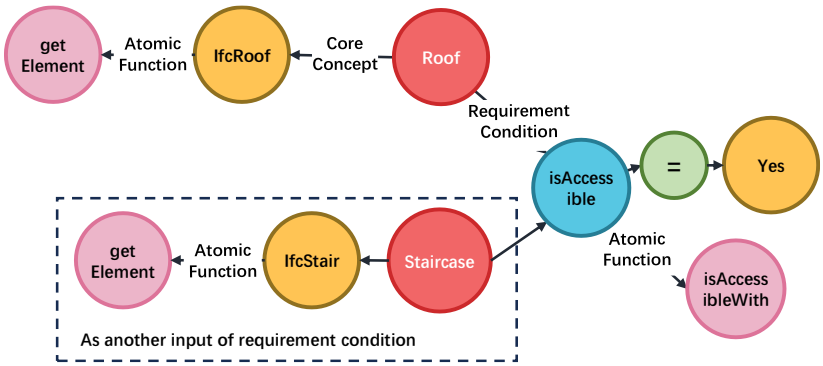


Figure 19 Graphic Representation of case4 clause

Given that the requirement clause involves two instance objects, the staircase and the roof, the code generation needs to address this requirement condition with particular attention. The core of the evaluation code is to check the roof and determine whether it is connected to the stair, which can be assessed using the atomic function "isAccessible". This atomic function takes input lists of staircases and roofs and checks whether any element in the roof list is connected to the staircase. In the design of the "isAccessible" atomic function, the second element is allowed to be a list data type. If the requirement is not met, the IDs of the staircases that do not meet the requirement are output. The final code is as follows.

```
# The staircase in the building should ideally extend to the roof
ifc_file = ifcopenshell.open( ifc_file_path )
list_IfcRoof = getElement ( ifc_file , "IfcRoof" )
list_IfcStair = getElement ( ifc_file , "IfcStair" )
for obj in list_IfcRoof:
        if not isAccessibleWith ( obj , list_IfcStair ) == 1:
                print( "Object: %s  NOT MEET: \"The staircase in the
building should ideally extend to the roof  \"" ) % ( obj.GlobalId )
```

Figure 20 Case 4 Code

Table 2 Examples of semantic mapping and atomic function mapping

Textual Clause	**Core Concept**			**Constraint Filter**				
	Object	Target value	Atomic function	Object	Attribute	Comparison symbol	Target value	Atomic function
A factory with a fire hazard category of Level D and a fire resistance	Factory	IfcBuilding	getBuilding	Factory	Fire hazard category	=	Level D	getProperty

rating of Level 4 must not exceed one floor				Factory	Fire resistance rating	=	Level 4	getProperty
The staircase in the building should ideally extend to the roof	Roof	IfcRoof	getElement	-	-	-	-	-
For buildings with a fire resistance rating of Level 1, the fire resistance of the firewall must not be less than 3 hours	Building	IfcBuilding	getBuilding	Factory	Fire resistance rating	=	Level 1	getProperty
	Wall	IfcWall	getElement					

668 Table 2 (Continued) Examples of semantic mapping and atomic function mapping

Requirement				
Object	Attribute	Comparison symbol	Target value	Atomic function
Factory	Floor number	⩽	1	getProperty
Staircase Roof	isAccessible	=	1	isAccessibleWith
Wall	Fire resistance rating	⩾	'3h'	getProperty

669

670 4.2. Experiments and proof of codes

671 This section continues with the methods proposed earlier and focuses on verifying
672 two aspects: First, by selecting a subset of regulatory clauses, it assesses the feasibility
673 of semantic modelling, specifically whether it is possible to represent these clauses in
674 logically correct graphical form. Second, it examines the feasibility of automated code
675 generation based on the semantic modelling results, i.e., whether the generated code
676 meets the requirements for automated review.

677 The "Chinese Code for Fire Protection Design of Buildings" (GB 50016-201(4))
678 was used as the normative source for case analysis. After preprocessing the clauses, 238
679 clauses were selected for case study analysis. These clauses cover both the simple and
680 complex clause cases mentioned above and indicate the usefulness of our approach.

4.2.1. Construction of the knowledge graph basement

For semantic mapping, we first adopted the NLP-CFG-based semantic annotation and parsing method[27] before pre-transforming text clauses into syntax trees. Based on these syntax trees, it is possible to initially define the proposed three types of semantic parts (core concepts, constraint filters, requirement conditions). We manually matched the target value to ensure it corresponds to the correct IFC class or the correct data format.

The next step is atomic function mapping. We mapped atomic functions manually. The node values to be mapped were first output, including the object in core concepts and the attributes in constraint filter or requirement condition, in the form of an Excel spreadsheet. The spreadsheet was then used to match the atomic functions to corresponding objects or attributes and ultimately to form the complete graphical model.

After semantic mapping and atomic function mapping, a program was developed to convert the clauses into Neo4j graph form (Figure 21). Each connected component corresponds to a complete semantic structure to be reviewed. 238 clauses generated 319 connected components, containing 3,886 nodes and 3,567 relationships. The number of connected graphs exceeds the total number of clauses because some contain parallel semantic objects, as illustrated in case 2. The existence of nodes with label "Robj" is caused by the hierarchical relationship as illustrated in case 3. Statistics on the frequency of use of atomic functions in the analyzed clauses are shown in Table 3, indicating that the selected clauses covered a variety of review objects, including various buildings, spaces, and elements, and that some complex computational logic is involved, including topological, quantitative aspects of the checking.

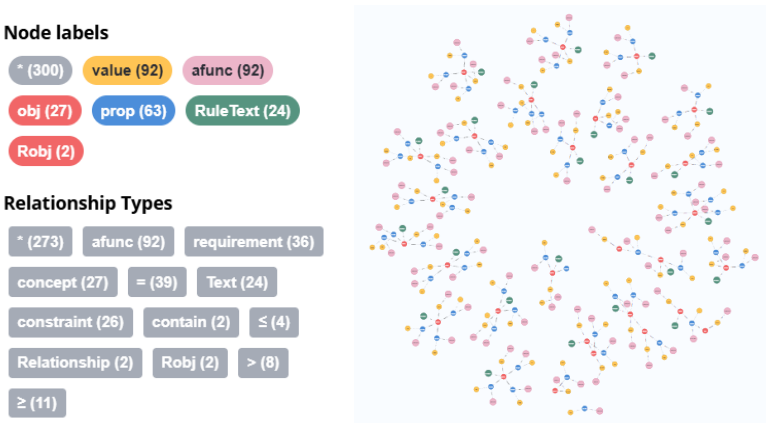


Figure 21 Some connected branches in the Neo4j graph database

Table 3 Frequency statistics table of atomic functions

Atomic Function	Frequency
getBuilding	235
getSpace	111
getElement	138
getProperty	664
isAccessibleWith	1
hasSpace	53
hasElement	92
getNumberOfElement	1
getElementWidth	13
getLinearDistance	18

4.2.2. Code generation validation

The code generation was performed for all 238 selected clauses based on their graphical representations. To further validate the effectiveness of the code, a building model was constructed in Revit and subsequently exported as an IFC format file to serve as the object for code execution. The selected regulations involved various types of instance objects, including architectural instances such as "factory", "high-rise residential building", and "public building", as well as spatial instances like "fire compartment", "karaoke hall", and "public restroom". In the actual case validation, to complete the review within a single model, all building instances in the code were processed as "IfcBuilding", and multiple named zones were set in the model. Additionally, properties were assigned to the respective instances according to the attribute values specified by the reviewed clauses, including some erroneous attributes to test the code's ability to effectively identify errors. For example, to check the clause "A civil building with fire resistant rating level 1 or level 2, the maximum fire protection zone area is 1500 m2", we set the fire protection zone as 2000 m2. Ultimately, 20 clauses were specifically chosen, ensuring that the model contained instances relevant to these clauses. The code performs well to figure out the ID of the invalid objects.

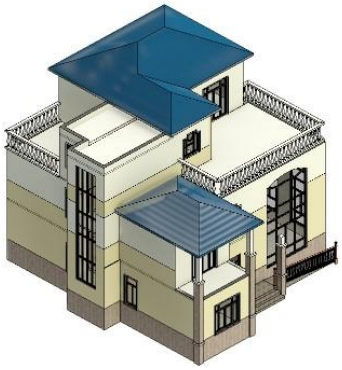


Figure 22 The constructed Revit model for case verification

5. Discussion

In terms of rule interpretation for ARC, most of the existing methods can deal with some simple clauses containing first-order logic but are weak in dealing with complex computational logic. At the same time, these methods can continue to improve the comprehensibility of rule interpretation results. To comprehensively interpret rules at the semantic and computable logic levels, ensuring comprehensibility, we propose a graph-based rule representation method. This method converts text clauses into graphs to express semantics and computable logic using atomic functions. This paper realizes the transformation of text clauses to graphs by semantic mapping and atomic function mapping and provides a method of automatically generating execution code according to the graph. When combined with the development of atomic functions, this approach can be applied to the process of automatic review. Our verification highlights the following implications:

1. The present study emphasizes the use of the semantic mapping approach to facilitate the transformation of clauses into graphs. The semantics of clauses are divided into three semantic parts: the core concept, the constraint filter and the requirement condition. Each part constitutes an individual branch, thereby elucidating the representation logic of the graph. Consequently, the semantics of text clauses are represented in the graph as nodes and edges, a method that has been proven to be more comprehensible and more suitable for the whole ARC system development process.

2. The atomic function mapping approach is used to express computational logic. The existing rule interpretation methods are somehow weak in the realization of complex computational logic, and using atomic functions as a common computational

logic representation provides a way to solve the problem. In combination with the previous expression of semantic logic, the incorporation of atomic functions results in the complete graphic representation of the rule, encompassing both semantic logic and computational logic, which can be processed further by computers. Graphically, each atomic function is represented as a node, which reduces the complexity of the graph compared to previous work and makes it easier to understand.

3. In the case study, some semantic mapping and atomic function mapping procedure involves manual processes. However, these manual steps can be somewhat replaced by automation. The procedure for semantic mapping can be executed in a specific manner that involves NLP[11], and the mapping of atomic functions could be assisted by LLMs[8]. In this process, the development of atomic functions is the only stage that necessitates manual intervention. However, given that it is a common computing unit, the cost of manual development is relatively low, and it is more flexible in terms of calling and debugging.

Limitations of several aspects of this study were identified and can be investigated in the future:

1. The graphical form representation method remains an area of significant potential for further development. Graphical representation has been shown to facilitate the intuitive expression of complex logical information, which itself can be further expressed in a more intricate graphical form. Graphs have great potential in complex logical representation.

2. The atomic function database can be extended to include a wider range of domains. When encountering other areas, such as earthquake resistance, energy and sustainability, additional atomic functions may be required. Future research could focus on enriching atomic functional databases to accommodate regulatory requirements in different fields.

3. It is possible to supplement automatic code generation methods. The automatic code generation method developed in this study can be applied to several atomic functions used in verification. When the atomic function changes, that is, when the input and output of the atomic function may have other conditions, and the law needs to be further summarized for the semantic-specific expression of the atomic function and the article clause itself.

6. Conclusion

With the advancement of digitalization and informatization in the construction industry, ARC has gained significant attention. However, existing methods for rule interpretation are often limited in their ability to represent complex computational logic, and the integration between complex logic representation and rule interpretation outcomes remains insufficient. To address these challenges, this study proposes a graph-based rule interpretation approach to represent textual clauses in graphic forms. Firstly, clauses are initially divided into three distinct concepts: the core concept, the constraint filter and the requirement condition through semantic mapping. The graph representation of each part is proposed, and the combined graph is formed as the interpretation results at the semantic level. Following this, an atomic function matching process is employed, which adds atomic function nodes to the corresponding branch of the graph. These nodes serve as the computational logic required to execute the branch, thereby achieving the result at the computational level. Then, based on this graph representation, an automatic generation approach of executable code is proposed. Combined with the development of atomic functions, these codes can be used for model checking. The feasibility and effectiveness of the proposed method are then verified through an experiment, which highlights several contributions. Firstly, the graph representation method combines the interpretation results at the semantic level and computational level, which can represent complex computational logic and enhance the interpretability of the results. Secondly, the utilization of graphs interprets results more comprehensibly, and our graphs are more concise. Furthermore, our approach possesses considerable potential for implementation at the fully automated level and can be further utilized as a reference for the development of the ARC system.

Acknowledgment

The authors are grateful for the financial support received from the National Key Research and Development Program of China (No. 2023YFC3804600) and National Natural Science Foundation of China (No. 52378306) .

Declaration of Competing Interest

The authors have no competing interests to declare that are relevant to the content of this article.

Author Contribution Statement

All authors have given approval to the final version of the manuscript.

Appendix

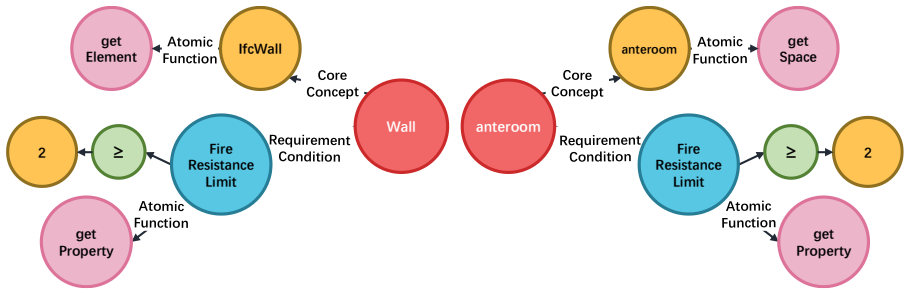


Figure 23 Graphic Representation of case2-1 clause

```
# " The fire resistance limit  of the anteroom must be no less than 2
hours ".
ifc_file = ifcopenshell.open( ifc_file_path )
list_IfcSpace = getSpace ( ifc_file , "anteroom" )
for obj in list_IfcSpace:
    if not getProperty ( obj , "Fire Resistance Limit" ) >= 2:
             print( "Object: %s  NOT MEET: \"The fire resistance
limit of the walls must be no less than 2 hours .\"" ) %
( obj.GlobalId )
```

```
# " The fire resistance limit of the walls must be no less than 2
hours".
ifc_file = ifcopenshell.open( ifc_file_path )
list_IfcWall = getElement ( ifc_file , "IfcWall" )
for obj in list_IfcWall:
    if not getProperty ( obj , "Fire Resistance Limit" ) >= 2:
             print( "Object: %s  NOT MEET: \"The fire resistance
limit of the walls must be no less than 2 hours .\"" ) %
( obj.GlobalId )
```

Figure 24 Case 2-1 Code

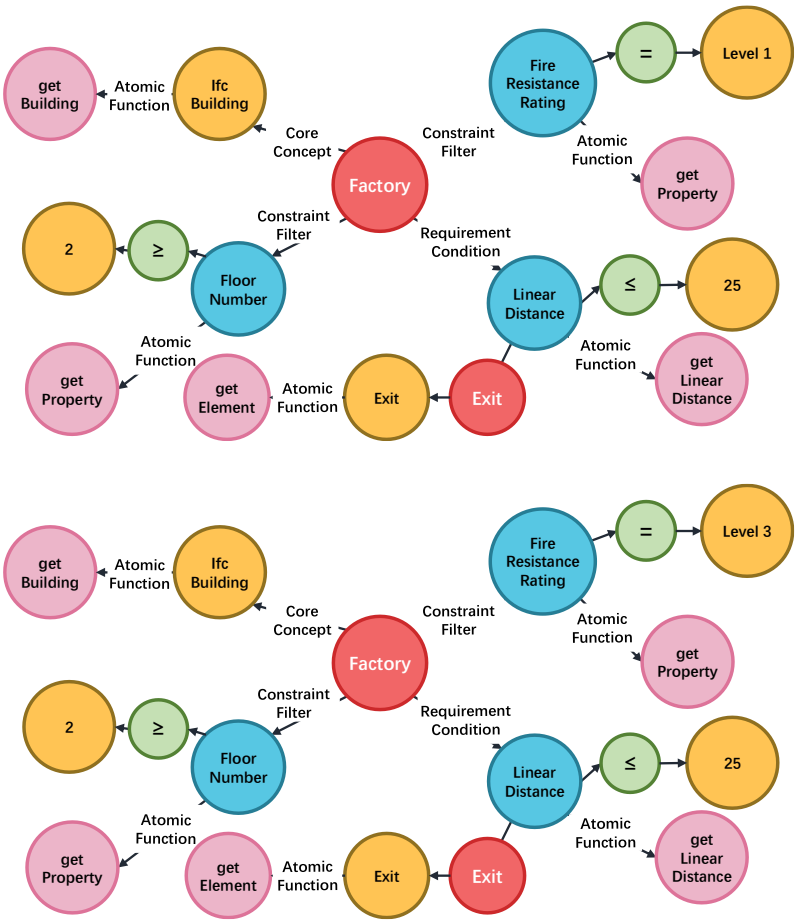
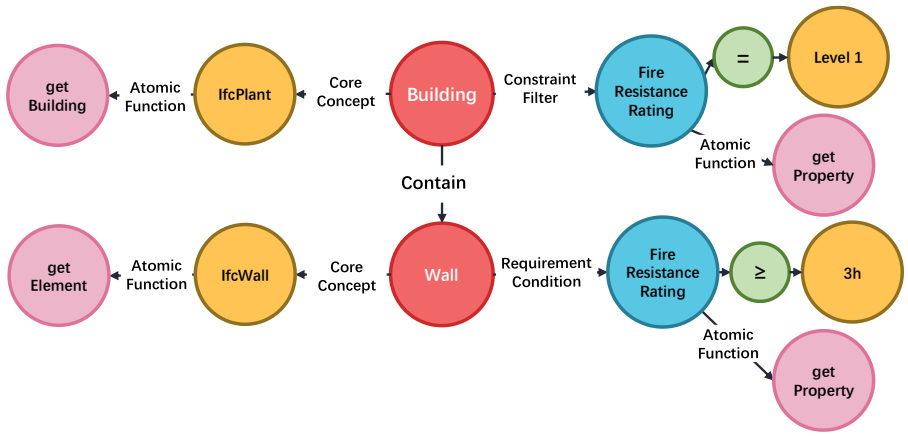


Figure 25 Graphic Representation of case2-2 clause

```
# "For multi-story factories with a fire resistance rating of Level
1, the straight-line distance from any point inside the factory to
the nearest safety exit must not exceed 25 meters. "
ifc_file = ifcopenshell.open( ifc_file_path )
list_IfcPlant = getBuilding ( ifc_file , "IfcPlant" )
List_IfcElement = getElement ( ifc_file , "Exit" )
list_IfcPlant = [ obj for obj in list_IfcPlant if getProperty ( obj ,
"Fire Resistance Rating" ) == "Level 1" ]
list_IfcPlant = [ obj for obj in list_IfcPlant if getProperty ( obj ,
"Floor Number" ) >= 2]
for obj in list_IfcPlant:
    if not getLinearDistance ( obj , List_IfcElement ) <= 25:
            print( "Object: %s  NOT MEET: \"For multi-story
factories with a fire resistance rating of Level 1, the straight -
line distance from any point inside the factory to the nearest
safety exit must not exceed 25 meters .\"" ) % ( obj.GlobalId )
```

Figure 26 Case 2-2 Code Snippet (the other one is similar)



832

833 Figure 27 Graphic Representation of case2-3 clause

```
# "In residential underground garages, the net height of the
driveway must be no less than 2.2 meters ".
ifc_file = ifcopenshell.open( ifc_file_path )
list_IfcSpace = getSpace ( ifc_file , "Driveway" )
for obj in list_IfcSpace:
    if not getProperty ( obj , "Hight" ) >= 2.2:
            print( "Object: %s  NOT MEET: \"In residential
underground garages, the net height of the driveway must be no less
than 2.2 meters.\"" ) % ( obj.GlobalId )
```

834

835 Figure 28 Case 2-3 Code Snippet (the other one is similar)

836

837

Reference

[1] Nawari, N. O. (2018). Building information modeling: Automated code checking and compliance processes. CRC Press.

[2] Soliman-Junior, J., Tzortzopoulos, P., Baldauf, J. P., Pedo, B., Kagioglou, M., Formoso, C. T., & Humphreys, J. (2021). Automated compliance checking in healthcare building design. Automation in construction, 129, 103822.

[3] Zhang, J., & El-Gohary, N. M. (2017). Integrating semantic NLP and logic reasoning into a unified system for fully-automated code checking. Automation in construction, 73, 45-57.

[4] Ilal, S. M., & Günaydın, H. M. (2017). Computer representation of building codes for automated compliance checking. Automation in construction, 82, 43-58.

[5] Motamedi, A., Hammad, A., & Asen, Y. (2014). Knowledge-assisted BIM-based visual analytics for failure root cause detection in facilities management. Automation in construction, 43, 73-83.

[6] Sobhkhiz, S., Zhou, Y. C., Lin, J. R., & El-Diraby, T. E. (2021). Framing and evaluating the best practices of IFC-based automated rule checking: A case study. Buildings, 11(10), 456.

[7] Eastman, C., Lee, J. M., Jeong, Y. S., & Lee, J. K. (2009). Automatic rule-based checking of building designs. Automation in construction, 18(8), 1011-1033.

[8] Zheng, Z., Zhou, Y. C., Chen, K. Y., Lu, X. Z., She, Z. T., & Lin, J. R. (2024). A text classification-based approach for evaluating and enhancing the machine interpretability of building codes. Engineering Applications of Artificial Intelligence, 127, 107207..

[9] Ismail, A. S., Ali, K. N., & Iahad, N. A. (2017, July). A review on BIM-based automated code compliance checking system. In 2017 international conference on research and innovation in information systems (icriis) (pp. 1-6). IEEE.

[10] Zhang, J., & El-Gohary, N. M. (2016). Semantic NLP-based information extraction from construction regulatory documents for automated compliance checking. Journal of computing in civil engineering, 30(2), 04015014.

867 [11] Zhou, Y. C., Zheng, Z., Lin, J. R., & Lu, X. Z. (2022). Integrating NLP and context-
868 free grammar for complex rule interpretation towards automated compliance
869 checking. Computers in Industry, 142, 103746.

870 [12] Solihin, W., & Eastman, C. (2015). Classification of rules for automated BIM rule
871 checking development. Automation in construction, 53, 69-82.

872 [13] Lee, J. K., Cho, K., Choi, H., Choi, S., Kim, S., & Cha, S. H. (2023). High-level
873 implementable methods for automated building code compliance checking.
874 Developments in the Built Environment, 15, 100174.

875 [14] Solihin, W., & Eastman, C. M. (2016). A knowledge representation approach in
876 BIM rule requirement analysis using the conceptual graph. J. Inf. Technol. Constr.,
877 21(Jun), 370-401.

878 [15] Wangara, J. (2018). Quality Management in BIM: Use of Solibri Model Checker
879 and CoBIM Guidlines for BIM Quality Validation.

880 [16] Dimyadi, J., Pauwels, P., & Amor, R. (2016). Modelling and accessing regulatory
881 knowledge for computer-assisted compliance audit. Journal of information
882 technology in construction, 21, 317-336.

883 [17] Nawari, N. O. (2019). A generalized adaptive framework (GAF) for automating
884 code compliance checking. Buildings, 9(4), 86.

885 [18] Hjelseth, E., & Nisbet, N. (2011, October). Capturing normative constraints by use
886 of the semantic mark-up RASE methodology. In Proceedings of CIB W78-W102
887 Conference (pp. 1-10).

888 [19] Beach, T. H., Rezgui, Y., Li, H., & Kasim, T. (2015). A rule-based semantic
889 approach for automated regulatory compliance in the construction sector. Expert
890 Systems with Applications, 42(12), 5219-5231.

891 [20] Horrocks, I., & Sattler, U. (2004). Decidability of SHIQ with complex role
892 inclusion axioms. Artificial Intelligence, 160(1-2), 79-104.

893 [21] Pauwels, P., Van Deursen, D., Verstraeten, R., De Roo, J., De Meyer, R., Van de
894 Walle, R., & Van Campenhout, J. (2011). A semantic rule checking environment
895 for building performance checking. Automation in construction, 20(5), 506-518.

[22] Yurchyshyna, A., & Zarli, A. (2009). An ontology-based approach for formalisation and semantic organisation of conformance requirements in construction. Automation in construction, 18(8), 1084-1098.

[23] Bouzidi, K. R., Fies, B., Faron-Zucker, C., Zarli, A., & Thanh, N. L. (2012). Semantic web approach to ease regulation compliance checking in construction industry. Future Internet, 4(3), 830-851.

[24] Zheng, Z., Zhou, Y. C., Lu, X. Z., & Lin, J. R. (2022). Knowledge-informed semantic alignment and rule interpretation for automated compliance checking. Automation in Construction, 142, 104524.

[25] Song, J., Kim, J., & Lee, J. K. (2018). NLP and deep learning-based analysis of building regulations to support automated rule checking system. In ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction (Vol. 35, pp. 1-7). IAARC Publications.

[26] Zhang, J., & El-Gohary, N. M. (2015). Automated information transformation for automated regulatory compliance checking in construction. Journal of Computing in Civil Engineering, 29(4), B4015001.

[27] Xu, X., & Cai, H. (2021). Ontology and rule-based natural language processing approach for interpreting textual regulations on underground utility infrastructure. Advanced Engineering Informatics, 48, 101288.

[28] Li, S., Wang, J., & Xu, Z. (2025). Automated compliance checking for BIM models based on Chinese-NLP and knowledge graph: an integrative conceptual framework. Engineering, Construction and Architectural Management, 32(6), 3832-3856.

[29] Kruiper, R., Kumar, B., Watson, R., Sadeghineko, F., Gray, A., & Konstas, I. (2024). A platform-based Natural Language processing-driven strategy for digitalising regulatory compliance processes for the built environment. Advanced Engineering Informatics, 62, 102653.

[30] Iversen, O. (2024). Leveraging large language models for BIM-based automated compliance checking of building regulations (Master's thesis, NTNU).

[31] Liu, X. (2025). Exploring the power of Large Language Models: Automated compliance checks in architecture engineering and construction industries (Doctoral dissertation, Cardiff University).

[32] Zheng, Z., Han, J., Chen, K. Y., Cao, X. Y., Lu, X. Z., & Lin, J. R. (2026). Translating regulatory clauses into executable codes for building design checking via large language model driven function matching and composing. Engineering Applications of Artificial Intelligence, 163, 112823.

[33] Chen, N., Lin, X., Jiang, H., & An, Y. (2024). Automated building information modeling compliance check through a large language model combined with deep learning and ontology. Buildings, 14(7), 1983.

[34] Kuske, D., & Schweikardt, N. (2017, June). First-order logic with counting. In 2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (pp. 1-12). IEEE.

[35] Lee, J. K., Eastman, C. M., & Lee, Y. C. (2015). Implementation of a BIM domain-specific language for the building environment rule and analysis. Journal of Intelligent & Robotic Systems, 79(3), 507-522.

[36] Daum, S., & Borrmann, A. (2015). Simplifying the analysis of building information models using tQL4BIM and vQL4BIM. In Proc. of the EG-ICE 2015.

[37] Zhang, C., Beetz, J., & de Vries, B. (2018). BimSPARQL: Domain-specific functional SPARQL extensions for querying RDF building data. Semantic Web, 9(6), 829-855.

[38] Uhm, M., Lee, G., Park, Y., Kim, S., Jung, J., & Lee, J. K. (2015). Requirements for computational rule checking of requests for proposals (RFPs) for building designs in South Korea. Advanced Engineering Informatics, 29(3), 602-615.

[39] Han, J., Lu, X. Z., & Lin, J. R. (2025). Pretrained graph neural network for embedding semantic, spatial, and topological data in building information models. Computer‐Aided Civil and Infrastructure Engineering, 40(26), 4607-4631.

[40] Fenves, S. J. (1966). Tabular decision logic for structural design. Journal of the Structural Division, 92(6), 473-490.

[41] Fuchs, S. (2021). Natural language processing for building code interpretation: systematic literature review report. August, 21, 2023.

[42] Li, S., Cai, H., & Kamat, V. R. (2016). Integrating natural language processing and spatial reasoning for utility compliance checking. Journal of Construction Engineering and Management, 142(12), 04016074.

[43] Preidel, C., & Borrmann, A. (2015). Automated code compliance checking based on a visual language and building information modeling. In ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction (Vol. 32, p. 1). IAARC Publications.

[44] Solihin, W. (2016). A simplified BIM data representation using a relational database schema for an efficient rule checking system and its associated rule checking language. Georgia Institute of Technology.

[45] Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. ACM computing surveys (CSUR), 37(4), 316-344.

[46] The Ministry of Public Security of the People's Republic of China (2014). Code for fire protection design of buildings (GB 50016-2014). (in Chinese)