ITcon
www.itcon.org - Journal of Information Technology in Construction - ISSN 1874-4753

INVARIANT SIGNATURE, LOGIC REASONING, AND SEMANTIC NATURAL LANGUAGE PROCESSING (NLP)-BASED AUTOMATED BUILDING CODE COMPLIANCE CHECKING (I-SNACC) FRAMEWORK

SUBMITTED: July 2021
REVISED: October 2022
PUBLISHED: January 2023
EDITOR: Robert Amor
DOI: 10.36680/j.itcon.2023.001

Jin Wu, Ph.D.
Automation and Intelligent Construction (AutoIC) Lab, School of Construction Management Technology, Purdue University, West Lafayette, IN, 47907, United States
wu1275@purdue.edu

Xiaorui Xue, Ph.D.
Automation and Intelligent Construction (AutoIC) Lab, School of Construction Management Technology, Purdue University, West Lafayette, IN, 47907, United States
xue39@purdue.edu

Jiansong Zhang, Ph.D.
Automation and Intelligent Construction (AutoIC) Lab, School of Construction Management Technology, Purdue University, West Lafayette, IN, 47907, United States
zhan3062@purdue.edu (*corresponding author)

SUMMARY: *Traditional manual building code compliance checking is costly, time-consuming, and human error-prone. With the adoption of Building Information Modeling (BIM), automation in such a checking process becomes more feasible. However, existing methods still face limited automation when applied to different building codes. To address that, in this paper, the authors proposed a new framework that requires minimal input from users and strives for full automation, namely, the Invariant signature, logic reasoning, and Semantic Natural language processing (NLP)-based Automated building Code compliance Checking (I-SNACC) framework. The authors developed an automated building code compliance checking (ACC) prototype system under this framework and tested it on Chapter 10 of the International Building Codes 2015 (IBC 2015). The system was tested on two real projects and achieved 95.2% precision and 100% recall in non-compliance detection. The experiment showed that the framework is promising in automating building code compliance checking. Compared to the state-of-the-art methods, the new framework increases the degree of automation and saves manual efforts for finding non-compliance cases.*

KEYWORDS: *Automated Compliance Checking, Building Codes, Invariant Signature, Building Information Modeling (BIM), Natural Language Processing, Logic Reasoning.*

REFERENCE: *Jin Wu, Xiaorui Xue, Jiansong Zhang (2023). Invariant Signature, Logic Reasoning, and Semantic Natural Language Processing (NLP)-Based Automated Building Code Compliance Checking (I-SNACC) Framework. Journal of Information Technology in Construction (ITcon), Special issue: 'The Eastman Symposium', Vol. 28, pg. 1-18, DOI: 10.36680/j.itcon.2023.001*

1. INTRODUCTION

Traditional manual building code compliance checking exhibits a wide range of shortcomings: (1) prolonged code compliance checking timespan, (2) tedious compliance checking effort, and (3) human error-prone code compliance checking results (Eastman et al, 2009, Dimyadi and Amor, 2013, Preidel and Borrmann, 2018, Xue et al, 2022). One of the main reasons for these shortcomings is the complexity of checking building designs. Plan reviewers check omissions, errors, and non-compliance cases in building designs against building codes. If such instances were found in building designs, designers will need to update the building designs or provide amendments to the building designs. The code compliance checking process continues until plan reviewers have verified that the building designs are error-free, omission-free, and fully compliant with building codes, where a building permit can then be issued. The plurality and diversity of available governing building codes and other regulatory documents make building code compliance checking even more complicated. Building codes can govern an entire building, such as the International Building Code (IBC) (International Code Council, 2015) and the International Fire Code (IFC) (International Code Council, 2000a), or a part of a building, such as the International Mechanical Code (IMC) (International Code Council, 1996) and the International Plumbing Code (IPC) (International Code Council, 1995). Some types of buildings need to comply with building codes specific to their building types. For example, residential buildings need to comply with the International Residential Code (IRC) (International Code Council, 2000b). The shortcomings and complexity of the traditional manual building code compliance checking process made the automation of code compliance checking an urgent demand which has achieved a growing consensus.

An automated code compliance checking system is a fast, inexpensive, and reliable alternative to the traditional code compliance checking method, which relied on manual efforts (i.e., manual interpretation, manual comparison). Recent years witnessed the rapid advancement of automated code compliance checking systems (Eastman et al, 2009, Lee et al, 2020). However, the majority of automated code compliance checking systems require manual conversion of building codes to computer processable formats, such as decision tables (Tan et al, 2010), semantic building code models (İlal and Günaydın, 2017), knowledge models (Dimyadi et al, 2016b), and logic rules (Zhong et al, 2012). (Pauwels et al, 2011) proposed an Industrial Foundation Classes (IFC)-based automated code compliance checking system, which relies on domain experts to convert building codes to semantic web data. (Hjelseth and Nisbet, 2011) proposed a manual mark-up method to convert building codes from normative text to structured logic statements. (Beach et al, 2015) developed a rule-based code compliance checking system that uses ontologies to map entities in IFC models of buildings to building code concepts. (Zhang et al, 2018) combined declarative rules with procedural programming to implement extended functions for querying IFC data that can be used for automated code compliance checking, based on a standard and expressive query language SPARQL and treating simplified properties and relationships as functions used in query runtime. The encoding of regulatory requirements in SPARQL still needed to be manually performed. Solibri Model Checker (SMC) contains a code compliance checking plugin that checks IFC models to a set of manually generated rules (Eastman et al, 2009). Construction and Real Estate Network (CORENET) project is an IFC-based code compliance checking system backed by the Singapore government (Sing and Zhong, 2001). Singapore government provides building codes for checking building designs in a computer-processable format, which were still generated manually as part of the project. KBimCode can check the compliance of IFC models against Korean building code, which relies on domain experts to convert the building code from natural language to computer-processable scripts (Park and Lee, 2016).

To address the heavy manual efforts required in the automated compliance checking system development, the authors propose to build a new integrated and comprehensive framework for automated code compliance checking system framework. The new system framework is named Invariant signature, logic reasoning, and Semantic Natural language processing (NLP)-based Automated building Code compliance Checking (I-SNACC) framework. Compared to existing logic-based BIM checking systems, such as the SPARQL query-based system (Zhang et al, 2018), where the rules were generated manually (e.g., by combining declarative rules with procedural programming), the proposed system supports automatic rule generation. In addition to the integration with automated processing (i.e., information extraction and transformation) of building code requirements and design information, and automated logic-based reasoning, both from the state-of-the-art methods, the authors also: (1) integrated a ruleset expansion method to enable efficient expansion of the range of checkable building code requirements of the automated building code compliance checking (ACC) systems (Xue and Zhang, 2022); (2) powered the extraction of building design facts by the state-of-the-art invariant signature-based model validation

algorithm, which significantly extends the extracted information from BIM instance models targeting the matching with concepts from building codes with high precision and recall (Wu and Zhang, 2022). Invariant signatures are "a set of intrinsic properties of the object that distinguish it from others and that do not change with data schema, software implementation, modeling decisions, and/or language/cultural contexts." (Wu et al, 2021). They can fully represent the BIM elements and be automatically processed into logic facts; (3) facilitated the automated code compliance checking process by designing building code requirement representations that are logic-based and extensible, and building design fact representations that are flexible and generalizable; (4) developed two new modules based on the SNACC prototype system developed by (Zhang and El-Gohary, 2017), namely, a semi-automated modification and verification module to refine the logic rules, and an interactive model validation module to allow developments on generating and correcting logic facts based on input models and building codes; (5) developed a new activation condition module based on checking the existence of entities that the logic rules are associated with; and (6) improved several other modules of the existing SNACC system (Zhang and El-Gohary, 2017).

2. BACKGROUND

2.1. Building information modeling for automated code compliance checking

The emergence of BIM as digital representations led to the development of BIM-based automated code compliance checking tools, such as CORENET e-PlanCheck (Sing and Zhong, 2001), Solibri Model Checker (SMC) (Khemlani, 2002), DesignCheck (Ding et al, 2006), and Compliance Audit Systems (Dimyadi et al, 2020). The potential synergy between BIM and automated code compliance checking also drew the attention of many researchers who have introduced different building code representations that could support automated code compliance checking systems. For example, (Li et al, 2021) proposed a new defeasible reasoning engine to avoid over complicated defeasible rules for normative provisions. (Sydora and Stroulia, 2020) proposed a rule-based language for describing building regulatory requirements in a BIM environment. (İlal and Günaydın, 2017) adopted a four-level building code representation (organizational network, information network, detail representation, and basic data item) to enhance BIM instance models for automated code compliance checking. (Beach et al, 2015) proposed a semantic mapping method to map IFC entities to concepts in building codes. Some researchers developed BIM-based automated code compliance checking systems for different types of built environments. For example, (Xu and Cai, 2020) introduced a semantic schema for heterogeneous data (e.g., ESRI Shapefiles, textual descriptions) with corresponding RDF converters, and a query mechanism with spatial extensions, for the detection of non-compliant utility instances by querying RDF data, to support the compliance checking of underground infrastructures. (Martins and Monteiro, 2013) developed a tool to check the hydraulic design of water distribution systems and utilized IFC as a standard format of information exchange between BIM software and their developed tool. Other researchers leveraged the fact that human is more capable of processing visual signals than processing textual information and proposed to use visual programming language. For example, (Häußler et al, 2020) used visual programming language to describe the process to check the compliance of railway geometry to building code requirements. (Ghannad et al, 2019) proposed an open visual programming language standard and a building code-neutral representation for supporting a BIM rule-checking platform. (Kim et al, 2019) developed a visual representation of KBimCode to aid the generation of machine-readable building code representations. Some research focused on checking the compliance of BIM against target building codes, such as the Florida Building Code (Nawari, 2019), International Building Code 2009 (Zhang and El-Gohary, 2015), and New Zealand Building Code (Dimyadi et al, 2016a).

2.2. Logic programming

The formal foundations of logic programming started in the late 1970's and were further developed in the early 1980's (Alferes, 1996). With its declarative nature, logic programming became a good candidate for knowledge representation. In addition, the close relations with deductive databases made logic programming even more suitable for knowledge representation. Logic programming provides machines with an explicit representation of the knowledge and makes the reasoning independent from implementations. It is context-free and easy to manipulate. Among the implementations of logic programming, Prolog (Max, 2013, Spivey, 1996) is the most widely used logic programming language. It does not require a background in mathematics, logic, or artificial intelligence (AI) to use.

2.3. Part-of-speech

English has eight major POS categories, namely: (1) noun, (2) verb, (3) adjective, (4) adverb, (5) pronoun, (6) preposition, (7) conjunction, and (8) interjection (Butte College, 2016). Words under the same part-of-speech (POS) category have the same grammatical or syntactic function (Petrov et al, 2011). POS tagging is a classification task for POS taggers to classify words into corresponding POS categories (Bird et al, 2009). A word can have multiple POS categories, and the POS category of a word can vary in different contexts. For example, the word "can" can be a modal verb or a noun according to its contexts. POS taggers started as rule-based taggers that either use expert-generated rules (Bird et al, 2009) or algorithm-generated rules (Brill, 1992). Then, the development of machine learning shifted POS taggers to the use of statistical models, such as Support Vector Machines (Giménez and Marquez, 2004), Decision Trees (Giménez and Marquez, 2004), or Hidden Mark Models (Brants, 2000). The advancement of deep learning also leads to the development of POS taggers that use deep learning techniques, such as Long-short Term Memory (Pota et al, 2019), Recurrent Neural Network (Shao et al, 2017), and attention mechanism (Cai et al, 2019). Syntactic information carried by POS tags has a wide range of applications in the construction domain. For example, (Zhou and El-Gohary, 2018) enriched building energy codes by POS tagging and developed a schema to match objects in BIMs to corresponding objects in building energy codes. (Zhang and El-Gohary, 2016) leveraged syntactic information with semantic information to automate the extraction of regulatory information from building code requirements.

3. PROPOSED NEW INVARIANT SIGNATURE, LOGIC PROGRAMING, AND SEMANTIC NATURAL LANGUAGE PROCESSING (NLP)-BASED AUTOMATED BUILDING CODE COMPLIANCE CHECKING SYSTEM (I-SNACC) FRAMEWORK

The authors propose a new framework to create, incorporate, and improve different components of automated code compliance checking systems to expand the checking range and enhance automation. The new framework is called an Invariant signature, logic programing, and Semantic NLP-based Automated building Code compliance Checking (I-SNACC) system framework. While the new system's name implicates its inheritance from the SNACC system (Zhang and El-Gohary, 2017) developed, the new system is different in many aspects, such as the level of inference, the logic rule validation, and the automated building design model pre-processing and validation, thus I-SNACC improved the degree of automation upon SNACC in the overall ACC process. The workflow for ACC systems under the I-SNACC framework is shown in Fig. 1. It also shows how the system processes the inputs and interacts with the users to generate the final compliance report. The inputs include an ontology, building codes, and IFC-based building design models.

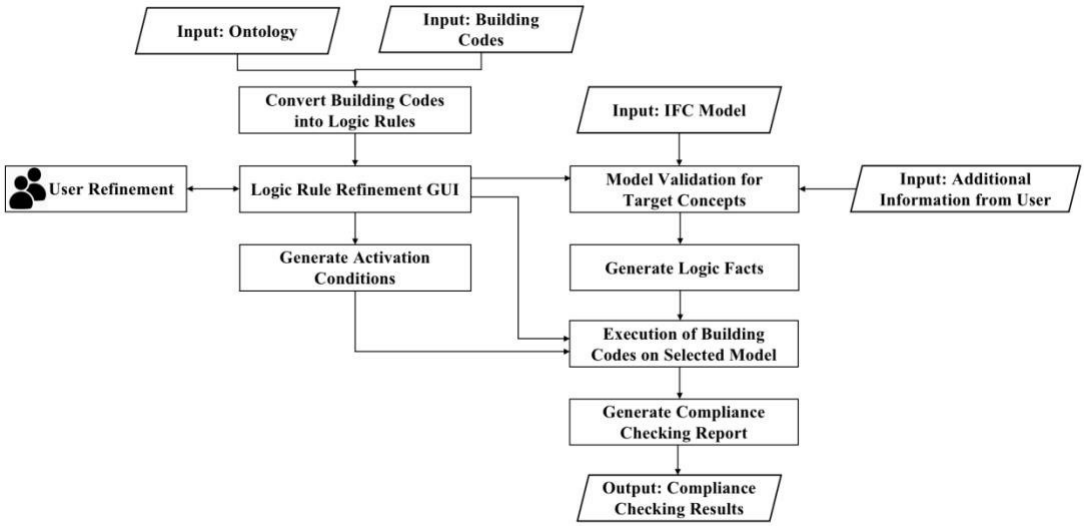

Fig. 1: *The workflow of the new I-SNACC system framework.*

To implement such a system, the authors proposed a new eight-step process to produce desirable results, as shown below.

1. Set up environment and identify system functions
2. Connect to refined logic rule generation module
3. Develop logic rule modification module
4. Develop new activation conditions module
5. Develop model validation module
6. Expand and verify the rule execution module
7. Connect to compliance checking report generation module
8. System testing and iterative improvement

In each step, one module of the system was either developed, connected, or refined. Graphical User Interfaces (GUI) were developed to interact with users, i.e., to present the information and allow the users to select and modify the information.

3.1. Set up environment and identify system functions

Before solving a problem, it is always important to clarify the problem. In this research, the integrated system needs to include and organize all necessary system functions, such as building code selection, generation, refinement; model selection, validation; logic facts generation; rule execution, and reports generation. It is essential to achieve those functions by reusing existing modules as much as possible, developing additional modules as necessary, and checking for compatibility during the integration process. To realize such a seamless integration process, each step and function of the system needs to be clearly identified and clarified. For the I-SNACC system framework to achieve success in automated building code compliance checking, each function needs to work correctly and generate expected intermediate results.

3.2. Connect to refined logic rule generation module

Building code requirements are represented as logic rules in the system. The system utilizes a set of pattern matching-based rules proposed by (Zhang and EI-Gohary, 2016) to convert building code requirements from natural language to Horn-Clause-type logic sentences. Pattern matching-based rules utilize syntactic and semantic features, such as POS tags, gazetteer lists, and phrasal structure tags, to extract regulatory information from building codes by matching texts in building codes with discovered patterns. Both syntactic and semantic information of building codes were considered in the extraction patterns of regulatory information. For example, the word “height” is recognized as a noun with its POS tag “NN” (i.e., POS tag for “singular or mass noun”). The word “minimum” in the phrase “minimum clearance” indicates clearance must be equal to or greater than a certain threshold because the word “minimum” carries this semantic meaning in English. Horn-Clause-type logic sentences are a structured format that can avoid ambiguity and support automated logic reasoning. To extend the range of checkable building codes, a ruleset expansion method was developed (Xue and Zhang, 2022), which introduces new pattern matching-based rules to an existing ruleset iteratively. New pattern matching-based rules were developed to capture regulatory information that was missed by the existing ruleset in a sample building code. New pattern matching-based rules were added one at a time until the ruleset identified all regulatory information in the sample building code. The new pattern matching-based rules met two criteria: (1) valid, and (2) general. Valid pattern matching-based rules do not generate logic rules that they are not designed to generate. General pattern matching-based rules need to be applied at least two times in the sample building code. The original ruleset, for example, lacked patterns for extracting regulatory information regarding the comparative relationship between two candidate subjects, thus a corresponding pattern and rule were created. The pattern in the rule is “candidate subject (potential building components that need to be checked), relation verb (verb that describes the relation between subjects or attributes), inter clause boundary relation (conjunction word that connects two clauses), candidate subject.” This rule extracts building code requirements from sentences that contain the pattern. For example, the following sentence in Section 505.2.1 of the IBC 2015 matches the rule: “*The aggregate area of mezzanines in buildings and structures of Type I or II construction shall be not greater than one-half of the floor area of the room in buildings and structures* (candidate subject) *equipped throughout* (relation verb) *with* (inter

clause boundary relation) *an approved automatic sprinkler system* (candidate subject) *in accordance with Section 903.3.1.1 and an approved emergency voice/alarm communication system in accordance with Section 907.5.2.2*." (IBC, 2015). The rule is valid because it only generates logic rules it was designed to generate. It is also general because it was applied in another sentence in Section 1019.3 of the IBC 2015: "*Exit access stairways and ramps in buildings* (candidate subject) *equipped throughout* (relation verb) *with* (inter clause boundary relation) *an automatic sprinkler system* (candidate subject) *in accordance with Section 903.3.1.1, where the area of the vertical opening between stories does not exceed twice the horizontal projected area of the stairway or ramp and the opening is protected by a draft curtain and closely spaced sprinklers in accordance with NFPA 13*." (IBC, 2015). In previous research (Xue and Zhang, 2022), 64 new rules were added into an existing ruleset to extend it to support two additional chapters of building codes. The original ruleset contained 306 rules. In the extension, one chapter of the building code was used as training data, and another chapter was used as the testing data. It is logical to extrapolate that at most 64 additional rules are needed to cover one more chapter of building codes and this is going to further decrease as more chapters are used in training. However, the exact number also depends on the length of a chapter. The experiment in the previous chapter does not reach 100% precision, recall, and F1-score. Therefore, the expanded ruleset is not saturated yet. However, the rapid decrease in the marginal cost of covering additional building code chapters (from the original 306 to the 64) shows the future saturation of the rules is promising.

3.3. Develop logic rule modification module

While the refinement of the logic rule generation module had significantly improved the logic rules to be closer to what a practical system needs, manual improvement of logic rules is still needed to fix the remaining errors as the automatically generated logic rules did not achieve 100% accuracy. However, such manual effort is minimal. Therefore, a manual rule-refinement module was introduced. The errors in the automatically generated logic rules are not extensive. This manual effort in refining the rules is much less than the otherwise manual effort in creating the rules from scratch. While this step involves the users modifying the rules manually, automation is leveraged to minimize the needed manual effort. For example, the module can conduct programming language grammar and syntax checks, format checks, and validity checks on all logic rules automatically. This step is designed for developer users, not for end-users who have less background in logic programming. Developer users are domain experts who are familiar with construction regulatory requirements and logic rule generation. In contrast, the system does not expect end-users to have any expertise in logic rule generation, as the logic rules refined by the developer users can be directly used by the system for automated reasoning. The system is expected to function with similar performance on other building code chapters with manual adaptations, and the needed adaptations are expected to gradually decrease as more development and testing is performed.

3.4. Develop new activation condition module

To allow the execution of the logic rules on the building design logic facts in a controlled manner, it is essential to generate activation conditions. Activation conditions are the logic clause representations that check the existence of entities that the logic rules are associated with. Such activation conditions can help identify and prevent false positives (i.e., detected non-compliant cases that are not really non-compliant) caused by missing information. For example, if a model does not contain any mezzanine, then the building code (in logic rule format) about the size of the mezzanine should not be activated for checking, which will prevent a false-alarm non-compliance case. In order to check the existence of the entities, it is essential to recognize target entities. The I-SNACC system framework supports instance level checking and reporting, i.e., for each rule, all correspondingly related entities are checked and reported. Thus, an algorithm to identify the target entities to be checked is needed. The target entities also need to satisfy certain preconditions. For example, a requirement on the spaces with occupant loads greater than 500 should not be applied to all spaces. The "occupant load greater than 500" is therefore a precondition of the target entity (i.e., space). In summary, the activation condition generation module is required to (1) identify the entities being checked, (2) identify preconditions of checking the entities, (3) generate logic clauses that can link, select, and filter the entities, i.e., by using identified preconditions, and (4) develop linking functions to record all the identified entities.

3.5. Develop model validation module

The model validation module checks for target concepts from building codes and identifies missing information in the building design model (in order to be checked with the building codes).

The model validation module processes IFC models into invariant signatures. The invariant signatures preserve geometric and locational information of the elements from the input BIM model, and can be matched to target concepts in building codes. The model validation process is based on the method presented in (Wu and Zhang, 2022). The developed model validation module classifies target building code concepts into the following four categories: (1) explicit concepts, which are "directly generable from the BIMs"; (2) inferable concepts, which "can be heuristically inferred from the explicit information in the model with consistency"; (3) user-assisted concepts, which "require user judgment"; and (4) system defaults, which are "not representing actual objects from a building design" (Wu and Zhang, 2022). For explicit concepts, the matching is straightforward. For example, a wall object from IFC (e.g., represented as an *IfcWallStandardCase*) can be used to directly generate an instance of the wall concept with all relevant attributes such as length, width, and thickness. For inferable concepts, the algorithm uses heuristic rules to infer the target information such as those for a main entrance. One possible heuristic rule for identifying the main entrance is that it should be a door opening (typically the largest) sitting at the boundary of the building on the first floor. For user-assisted concepts, a graphical user interface (GUI) will guide users to input the missing information, for which the needed user input is minimized based on automated inference to the extent possible. For system default concepts, no action is needed as these concepts do not directly map to BIMs and are only needed in the later reasoning process (Wu and Zhang, 2022). The authors utilized these target concept matching algorithms to conduct model validation for the selected building design model. During this process, each instance of a target concept is identified, with and without the help of user inputs. Then all the matched information is converted to logic facts to represent the building design models with enriched information for code checking. In summary, this module takes logic rules (i.e., representing building code requirements) and BIM instance models (i.e., representing building design) as input, and output logic facts that are ready to be used for checking with the logic rules.

3.6. Expand and verify the rule execution module

To execute the checking and detect non-compliance cases, three sources of input are fed to the execution module: (1) logic rules, (2) logic facts, and (3) activation conditions. During this step, all compliance and non-compliance cases can be detected automatically through logic reasoning.

The execution starts with activation conditions. The activation conditions select and filter logic facts of target concepts based on encoded preconditions. The logic facts that do not meet the preconditions are eliminated to prevent the otherwise resulted false-positive non-compliance cases. The logic facts that meet the preconditions are then checked by the corresponding logic rules. For each filtered entity in the logic facts, logic rules are executed on that entity to detect non-compliance cases. In this way, the I-SNACC system framework can check for design facts' instance level compliance, i.e., each instance of a target building code concept is checked for any possible non-compliance case related to that instance. For example, for the target concept "egress", every egress of the input building design model is detected and checked against rules related to egresses. To represent the checking result, a list is generated for each rule to store the compliance and non-compliance cases corresponding to that rule. In summary, the execution module takes the logic rules, logic facts, and activation conditions as input, and output lists of compliance checking results, in which each rule corresponds to a list and each instance of a target concept corresponds to an element in the list.

3.7. Connect to compliance checking report generation module

After the execution of the logic rules on the logic facts, a report of the compliance checking result will be generated. Because the reporting format and content was changed from the SNACC reporting module (Zhang and El-Gohary, 2017), the algorithms of the SNACC reporting module were not reused. As the directly generated results from the logic reasoning process contain lists of compliance and non-compliance cases, the system needs to convert the lists of compliance checking results into human-readable outputs. To achieve that, the results are linked with the corresponding building code requirements in the original text (instead of logic rules), because the end users are not expected to have a background in logic programming or be able to interpret logic clauses.

In summary, the goal of the report generation module is that the users without any background in logic programming or computer programming in general are able to see and understand the original building code requirements and their corresponding results.

3.8. System testing and iterative improvement

To test the functionality of the proposed I-SNACC system framework, both the usability of each system function and the connection between different system functions shall be tested. For example, after changing a selected building design model in the system, the subsequent modules, such as the model validation module, shall all update the selected model accordingly.

In addition to checking the connection between adjacent functions, the independence between different functions shall also be tested. For example, although building design models are processed after building codes, given that the building code processing and building design model processing are relatively independent, the I-SNACC framework should still allow users to change the building code inputs even after modifying the logic facts. In summary, the system should allow certain flexibility for users to jump around to different functions. For any error identified in this step when using the training models, such as compilation error due to human input, inaccurate results due to inaccurate logic rules or activate conditions or logic facts, the I-SNACC framework is applied to fix the error until the system can achieve 100% precision and 100% recall in non-compliance detection on the training models. Then the system can be evaluated on testing models for final evaluation.

4. EXPERIMENT

The authors followed the proposed framework and developed an I-SNACC system that incorporated building code processing, building design model processing, and building code checking execution and reporting functions. The authors used three real-world project models to conduct the training, and tested the results on two additional real-world models. The testing models were held out during the training period. The authors selected Chapter 10 of the IBC 2015 (IBC, 2015) as the rule bases, which contains many spatial relations and quantitative rules. Aligned with the previous SNACC system, the I-SNACC system also focuses on checking quantitative requirements. Qualitative requirements are out of the scope of this study and could be pursued in future research.

4.1. Set up environment and identify system functions

The authors chose Java as the main language for the integrated system because most existing functions were written in Java, such as invariant signature extraction and object classification in the model validation module (Wu and Zhang, 2022). The IFC processing toolkit (Apstex, 2018) provided libraries to interact with IFC models, and the ProcessBuilder library (Oracle, 2020a) can run Python scripts that some other modules were developed in.

The authors analyzed and summarized ten functions in the I-SNACC system as follows, in comparison with the predecessor SNACC system (Zhang and El-Gohary, 2017). Labels are added to show if the functions are newly added, reused, or modified from the predecessor SNACC system.

1. Select the ontology to use (for processing building code) (reused)
2. Select the building codes (reused)
3. Process building codes and generate logic rules (modified)
4. Verify the logic rules (new)
5. Generate activation conditions (modified)
6. Select the design model to check (reused)
7. Validate the design model (new)
8. Allow user to input missing information (new)
9. Generate logic facts (modified)
10. Check compliance and generate report (modified)

These functions are high level breakdown of the system functionality. During the experiment of applying the proposed eight-step process under the I-SNACC framework, these ten functions were achieved. A breakdown of the functions developed during each step is shown in Fig. 2.

Step Number	Corresponding Description in Framework	Corresponding System Function
Step 1	Set up environment and identify system functions	N/A
Step 2	Connect to refined logic rule generation module	1. Select the ontology to use
		2. Select the building codes
		3. Process building codes and generate logic rules
Step 3	Develop logic rule modification module	4. Verify the logic rules
Step 4	Develop new activation condition module	5. Generate activation conditions
Step 5	Develop model validation module	6. Select the design model to check
		7. Validate the design model
		8. Allow user to input missing information
		9. Generate logic facts
Step 6	Expand and verify rule execution module	10. Check compliance and generate report
Step 7	Develop report generation module	
Step 8	System testing and iterative improvement	N/A

Fig. 2: *System functions and their development under the proposed I-SNACC framework.*

To develop the system functions in the prototype environment, the authors modified the main GUI panel from the predecessor SNACC system (Zhang and El-Gohary, 2017) to connect all the ten functions (Fig. 3). In addition to the ten functions that the users can click the corresponding buttons to activate, the main panel also included a text field that could display the outputs for showing the progress or instructions of the next steps. With the main GUI panel, the environment was set up and ready to be developed with actual functions.

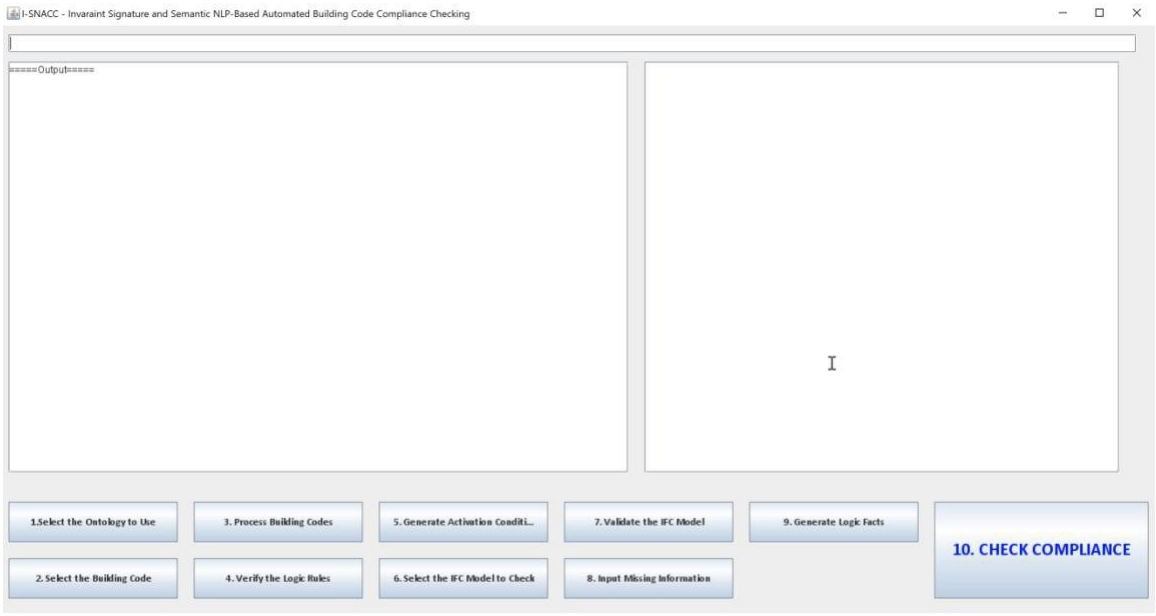

Fig. 3: *A prototype of the I-SNACC system.*

4.2. Connect to refined logic rules generation module

The rule generation module was implemented in Python 2 (Van Rossum, 2007), because the module adopted many NLP packages, which were well developed in Python. In contrast, the I-SNACC system used Java in most of the components. To allow full automation, the system incorporated the rule processing module into the java-based I-

SNACC system using the operating system processes creation package (Oracle, 2020a) for connecting Java to Python scripts.

The authors implemented I-SNACC system processes to enable the selection of building code being checked, the selection of the ontology that the module needs, and execution of the developed Python scripts, of the refined logic rule generation module. Then the system reads and displays the outputs from the module. To store the intermediate results, the authors created a temporary file to store the generated logic rules for further processing. This was needed because in the iterative improvement of Step 8, the temporary file would allow jumping to the current module from different functions.

4.3. Develop logic rule modification module

As described in the proposed framework section, the existing logic rule generation module did not generate 100% accurate results for the logic rules yet, so it is crucial to have a correction module to allow developer users to fix any potential error in the automatically generated logic rules. The basic functions of the logic rule modification module included: (1) presenting the current logic rules, (2) modifying and saving the current logic rules, and (3) jumping between different rules. More importantly, the logic rule modification module shall conduct automated checking and validation on the provided rules to achieve semi-automated rule modification.

For the three basic functions, the authors developed a user-friendly GUI in the system with default options in accordance with user habits. For example, by default, after modifying one logic rule, its immediate next logic rule will be displayed. Another critical function is jumping forward and backward to different rules, e.g., the GUI shall allow users to jump back to a previous logic rule or jump to any logic rules identified by their rule numbers.

As a result, the authors developed the logic rule refinement GUI with six buttons (Fig. 4), one display window for displaying messages, and another console window for taking refined logic rules input. The six buttons' functions are as follows:

1. Go to the next rule and print that rule
2. Fix the current rule by replacing the original rule with the user typed input
3. Go back to the previous rule and print that rule
4. Jump to any rule with a user input rule number
5. Print all logic rules for easy inspection of all the clauses
6. Save current rules and exit the program

Fig. 4: *The developed GUI for logic rule refinement module.*

To promote automation in rule validation and ensure that the users' input is compatible with the system, a rule checking and validation system was embedded in this module.

For grammar, syntax, and format checking, firstly, the input rules must follow a predefined naming standard, e.g., that the rule must start with "*compliance_of_*," otherwise an error message "*Please enter rule following the format: compliance_of_xx(Var):- conditions.*" would be prompted to the developer user. Second, the rule cannot share the same name with other existing rules in the system, which can cause system errors in interpreting the rules, e.g., two rules about hardware should start as "*compliance_hardware1*" and "*compliance_hardware2*" to differentiate them. Third, the predicate for each rule clause cannot use built-in keywords in the backbone B-Prolog language (Zhou, 2014). For example, although "*exit*" is a common concept in a building design, the word "*exit*" cannot be used as a predicate name as it is a built-in predicate to indicate the termination of a logic program in B-Prolog (Zhou, 2014). Alternatively, "*exit_*" or "*exits*" can be used. For example, "*exit_(Exit)*" and "*exits(Exits)*" for one or multiple exits are allowed in the rule, and "*exit(Exit)*" cannot be used. Last but not least, a simulation run of the rule is conducted to check for additional syntactic issues, to ensure the rule can compile without any unexpected

errors. This is done by checking if the logic rule can return true when each conjunct in the body of the rule is true. In this way, common mistakes such as missing a comma, unbalanced parentheses, invalid variables, and inconsistency or typos can be detected and resolved.

In addition to the four checking considerations above, the authors also conducted a validity check for the logic rules using graph theory methods to automatically build graphs composed with nodes and edges. Taking Rule 66 (Fig. 5) for example. In one version of the logic facts, the predicate "*provided(Space, Exits)*" was missing. Fig. 6 shows a visualization of the graphs by representing nodes as circles, and edges as bi-directional arrows. This rule was grammatically correct, but it will not function as expected because the conditions that limit the exits within certain spaces do not take effect without the clause "*provided(Space, Exits)*". This will result in applying the rules to all exits, whereas the rule was expected to only apply to a subset of them. To achieve the validity checking function, the authors used a graph theory method to build each statement as a node, and build each variable as a bi-directional edge to conduct breadth-first-search (BFS) traversal. If each node on the graph can be traversed from any other node, then the rule is connected. For the graph that is not connected, the system was developed to prompt the user with validation error messages saying that the rule is not valid because of the separation of logic clause elements. With the automated checking function, rule modification can be finished more efficiently by further saving otherwise needed manual effort in checking and verification.

```
# Three exits or exit access doorways shall be provided from any
# space with an occupant load of 501 to 1,000.

# Logic Rule
compliance_of_Occupant_load_66(Occupant_load):-
    (exits(Exits); exit_access_doorways(Exits)),
    space(Space),
    occupant_load(Occupant_load),
    with(Space,Occupant_load),
    greater_than(Occupant_load, quantity(501,One)),
    not greater_than(Occupant_load, quantity(1000,One)),
    number_of(Exits, Number_of_exits),
    greater_than_or_equal_to(Number_of_exits, 3).

# Connected Logic Rule
compliance_of_Occupant_load_66(Occupant_load):-
    (exits(Exits); exit_access_doorways(Exits)),
    space(Space),
    occupant_load(Occupant_load),
    with(Space,Occupant_load),
    greater_than(Occupant_load, quantity(501,One)),
    not greater_than(Occupant_load, quantity(1000,One)),
    provided(Space, Exits),
    number_of(Exits, Number_of_exits),
    greater_than_or_equal_to(Number_of_exits, 3).
```

Fig. 5: *Rule 66 from the logic rules.*

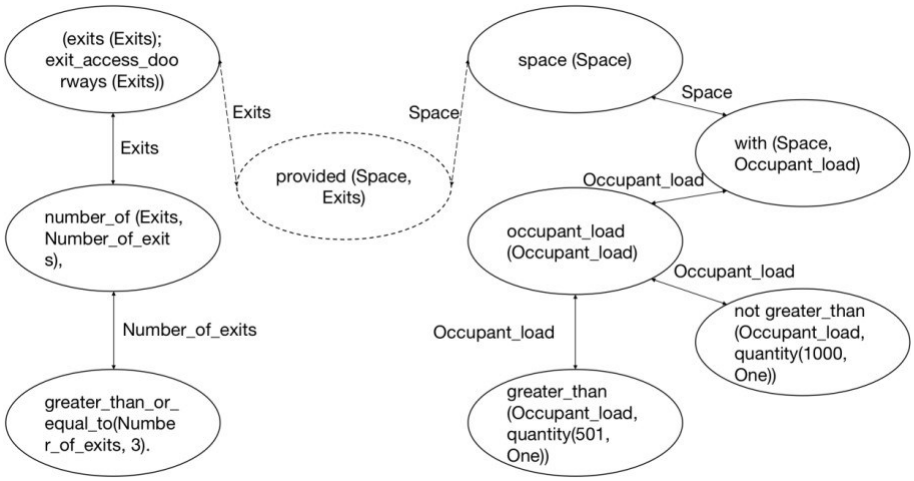

Fig. 6: *Graph representation of Rule 66 for validity check.*

4.4. Develop new activation conditions module

The authors developed activation conditions based on the identification of the target checking entity, which is usually the subject of a sentence. In the activation condition generation module, a refined logic rule was processed in four steps: (1) identify checking target, (2) identify checking pre-conditions, (3) generate activation clauses for pre-condition validation, and (4) generate activation clauses for results recording.

For example, for Rule No. 12 (Fig. 7) in the refined rules, "*Means_of_egress*" is the checking target. The checking pre-conditions are the "*means_of_egress(Means_of_egress)*", "*ceiling_height(Ceiling_height)*", and "*has(Means_of_egress, Ceiling_height)*". The activation conditions used the "*findall*" statement to check all egresses, and then filtered them by the pre-conditions. The qualified instances would be stored in a list *L*. Then for each instance in the list *L*, the rule was activated, and the filtered instances could be reported using the "*javeMethod*" to call the corresponding result recording function "*record*" in Java. The "*record*" function can record the result of compliance checking of the instance, with "1" representing true and "0" representing false.

```
# Logic rule No. 12
compliance_of_Ceiling_height_12(Ceiling_height):-
    means_of_egress(Means_of_egress),
    ceiling_height(Ceiling_height),
    has(Means_of_egress,Ceiling_height),
    not less_than(Ceiling_height, quantity(90, inches)).

# Activation conditions for No. 12
compliance_of_Ceiling_height_12_act(ACC):-
    findall(Means_of_egress,
            (means_of_egress(Means_of_egress),
             ceiling_height(Ceiling_height),
             has(Means_of_egress,Ceiling_height))),
            L),
    foreach(X in L,
            (compliance_of_Ceiling_height_12(X)
                -> javaMethod(ACC,record(X,1));
                javaMethod(ACC, record(X,0)))).
```

Fig. 7: *Logic Rule No. 12 with its original text and generated activation conditions.*

4.5. Develop model validation module

In this step, the authors developed the validation module to incorporate the model validation method (Wu and Zhang, 2022) into the I-SNACC system. For this step, Functions 6 to 9 shown in Fig. 2 were implemented. Function 6 "Select the design model to check" was developed using the Java file input and output packages (Oracle, 2020b), the user can select an IFC model to be validated. For the validation process, the system incorporated the developed model validation algorithm that processes IFC models into invariant signatures to match the following four categories of target concepts. For explicit concepts, the algorithm was able to generate logic facts directly based on invariant signatures. For inferable concepts, the algorithm was able to conduct inferences based on the invariant signatures and then generate logic facts. For user-assisted concepts, the algorithm would create simple multiple-choice questions for the user to input corresponding information. This function ensures the input from users is minimal. The interaction with users was achieved by a simple GUI with a textbox for output and a textbox for input (Function 8). For system defaults, the algorithm generated those clauses and set them to true by default, as they do not directly interact with compliance checking result, therefore this will not affect the validity of the test results. The authors incorporated and connected the model validation module and adapted it to Functions 6-9 of the I-SNACC system. With this module, logic facts can be generated to represent all needed building design information to check.

4.6. Expand and verify the rule execution module

The authors developed logic facts expansion algorithms for creating relations that connect concepts together. For example, "*lead_directly_to(Exterior_exit_doors, Exit_discharge)*" was a relation specifying that the exterior exit door must lead directly to the exit discharge, and this was implemented by checking if the egress was connected directly to the outside. Another example is the "*within(Door_opening, Dwelling_unit).*" This predicate was implemented by iterating all the dwelling units and checking if any door is inside or at the boundary of the unit. The checking is performed by analyzing the geometric and locational information using invariant signatures. With the expanded logic facts, the logic rules that represent requirements from building codes were ready to be executed. The predicate was generated using invariant signatures with the algorithms implemented in Java directly, as the values were already hashed in the memory in the previous module. The authors used a data-driven approach to expand relations and developed a total of 14 new relations. While each time there is a new relation predicate there needs to be such a development, those relations are reusable by different concepts and different chapters. As the development covers more training code chapters, the relational predicates will be accumulated and eventually become comprehensive to cover any codes.

4.7. Connect to compliance checking report generation module

After the rule execution module for applying the logic rules to the logic facts, the authors developed a new report generation module for reporting the compliance checking results.

Each generated report started with the description and statistics of the compliance checking. Then for each non-compliance case, the corresponding information is reported. For each non-compliance case, the system was able to display the violating entity, the violated logic rule, and the corresponding building code requirement. Fig. 8 shows an example of the report. The values of the violating object can be checked by observing its invariant signatures.

```
================================================
-----Running Step 11-----
================================================
System found 1 non-compliance
================================================
Non-compliace 1:
   Door 9 (1qPjXNL6r8rRF28rPA_nZA) and Rule 12
------------------------------------------------
Rule 12:
   compliance_of_Ceiling_height_12(Ceiling_height):-means_of_egress(Means_of_egress), ceiling_height(Ceiling_height),
has(Means_of_egress,Ceiling_height), not less_than(Ceiling_height, quantity(90, inches))
------------------------------------------------
Building Code:
   The means of egress shall have a ceiling height of not less than 7 feet 6 inches (2286 mm).
================================================
-----Finished Step 11-----
================================================
```

Fig. 8: *Example report presenting one non-compliance case.*

4.8. System testing and iterative refinement

The authors manually developed 60 non-compliance cases by adding or deleting objects and modifying the dimensions of objects in the training models, with 20 cases for each model. Modifying existing real-world projects provides more realistic cases than creating completely new models from scratch for non-compliance detection (Engels and Walz, 2018). During the iterative refinement of the system components, the system was able to achieve 100% precision and recall in non-compliance detection, improved from 51.7% precision and recall in the first iteration. Table 1 shows the corresponding results. Table 2 shows a few examples of the non-compliance cases detected. Compared to a manual development, the developing time was reduced from 24.23 minutes per non-compliance case to 13.22 minutes per non-compliance case (45.4% time saving), which is expected to further decrease as more code chapters are covered in further development. Compared to a manual compliance checking, the running time was reduced from 2.47 minutes per non-compliance case to 0.04 minutes per case (98.4% time saving).

Table 1. *Compliance checking results on training data*

Model	Non-compliance count (count without manual modifications)	Precision	Recall
Italian Restaurant	20 (11)	100% (55%)	100% (55%)
Warehouse	20 (7)	100% (35%)	100% (35%)
Convenience Store	20 (13)	100% (65%)	100% (65%)
Total	60 (31)	100% (51.7%)	100% (51.7%)

Table 2. *Example non-compliance cases detected*

Model	Example non-compliance	Description
Italian Restaurant	Rule 12	Compliance of ceiling height. Door 9 is less than 90 inches high.
Warehouse	Rule 294	Compliance of clear width. The width of door 4 is less than 32 inches.
Convenience Store	Rule 66	Compliance of occupant load. There are less than 3 (found 2) egresses for a space with an occupant load of 681.

5. RESULTS AND ANALYSIS

The system was successfully integrated and tested on the three training models. To validate the robustness of the system, the authors tested the system on two testing models, which were unseen and not used during the development phase. The precision and recall of those two models can reflect the robustness of the system. Comparing to 40 non-compliance cases in the testing models, the system was able to detect 42 non-compliance cases. Among the 42 detected non-compliance, 40 were correct. As a result, 95.2% precision and 100% recall were achieved on the testing data. Table 3 shows the detailed results.

Table 3. *Compliance checking results on testing data*

Model	Non-compliance count	Gold standard	Precision	Recall
Fast-food Restaurant	20	20	100%	100%
Hotel	22	20	91%	100%
Total	42	40	95.2%	100%

Our error analysis found that the error occurred because of an imperfection in the logic facts. In the hotel model, two interior doors were mistakenly treated as egresses, which caused the requirements for the egresses to be mistakenly applied to them. As a result, two false alarm non-compliance cases were reported.

There are at least two methods to fix the error. First, the error can be prevented by manually checking and modifying the logic facts prior to the final rule execution. Theoretically, all possible errors can be fixed manually by an expert in logic reasoning and compliance checking. While this approach is not desired, the current framework can still significantly reduce the manual efforts during that checking. The verification of logic facts is still much easier compared to generating them from scratch. As a result, the automation of the ACC is still significantly improved. Alternatively, the error could be fixed by perfecting the model validation module. The state-of-the-art model validation method can generate logic facts with an accuracy of 99% (Wu and Zhang, 2022). With 100% accuracy, such errors in checking can be avoided.

6. CONTRIBUTIONS TO THE BODY OF KNOWLEDGE

The authors proposed a new automated building code compliance checking framework, namely the I-SNACC system framework. First and foremost, this new framework promotes automation of ACC by developing a semi-automated approach to help validate the logic rules, for which the current SNACC system requires significant manual efforts. The developed I-SNACC system under this framework has a logic rule modification module with automated checking on grammar, syntax, format, and validity. Comparing to the state-of-the-art systems, this new

logic rule modification module can significantly reduce the needed manual efforts in the refinement of logic rules, by saving 45.4% of the time in rule refinement. Second, the new system incorporated a model validation module, which only requires minimal input from users about the building design models. The model validation module can make full use of the model's existing information and combine it with user input to generate extended logic facts needed for ACC, instead of manual conversion in the current SNACC system and other similar systems. Third, the authors refined and developed several other modules, such as a new activation conditions module, which can identify the entity being checked and use pre-conditions to help perform instance-level compliance checking with the rules. Last but not least, the system architecture integrated the state-of-the-art works and separated the functions from ACC modules under the new I-SNACC framework, so the system is easier to expand with more building codes. Each module and function of the system became relatively independent, to allow future improvements on each module. Most importantly, the authors demonstrated through an experiment that by integrating all advancements as mentioned above in our I-SNACC system, a 100% recall in non-compliance detection was achieved. Such contribution is critical to help bring fully automated building code compliance checking to practice.

7. CONCLUSIONS

The authors presented a new framework to develop an integrated system for ACC. The framework incorporated several computing modules, such as interactive rule modification module, and invariant signature-based model validation module. With the newly developed modules, more logic reasoning can be performed automatically. Under the proposed framework, an ACC system - the I-SNACC system was successfully implemented with ten functions designed for developer users and end-users. Each of the functions was implemented and tested to function as expected. For the testing data, the system achieved 95.2% precision and 100% recall in non-compliance detection. The high precision and recall showed that the new framework has great potential in producing a fully automated compliance checking system for all building codes in the future.

8. LIMITATIONS AND FUTURE WORK

The authors evaluated the implemented system on Chapter 10 of the International Building Codes 2015 (IBC, 2015). While the prototype system showed excellent results in this Chapter 10, which implied good potential on the other chapters, this system still needs to be further tested on more chapters.

Although the proposed framework could promote automation in building code compliance checking by saving 45.4% of manual rule refinement time, some of the development still involved manual efforts, such as in modifying logic rules (by developer users). This will be the case until the ruleset becomes saturated, which will still require more development and testing for an extended period of time in the foreseeable future. An ideal system should be able to automatically generate correct logic rules based on input building codes. In addition, the system still requires manual development of pattern matching-based rules in model validation to expand the range of checkable building code requirements. We do not intend the I-SNACC systems to have perfect automation coverage for now. However, 100% recall in non-compliance detection was achieved in the testing results. It is acknowledged that full automation still has a long way to go. The main barrier is machine's limited understanding of building codes, and the ambiguity of natural languages in general. Allowing the system to fully understand the building codes requires further development on the related NLP techniques. That is one of the goals of general artificial intelligence, which is beyond the scope of this paper but could be part of the envisioned future research.

ACKNOWLEDGEMENT

The authors would like to thank the National Science Foundation (NSF). This material is based on work supported by the NSF under Grant No. 1827733. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

Alferes J.J. (1994). Reasoning with logic programming. Springer, Berlin Heidelberg.

Apstex. (2018). IFC framework. <www.apstex.com> (Jul 29, 2021).

Beach T. H., Rezgui Y., Li H. and Kasim T. (2015). A rule-based semantic approach for automated regulatory compliance in the construction sector. *Expert Systems with Applications*, 42(12), 5219-5231.

Bird S., Klein E. and Loper E. (2009). Natural language processing with Python: analyzing text with the natural language toolkit. O'Reilly Media, Inc.

Brants T. (2000). TnT: a statistical part-of-speech tagger. *Proc., sixth conference on Applied natural language processing*, Association for Computational Linguistics, 224-231.

Brill E. (1992). A simple rule-based part of speech tagger. *Proc., third conference on Applied natural language processing*, Association for Computational Linguistics, 152-155.

Butte College. (2016). The eight parts of speech. <http://www.butte.edu/departments/cas/tipsheets/grammar/parts_of_speech.html>. (Sep 11st, 2019).

Cai X., Dong S. and Hu J. (2019). A deep learning model incorporating part of speech and self-matching attention for named entity recognition of Chinese electronic medical records. *BMC Medical Informatics and Decision Making*, 19(2), 65.

Dimyadi J. and Amor R. (2013). Automated building code compliance checking - where is it at? *Proc. 19th Int. CIB World Build. Congress*, Brisbane, Australia.

Dimyadi J., Clifton G., Spearpoint M. and Amor R. (2016a). Computerizing regulatory knowledge for building engineering design. *Journal of Computing in Civil Engineering*, C4016001.

Dimyadi J., Pauwels P. and Amor R. (2016b). Modelling and accessing regulatory knowledge for computer-assisted compliance audit. *ITcon, Special issue CIB W78 2015 Special track on Compliance Checking*, 21, 317-336.

Dimyadi J., Davies K., Fernando S. and Amor R. (2020). Computerising the New Zealand building code for automated compliance audit. *Proc., 6th New Zealand Built Environment Symposium*, New Zealand.

Ding L., Drogemuller R., Rosenman M., Marchant D. and Gero J. (2006). Automating code checking for building designs-DesignCheck. *Clients Driving Innovation: Moving Ideas into Practice*.

Eastman C., Lee J., Jeong Y. and Lee J. (2009). Automatic rule-based checking of building designs. *Automation in Construction*, 18(8), 1011-1033.

Engels, Anita, Walz, Kerstin 2018. Dealing with multi-perspectivity in real-world laboratories: Experiences from the transdisciplinary research project urban transformation. Gaia (Heidelberg, Germany), 2018, 27(S1).

Ghannad P., Lee Y., Dimyadi J. and Solihin W. (2019). Automated BIM data validation integrating open-standard schema with visual programming language. *Advanced Engineering Informatics*, 40, 14-28.

Giménez J. and Marquez L. (2004). Fast and accurate part-of-speech tagging: The SVM approach revisited. *Recent Advances in Natural Language Processing III*, 153-162.

Häußler M., Esser S. and Borrmann A. (2020). Code compliance checking of railway designs by integrating BIM, BPMN and DMN. *Automation in Construction*, 121, 103427.

Hjelseth E. and Nisbet N. (2011). Capturing normative constraints by use of the semantic mark-up RASE methodology. *Proc., CIB W78-W102 Conference*, 1-10.

İlal S. M. and Günaydın H. M. (2017). Computer representation of building codes for automated compliance checking. *Automation in Construction*, 82, 43-58.

International Code Council. (2015). International Building Code. International Code Council, Falls Church, VA.

International Code Council. (2000a). International Fire Code. International Code Council, Falls Church, VA.

International Code Council. (1996). International Mechanical Code. International Code Council, Falls Church, VA.

International Code Council. (1995). International Plumbing Code. Building Officials and Code Administrators International, Country Club Hills, IL.

International Code Council. (2000b). International Residential Code for One-and two-family Dwellings. International Code Council, Falls Church, VA.

Khemlani L. (2002). Solibri model checker. *Cadence*, Austin,32-34.

Kim H., Lee J.K., Shin J. and Choi, J. (2019). Visual language approach to representing KBimCode-based Korea building code sentences for automated rule checking. *Journal of Computational Design and Engineering*, 6(2), 143-148.

Li B., Schultz C., Dimyadi J. and Amor R. (2021). Defeasible reasoning for automated building code compliance checking. *Proc., ECPPM 2021-eWork and eBusiness in Architecture, Engineering and Construction: Proc., 13th European Conference on Product & Process Modelling (ECPPM 2021)*, 15-17 September 2021, Moscow, Russia, CRC Press, 229.

Lee Y.C., Ghannad P., Dimyadi J., Lee J.K., Solihin W. and Zhang J. (2020). A comparative analysis of five rule-based model checking platforms. *Proc., Construction Research Congress 2020,* ASCE, Reston, VA, USA, 1127-1136.

Max B. (2013). Logic programming with Prolog. Springer, London.

Martins J. P., and Monteiro A. (2013). LicA: A BIM based automated code-checking application for water distribution systems. *Automation in Construction*, 29, 12-23.

Nawari N. O. (2019). A generalized adaptive framework (GAF) for automating code compliance checking. *Buildings*, 9(4), 86.

Oracle. (2020a). Class ProcessBuilder. <https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html> (Jul 29, 2021).

Oracle. (2020b). Class JFileChooser. <https://docs.oracle.com/javase/7/docs/api/javax/swing/JFileChooser.html> (Jul 29, 2021).

Park S., Lee J.K. (2016). KBimCode-Based Applications for the Representation, Definition and Evaluation of Building Permit Rules. *Proc., 33rd International Symposium on Automation and Robotics in Construction (ISARC)*, International Association for Automation and Robotics in Construction (IAARC), 720-728.

Pauwels P., Van Deursen D., Verstraeten R., De Roo J., De Meyer, R. Van de Walle,R. and Van Campenhout J. (2011). A semantic rule checking environment for building performance checking. *Automation in Construction*, 20(5), 506-518.

Petrov S., Das D. and McDonald R. (2011). A universal part-of-speech tagset. arXiv preprint arXiv:1104.2086.

Pota M., Marulli F., Esposito M., De Pietro G. and Fujita H. (2019). Multilingual POS tagging by a composite deep architecture based on character-level features and on-the-fly enriched Word Embeddings. *Knowledge-Based Systems*, 164, 309-323.

Preidel C. and Borrmann A. (2018). BIM-based code compliance checking. Building information modeling: Technology foundations and industry practice. pp. 367–381. Springer Nature, Switzerland.

Shao Y., Hardmeier C., Tiedemann J. and Nivre J. (2017). Character-based joint segmentation and POS tagging for Chinese using bidirectional RNN-CRF. arXiv preprint arXiv:1704.01314.

Sing T. F. and Zhong Q. (2001). Construction and real estate NETwork (CORENET). *Facilities*, 19(11-12), 419-428.

Spivey J. M. (1996). An introduction to logic programming through Prolog. London, New York, Prentice Hall.

Sydora C., and Stroulia E. (2020). Rule-based compliance checking and generative design for building interiors using BIM. *Automation in Construction*, 120, 103368.

Tan X., Hammad A., and Fazio P. (2010). Automated code compliance checking for building envelope design. *Journal of Computing in Civil Engineering*, 24(2), 203-211.

Van Rossum, G. (2007). Python Programming Language. *Proc., USENIX annual technical conference, 36.*

Wu J., Sadraddin H.L., Ren R., Zhang J. and Shao X. (2021). Invariant signatures of architecture, engineering, and construction objects to support BIM interoperability between architectural design and structural analysis. *Journal of Construction Engineering and Management*, 147(1).

Wu J. and Zhang J. (2022). Model validation using invariant signatures and logic-based inference for automated building code compliance checking. *Journal of Computing in Civil Engineering*, 36(3), 04022002.

Xu X. and Cai H. (2020). Semantic approach to compliance checking of underground utilities. *Automation in Construction*, 109, 103006.

Xue X., Wu J. and Zhang J. (2022). Semi-automated generation of logic rules for tabular information in building codes to support automated code compliance checking. *Journal of Computing in Civil Engineering*, 36(1), 04021033.

Xue X. and Zhang J. (2022). Regulatory Information Transformation Ruleset Expansion to Support Automated Building Code Compliance Checking. *Automation in Construction*, 138, 104230.

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

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

Zhang J. and 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.

Zhang J. and El-Gohary N. (2017). Integrating semantic NLP and logic reasoning into a unified system for fully-automated code checking. *Automation in Construction*, 73, 45-57.

Zhong B., Ding L., Luo H., Zhou Y., Hu Y. and Hu H. (2012). Ontology-based semantic modeling of regulation constraint for automated construction quality compliance checking. *Automation in Construction*, 28, 58-70.

Zhou N.-F. (2014). B-prolog user's manual (version 8.1): Prolog, agent, and constraint programming. <http://www.picat-lang.org/bprolog/download/manual.pdf> (Jan 16, 2023).

Zhou P. and El-Gohary N. (2018). Automated matching of design information in BIM to regulatory information in energy codes. *Proc., Construction Research Congress 2018*, 75-85.