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

TOWARDS CODE COMPLIANCE CHECKING ON THE BASIS OF A VISUAL PROGRAMMING LANGUAGE

SUBMITTED: July 2016
REVISED: October 216
PUBLISHED: November 2016 at http://www.itcon.org/2016/25
GUEST EDITORS: Dimyadi J. & Solihin W.

Cornelius Preidel, MSc,
Chair of Computational Modeling and Simulation, Technical University of Munich;
cornelius.preidel@tum.de

André Borrmann, Prof. Dr.-Ing.,
Chair of Computational Modeling and Simulation, Technical University of Munich;
andre.borrmann@tum.de

SUMMARY: *In the AEC industry, there is a large number of standards and codes which ensure the structural stability, reliability, usability of the building under design. Accordingly, checking the conformity of the building design with these requirements is a crucial process. Nowadays this checking is performed to a large extent manually based on two-dimensional technical drawings and textual documents. Due to the low level of automation, the conventional checking procedure is laborious, cumbersome and error-prone. As Building Information Modeling (BIM) becomes more and more mature, a suitable digital information basis also becomes available to enable automating the process. The commercial solutions for code compliance checking available so far mainly follow a black-box approach where the rules that make up a certain regulation are implemented in a hard-wired fashion rendering their implementation in-transparent and non-extendable. A number of researchers have tackled this problem and have proposed various ways that allow the user to define rules, either in a standard programming language or in a dedicated language. However, AEC domain experts usually do not have the required programming skills to use these languages appropriately. To overcome this issue, we introduce the Visual Code Checking Language (VCCL), which uses a graphical notation in order to represent the rules of a code in a machine- and human-readable language. The paper presents the features and functionalities of the VCCL in detail and shows its application in a number of case studies.*

KEYWORDS: *Code Compliance Checking, Visual Programming Language, Building Information Modeling*

REFERENCE: *Cornelius Preidel, André Borrmann (2016). Towards code compliance checking on the basis of a visual programming language. Journal of Information Technology in Construction (ITcon), Special issue: CIB W78 2015 Special track on Compliance Checking, Vol. 21, pg. 402-421, http://www.itcon.org/2016/25*

1. INTRODUCTION

In the design and engineering of buildings, a large amount of building codes and regulations have to be taken into account. These codes and regulations serve not only to ensure that the building to be erected functions correctly, but particularly to guarantee the safety of its users. Today, the compliance of building designs with such regulations is checked to a large extent manually; both by the responsible planning consultant as well as the building authority officers.

The manual checking approach prevalent today is based primarily on construction plans (two-dimensional technical drawings) and additional textual documents. Since almost no automation support is available, the code compliance checking procedure is laborious, cumbersome and error-prone. In many cases, unwanted iteration cycles become necessary due to modifications demanded by the respective authorities. As a consequence, the checking of building code compliance is a major cause of delays and cost increases in construction planning (Preidel *et al.*, 2015).

A number of research projects around the world aim to undertake a major step forward through the development of computer-aided methods and technologies for automating extensive aspects of the relevant checking processes. With the aid of such tools, the manual effort required for code compliance checking can be drastically reduced, thus providing a key contribution to increasing the efficiency of building design and engineering processes.

The technological basis for the aspired automation in construction code checking is based on the concept of Building Information Modeling (BIM) which is currently comprehensively transforming the working processes of the architecture, engineering and construction (AEC) industry. While the advantages of BIM technology have been researched and documented for large parts of the AEC processes, its great potential for automating the building code compliance checking process has not yet been fully exploited.

The chosen methodological approach is based on the conception and development of a formal code representation language for encoding construction codes in a computer-interpretable form. In contrast to existing approaches, the proposed language is not a textual language, but a visual one based on a graphical notation with well-defined semantics. Given the success of visual programming languages in the domain of (algorithmic) architectural design (Anton and Tănase, 2016), the authors argue that a visual code checking language is much more accessible to AEC domain experts with limited programming skills. This claim is supported by more general investigations of the usability of visual programming languages (Catarci and Santucci, 1995).

2. STATE OF THE ART

So far, there have only been limited investigations into the use of digital building models for checking a building's compliance with codes and regulations. A good overview of the state of the art in this field as well as the most important open research questions is provided by Eastman *et al.* (2009). The authors identify a great need for research with respect to the separation of the representation of rules and the techniques required for processing and checking them. They state that "no language for building model rule checking is known to have been proposed".

In most of the known approaches, the rules have been implemented by software developers as procedural code embedded within the code checking system (Solihin, 2004; Ding *et al.*, 2006; Eastman *et al.*, 2009). In most cases the code is not accessible for third-party developers or domain experts – its correctness is thus not verifiable. Nisbet *et al.* (2008a) call this approach a "black box" implementation, which domain experts do usually not trust. In addition, modifications to the implemented code in response to changes to the regulatory documents can only be conducted by software experts.

An example of such a black-box implementation is the system *CORENET e-PlanCheck*, which is in use in Singapore for checking the compliance of digital building models with building codes in the areas of building control, barrier-free access and fire safety (Solihin, 2004). The main component of the code checking system is the FORNAX library which has been developed and maintained by a private company. As a consequence, the individual steps of checking process are not visible so that there is no or limited way for the users to alter it. Extensions and modifications have to be carried out by the original provider.

Similar limitations apply to the software product *Solibri Model Checker (SMC)* which provides code checking functionality within rule templates, which are based on hard-coded rules. Though these rule templates may be adjusted by a given limited number of parameters and composed by the user to define a certain individual checking process, they are nevertheless basically implemented as black-box procedures. Also, these rules cannot be represented by means of a human-readable external format but have to be implemented by means of a native data format. Nevertheless, SMC represents one of the most advanced representatives in terms of code checking since the application is exclusively based on the open IFC standard and enables advanced and detailed checking routines like accessibility checks according to the ISO DIS 21542 or ADA- and ABA guidelines (ISO, 2011; United States Access Board, 2014). For this reason, it has also been employed in a number of code checking research projects, among them the Norwegian project HITOS (Lê *et al.*, 2006), where the International Code for Accessibility Design (ICC/ANSI A117.1.) has been implemented by means of SMC rules. In a project led by the U.S. American General Service Administration (GSA), circulation and security rules for U.S. courthouses have been implemented by means of SMC (Eastman, 2009). In both cases, an external computer-interpretable representation of the rules, which would allow the usage of alternative code checking software and thus provide the desired degree of independence, has not been realized.

One of the first documented approaches for the representation of regulations in a system-independent computer-processable form was realized by Kerrigan and Law (2003) who applied First Order Predicate Logic for the formal expression of rules. However, they did not use a digital building model as input for the compliance check but instead gathered the required facts by means of an interactive system where the user has to answer a large number of questions. The gathered facts were subsequently used by a reasoning engine to formally check the compliance of the project with the US regulations for the protection of the environment. Rules with geometric or spatial semantics were not subject of the research project.

An important step towards the use of an independent rule representation has been conducted by the Australian code checking project *DesignCheck*, where the Australian code "Design for access and mobility" (AS 1428.1) has been implemented on the basis of the commercial system *EDModelChecker* (Ding *et al.*, 2006). In this project, the computer language EXPRESS (Wilson, 1988) has been employed for encoding the rules. However, as EXPRESS is a not widespread and hard-to-learn language that is not widely supported by software tools, the encoding of the rules can again only be realized and checked by trained software specialists.

The international project *SMARTCodes* led by the International Code Council (ICC) aimed to overcome this deficiency by introducing a two-step process where in the first step domain experts enrich the regulation texts written in human language with semantic mark-up, and in the second step the enriched text is semi-automatically transferred into a computer-processable form (Nisbet *et al.*, 2008a; Hjelseth and Nisbet, 2011; Hjelseth, 2012). For the first step, the well-known and widespread data modeling standard Extended Markup Language (XML) is applied, resulting in an easily verifiable and maintainable code representation. For the formalization of guidelines, SMARTCodes uses the RASE-Syntax. Using this tool, all elements that are used in a guideline can be categorized into the four different classes Requirement, Applicability, Selection und Exception. Doing so, even complex contents of codes and guidelines may be formalized and divided into these basic components. The result of this categorization can be illustrated and marked within the guidelines flow text and there-fore the result is readable for machines as well as humans. In principal, this method provides a practical and efficient approach. A major limitation of this approach is that only the content of a standard can be covered, but not expert knowledge, which results from the experience of the respective reviewer. Furthermore, the approach does not cover procedural knowledge; so how objects and information to which the guideline refers to, is produced in order to check the rule. Semantically higher predicates, such as topological or geometric relationships, are not represented.

An alternative direction of research for automated code compliance checking is based on the usage of an ontological description of both the building regulations and the digital building model to be checked. For example, in (Lange, 2008) the applicability of ontology-based knowledge representation and reasoning for checking the compliance of building models with the German fire codes has been investigated. The author comes to the conclusion that only a very small part of the regulations can be transferred into an ontology-based representation. One of the main limitations is the lack of support for semantically higher concepts, such as spatial relationships. Accordingly, the achieved compliance checking was restricted to simple rules, such as checking the fire resistance class of a certain wall. In (Kim and Grobler, 2009) ontological consistency checking mechanisms have been employed to check the conformance of a building design with constraints and requirements. Again, only very simple checks have been implemented referring to alphanumeric properties of building elements, such as a slab's thickness.

In the context of the research project C3R (Conformance Checking in Construction - Reasoning), conformance requirements have been transformed into queries formulated in the ontology query language SPARQL (FIG. 1) and subsequently applied to extract information from a given digital building model represented by an ontological representation (Yurchyshyna *et al.*, 2008; Yurchyshyna and Zarli, 2009). Also in this research, spatial relationships and properties have been completely neglected.

```
PREFIX ontoCC: <http://our_ontology.owl#>
SELECT ?portique display xml
where { ?portique rdf:type ontoCC:PortiqueSecurite
OPTIONAL
{ ?portique ontoCC:overallWidth ?width
FILTER ( xsd:integer(?width) >= 80) }
FILTER (! BOUND( ?width) ) }
```

FIG. 1 The SPARQL statement used in (Yurchyshyna and Zarli, 2009) for defining the constraint that the minimal width of a security door is 80cm. The example clearly illustrates the high complexity of the employed language.

For the encoding of more complex rules, (Lee, 2011; Lee *et al.*, 2015) developed the Building Environment and Analysis Language (BERA) and applied it for evaluating a building's circulation and its spatial programme. The language provides a set of spatial operators for the definition of rules in the context of these application areas. Though the developed language lacks the desired generality aimed at by the project proposed here, it forms a highly relevant foundation and an important point of departure.

As a part of the Korean KBim project, Park and Lee (2016) introduce KBimCode, a standardized scripting language for the representation, definition and evaluation of building permit rules. With this language, natural language rules can be translated into KBimCode through a logic rule-based mechanism. Since this is basically a programming language, the translation of the rules can again only be realized by software specialists.

In (Uhm *et al.*, 2015), request for proposals for large buildings in South Korea have been analyzed for the possibility to automate the compliance checking procedures. The study deployed a context-free grammar for processing the rules defined in natural language and translated them into computer-interpretable Semantic Web Rule Language (SWRL) rules.

Another important approach to checking Building Information Models for compliance with geometric rules was realized in (Zhang *et al.*, 2013), where safety rules for preventing fall-related hazards have been implemented. However, the rules are not represented in a neutral format but have been hard-coded into the BIM system.

In summary it can be stated that significant research has been conducted with respect to automating the code compliance checking for digital building models. However, in many cases, the rules contained in building regulations have been hard-coded into the code checking software, which results in a severe lack of transparency and flexibility of the encoded rule system. Only a small number of research projects have investigated the employment of a computer-processable intermediate representation of the building code.

In the majority of the work published in this regard, an ontology-based approach for the representation and checking of rules has been implemented. However, this ontology-based approach exhibits significant limitations: firstly, the encoding of rules into an ontological representation is too complicated to be directly performed by domain experts (see FIG. 1 for an example). Secondly, the ability of the resulting systems to represent semantically higher constructs is very restricted. In particular, abstract geometric and topological constraints and requirements can neither be represented nor checked, although they form an important part of many building codes and regulations.

In consequence there is a strong need for research to develop a code representation language which is (1) capable for the expression of geometric as well as non-geometric constraints and requirements, (2) accessible for domain experts with limited software development knowledge, and (3) independent from the employment of a particular code checking engine. This article aims at contributing to fill this important research gap.

3. VISUAL CODE COMPLIANCE LANGUAGE

As discussed in Section 2, there are a number of approaches to automate code compliance checking. However, there are still a lot of inadequacies, among them the inaccessibility of textual programming languages for domain experts. To overcome this deficit, we introduce a new approach here, which is based on a visual language for representing the Code Compliance Checking process. A first preliminary work on this approach has already been developed in (Preidel and Borrmann, 2015, 2016).

3.1 Methodological approach

In general, a visual language can be defined as a "formal language with a graphical notation", which means that it represents a modular system of signs and rules using visual elements instead of textual ones on the semantic and syntactic level (Myers, 1990; Hils, 1992; Schiffer, 1998). Information systems, which are described by a visual language can be interpreted much faster and easier by humans. Visual programming languages are often also called flow-based, since they display complex processing structures as a flow of information. The reason for the higher interpretation capability can be found in cognitive psychology, which states that visual information can be processed with two instead of only one hemisphere of the human brain in parallel. Schiffer (1998) performs a detailed discussion of the advantages and disadvantages of visual languages.

In recent years, Visual Programming Languages (VPL), have been established particularly in the field of building design creation. Known software products in the context of building design are in particular the plug-in Grasshopper for Rhinoceros3D, Dynamo for Autodesk Revit (or as a standalone application) and Marionette for Vectorworks. Although these representatives focused initially especially on the 3D parametric modeling, they were significantly extended by further functionalities since there is a huge third party community supporting these projects. However, since the existing VPL systems in AEC industry focus on tasks like architectural and geometric modeling, the language is designed to handle intuitive design tasks. In order to give the user as much freedom as possible for these tasks, the VPL systems usually have a low error tolerance, so that these systems usually lack rigidness and strictness. As an example, many VPL systems allow to define generic (= untyped) data transfer, which can lead to manifold errors. In contrast, the code checking process demands a high level of accuracy and correctness since it represents an essential process as discussed in Section 1. If the accuracy is not given, the users might lose the trust in the software product and the acceptance of the automation is significantly decreased. Nisbet *et al.* (2008b) state, that this acceptance is a key criterion for the successful introduction of an Automated Code Compliance Checking. Furthermore, the existing VPL systems like Dynamo (Autodesk) or Marionette (Vectorworks) are geared towards working closely with corresponding native BIM authoring tools and therefore do not themselves represent open software solutions in the sense of an openBIM approach. However, since the model as well as Code Compliance Checking is a central task that affects several disciplines and models at the same time, the checking must be applicable for any kind of model. Therefore, it makes sense to pursue a technically neutral approach, which is able to work with information of an open standard, e.g. IFC. To meet the above requirements, the authors do not extend an existing VPL system but build a new visual language design from scratch.

By adapting a visual language to the specific needs and requirements of Code Compliance Checking, one of the main shortcomings of the existing approaches to automated code compliance checking – the inaccessibility of the rule definition to domain experts - can be overcome. The developed approach focusses in particular on the human-machine-communication, which represents a mandatory requirement for the success of automating Code Compliance Checking. At any time and degree of completion of the visual processing system, the user is able to understand and inspect every single processing step, which is particularly important, since the accuracy of the results of the compliance checks are in the responsibility of the reviewer. If errors are identified in the processing chain, the system can be adjusted very quickly and easily according to the user's requirements. With the help of such a visual language, it is possible to describe most parts of prescriptive compliance checks without sacrificing the transparency for the user. So it makes a lot of sense to speak of a semi-automatic process as it is inevitable to involve the user into the checking process.

The VCCL implements two basic principles: genericity and finest granularity. The genericity describes the property of the VCCL that all elements must be defined as generic as possible regardless of the level of complexity. As a result, each element can be used in any situation and on any point of a desired code checking process. At the same time, it must be possible to break down each element to its lowest level. This property of the VCCL is called finest granularity. These two features cause a maximum of flexibility for the user, who can formulate the desired content. The principle behind this approach is to make the overall process of compliance checking both transparent and flexible, by allowing the user to compose the overall checking procedure from individual process steps. Each of these elements is a single white box, which can be considered as a small module of the whole process. To this end, we introduce a modular principle (see Section 3.2), which can be used by any user even without profound programming skills. In this way we allow that any engineer can bring his professional skills and his experience into the process. The VCCL is not only intended to store the contents, which are represented within codes or guidelines, but also to cover the procedural knowledge. This means, that the VCCL describes, how information must be processed in order to describe any kind of content, which is a part of a guideline. In this sense, the VCCL represents a far more advanced approach compared to mechanisms, which cover only the guideline knowledge.

3.2 Basic elements of the VCCL

To define the language both fundamental aspects, semantics and syntax, need to be specified. In the following sections, the elements of VCCL and their graphical representation are presented.

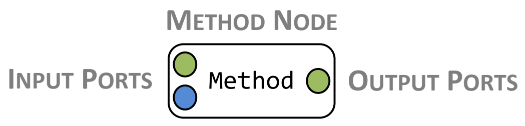

FIG. 2: Representation of a VCCL method node

VCCL is a strongly typed, object-oriented language. On the semantic level, the VCCL provides two different elements: methods and ports. Within a VCCL system data processing steps are realized by means of methods. To represent these methods in a graphical notation, the VCCL provides method nodes visualized by a rectangle with rounded corners (FIG. 2). Such a node describes a well-defined operation on a specified number of input variables, the operands, and generates a corresponding result.

The method node provides dedicated input ports (left end) and output ports (right end) which allow to connect other VCCL elements to it. These ports represent data objects of specific data types, which are handled within a VCCL program, and are visualized as colored circles. To each port a well-defined data type is assigned, meaning that only ports of the defined type can be connected to it. In this way, the data processing within a VCCL program is tightly controlled and the accuracy is increased since information cannot be misled. To improve the clarity of the overall program and support the user in terms of understanding the processing of the information, the ports are visualized with a specific colour representing the corresponding data type. Furthermore, any port may also be visualized with an extended badge, which illustrates and specifies the referred data type. A selection of ports of different data types as well as the corresponding badges are shown in FIG.3. In order to provide the necessary flexibility to meet the demands of a wide range of code checking scenarios, the type system of VCCL is extendable. VCCL provides currently a large set of pre-defined object types in its type library. This includes types representing atomic values (String, Boolean, Float) as well as relations or collections. A collection such as a set of a certain data type is visualized as a double circle (see FIG. 3).

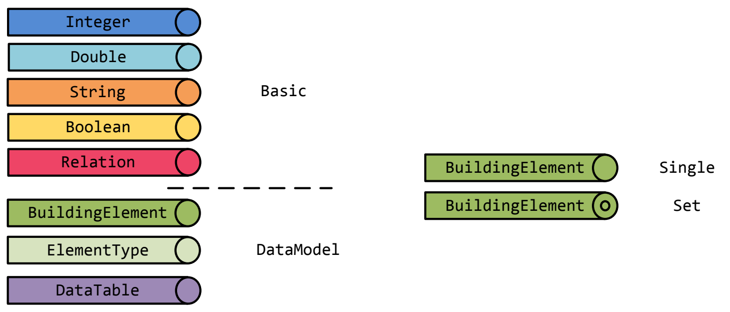

FIG. 3: left: Selection of different ports with badges representing different data types; right: Representation of a single and a set instance

To build up a processing chain, methods are connected through their port by means of directed edges. An edge forwards information from the output port of the source node to the input port of target node. The forwarded information can only be transferred in one direction, so that the transmission is unambiguously. Since the edges may only connect an output (right) with an input port (left), the resulting overall program can be interpreted as a flow of information from the left to the right. For the creation of a VCCL graph, the user is provided with a three-part control, which is divided into an input and output as well as a process environment. To define a distinct starting and ending point of each VCCL program, the global input and output area have a number of user-defined ports, which serve as initial information source and resulting ending point for the defined routine. When creating a VCCL program, the user has to define which type of information is provided as the initial input and which kind of information is produced as the final result of the program. The different parts of a VCCL program, as well as an exemplary creation of a generic VCCL program is shown in FIG. 4.

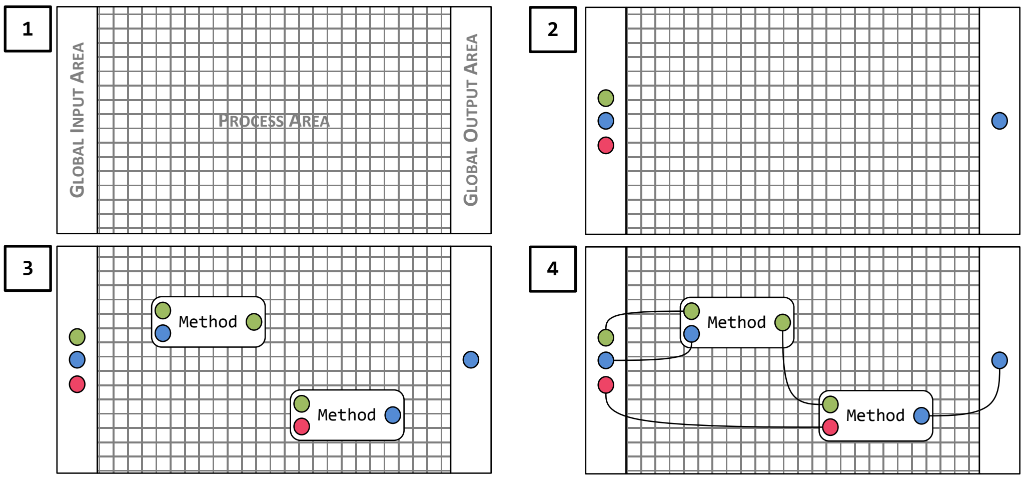

FIG. 4: Creating a generic VCCL program with the basic control

In order to ensure the robustness of the created VCCL programs, they must be validated. Each VCCL program is inherently verifiable as it follows the principle of information flow. Starting with the initial given information - represented by the global input ports – the information flows through the VCCL program, passing information from port to port via the directed edges. The information flow principle is schematically shown for a generic valid program in FIG. 5.

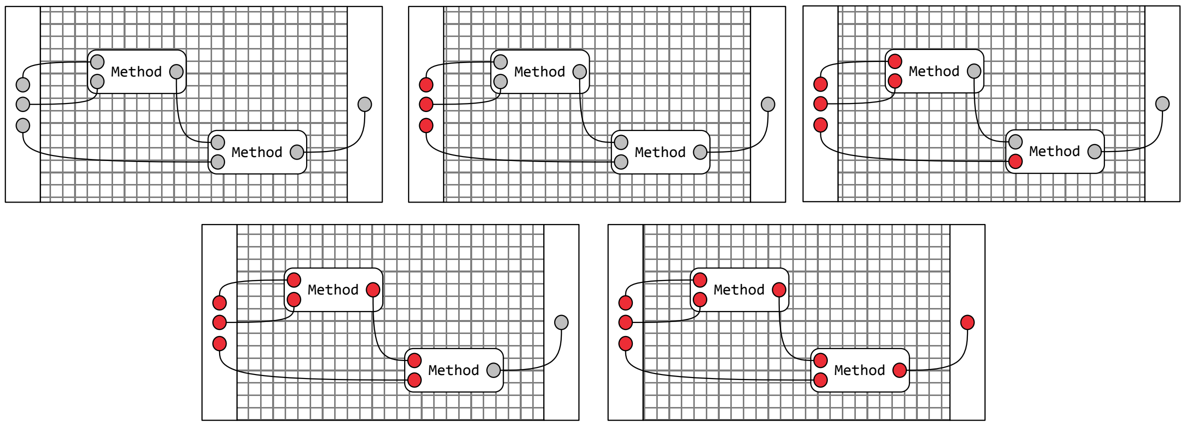

FIG. 5: Processing of information within a schematic VCCL programs based on the flooding principle

As described in Section 3.1, the VCCL follows the principle of realizing the finest processing granularity possible. This means, that the user must be able to have an insight in every single part of the processing chain of a VCCL program. To enable this, but also to avoid confusingly large VCCL networks, we introduce a nesting approach which makes use of the global input and output areas.

Since a VCCL method describes an operation, it can also be displayed as a separate VCCL graph and therefore be shown as a nested control. The concept can be compared to functions or subroutines in textual programming languages. In FIG. 6 the basic principle of such a nested VCCL program is shown.

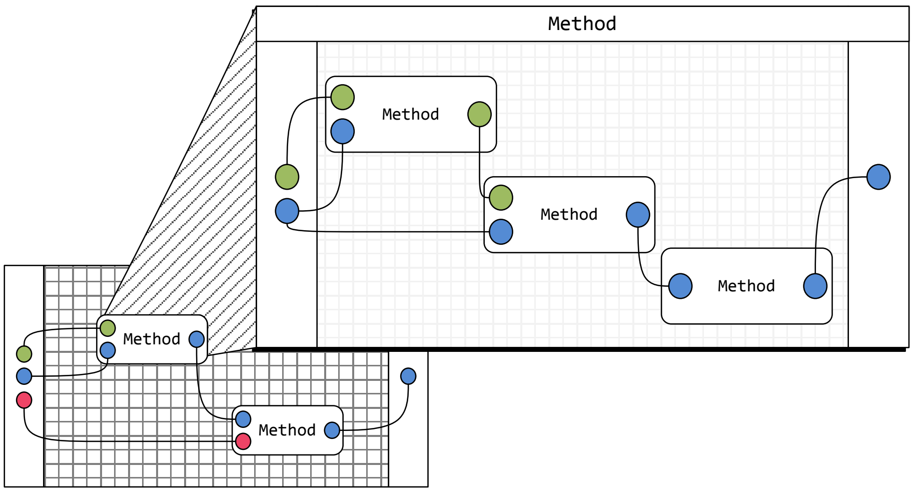

FIG. 6: Schematic illustration of a nested method within a generic VCCL program

A nested VCCL program realizes a number of well-defined processing steps which are hidden on the upper graph level in order to reduce the complexity of the overall VCCL graph. The nested program has clearly defined input and output ports and can be re-used in any given VCCL program. The input and output ports of the affected method node define the ports of the global input and output area. Accordingly, the user is able to built-up a hierarchic VCCL library which fits the particular needs of the code checking procedures he is confronted with. Based on this principle VCCL programs can be composed of basic methods (and corresponding ports), which are provided in a basic method library. The resulting hierarchic structure of the VCCL elements is shown in FIG. 7.

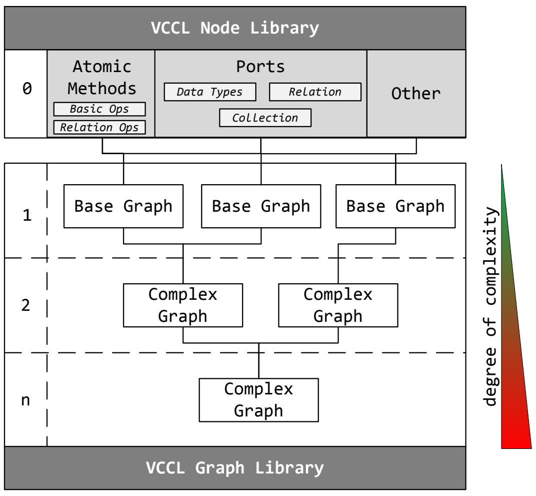

FIG. 7: Schematic illustration of the VCCL node library and its resultant VCCL graph library with its ascending degrees of complexity

To realize this approach, the basic methods must be defined beforehand. As a starting point, the VCCL provides methods which are given as basic routines. These methods describe fundamental operations in a VCCL program, which describe fundamental operations, whose semantics are unambiguously well defined. Therefore, these methods are called atomic methods. As their internal structure of the atomic methods is invisible to the user, they represent black-boxes. However, we assume that a certain "black-box level" is acceptable for a given application area, where deeper control and insight is neither desired nor necessary and too much effort would be created for implementing the provided functionality using VCCL instead of a standard programming language. A good example is the evaluation of geometric information such as the computation of a shortest distance route for a given floor plan. The computation is very sophisticated and the end user is most likely not interested in having insight in this routine. So this computation can be given as an atomic method and reused for different applications like fire escape routing or process distance computation. One important characteristic of such atomic methods is the genericity so that the methods can be used for different purposes. Certainly different black box limitations have to be considered for different applications and different levels of experience.

There is a large set of pre-defined atomic methods in VCCL. These include:

- Logical Methods
  - comparison operators comparing two values and returning a Boolean value, including *Equal, Unequal, LargerThan, SmallerThan*, etc.
  - operators combining Boolean values, such as *And, Or, Not, Xor, IsNotDefined*
- Mathematical Methods
  - basic calculation operators: *Plus, Minus, Product, Division*
  - set operators: *Union, Intersection*, *Complement, Difference*
- Geometric-Topological Methods
  - spatial predicates returning a Boolean value including topological predicates (*Equal, Disjoint, Touch, Overlap, Contains, Within*), geometric predicates (*CloserThan, FartherThan, ...*) and directional predicates (*Above, Below,* etc.). Further details are provided in (Borrmann et al. 2009; Borrmann and Rank 2009a; Borrmann and Rank 2009b; Daum and Borrmann 2013)
  - geometric operations generating geometric objects from existing ones, including *ConvexHull, Sceleton, etc.*
  - geometric evaluation operators: *ShortestDistance*, *MaximalDistance, ...*
- Relational Methods
  - operators for the Relational Algebra: *Projection*, *Selection*, *Join, ...*
  - aggregation operators: compute values in the course of loops, includes *Sum*, *Average, Min, Max, Count*
- Building Model Related Methods:
  - operators for the selection of building elements: *TypeFilter*, *Filter*, *Selection*
  - operators for accessing and retrieving attribute data: *GetProperty*, *GetRelated*
- Utility Methods:
  - By connecting an output port with a utility method node, the user is able to show alpha-numeric or visualize geometric contents. In this way input data, intermediate results, or final results can be presented and the content can be checked by the user.

To avoid overly complex VCCL programs, method nodes can include embedded UI controls designed to facilitate the selection of different options for the user, thus reducing the complexity of the overall program. Due to this embedding, no additional user inputs have to be mapped via global input ports and the number of edges is also significantly reduced. A selection of various atomic methods with embedded UI controls are shown in FIG. 8.

In most codes and regulations, tables represent a major source of information. These tables usually provide a value for a number of input values. To take this into account, VCCL provides the *Data Table* node as atomic method. Accordingly, the corresponding method is able to query any kind of data table regarding the given input variables. An exemplary application of these data table elements is described in Section 4.

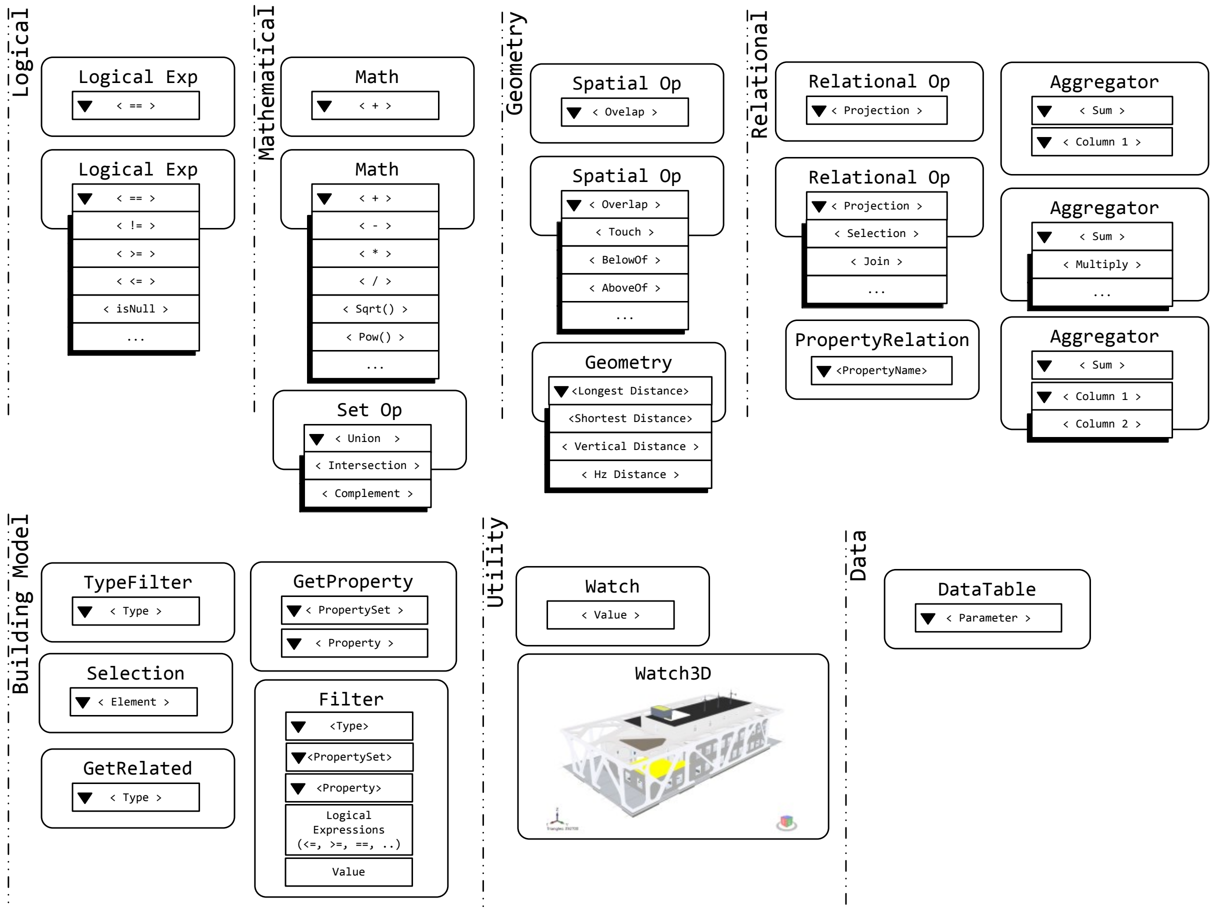

FIG. 8: Selection of atomic methods available within the VCCL

3.3 Handling of relations

The mathematical concept of relations has shown to be extremely useful in the context of algorithmic code compliance checking. Formally, a relation is defined as the subset of the Cartesian product of a given number of sets. More vividly, a relation can be described as a set of n-tuples where each tuple combines those objects that are in a certain relation with each other. The relational algebra defined by (Codd, 1970; Codd, 1991) allows to operate on, process and analyse a set of relations. The concept is well-known from relational databases where the theory of relational algebra has been very successfully implemented in the widespread declarative query language SQL (FIG. 9).

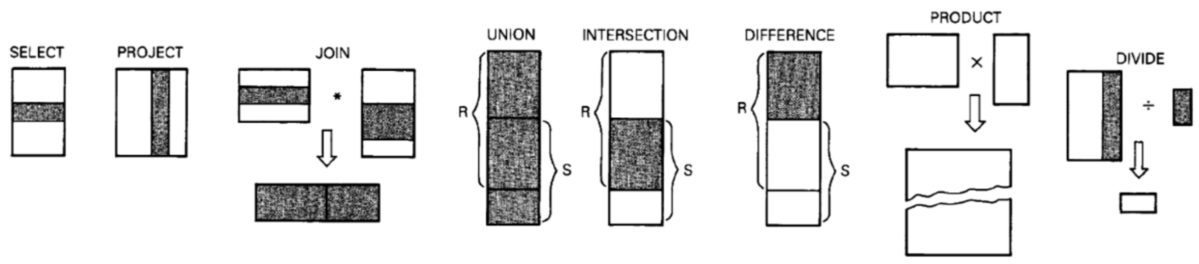

FIG. 9: Base Operators of the Relational Algebra, according to (Codd, 1991)

The VCCL integrates the possibility to work with relations by providing the basic data type *Relation* and providing the operators of the relational algebra (see Section 3.2) as dedicated operator nodes of the VCCL (Preidel and Borrmann, 2016). In this respect it has to be noted that, by contrast to SQL which is declarative language implementing the relational algebra, VCCL provides an imperative implementation of the relational algebra, i.e. the operators are applied in a procedural manner. This however, does not at all restrict the applicability of the relational algebra.

Various methods within the VCCL produce relational output results. Usually such a relation is created out of at least two input objects by checking if given criteria of the elements of these input sets are met. If the criteria are fulfilled, the elements have a relationship and are added to the relation as an n-tuple (pair, triple, etc.). A simple example for the mechanism of such a relation creation is shown in FIG. 10. Here the geometric-topological method *isInside* identifies those pairs of Wall and Opening objects where the geometric relationship "is inside" is given.

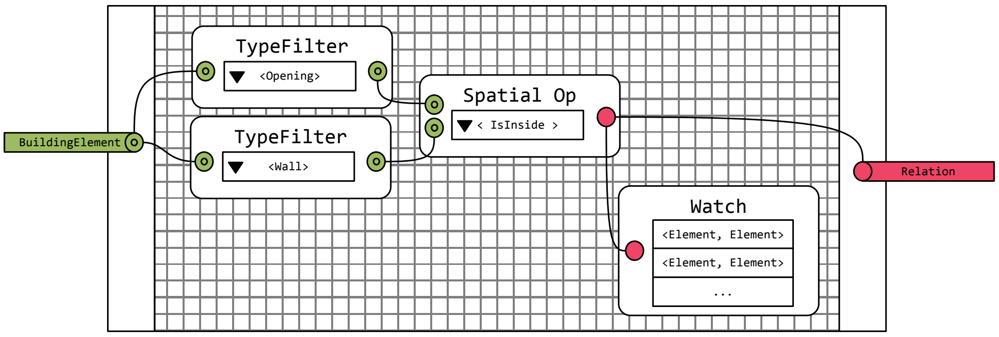

FIG. 10: Creation of a relational data object by a geometric-topological evaluation

The necessary checking processes of such a method must be implemented by means of a dedicated algorithm, which is typically formulated using a textual notation. In the presented example, the creator checks which building elements of the input sets are related by the criterion *isInside*. Depending on the underlying data model, this information can either be directly queried from the data model or it has to be processed by (e.g. geometric) algorithms.

FIG. 11 shows a more complex example where two relations have been created and subsequently combined using relational operators to identify pairs of Walls and Windows/Doors that are related to each other. The resulting relation can be used for further processing.

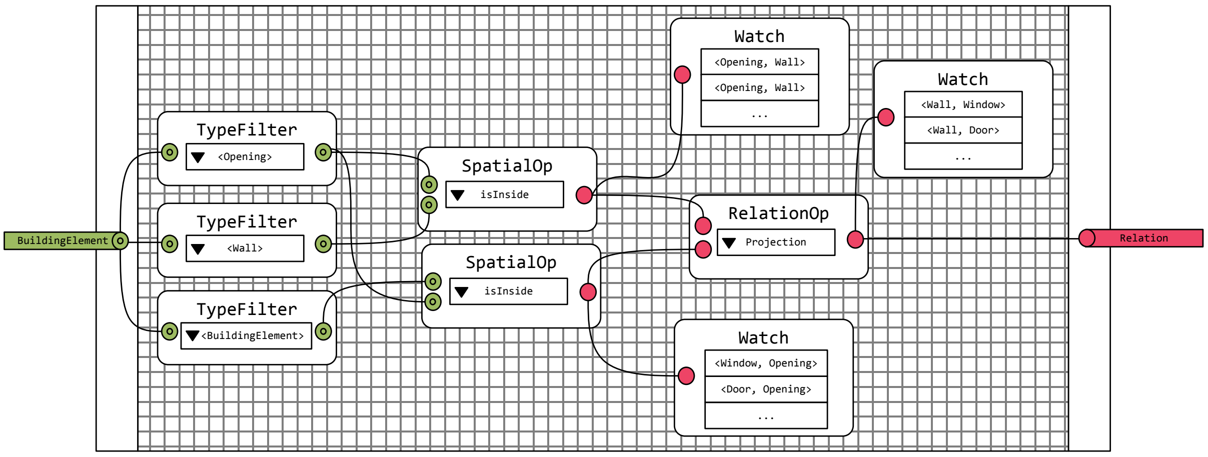

FIG. 11: Exemplary application of the relational methods for the evaluation of topological relationships

3.4 Control flow

The demands of more complex code compliance checking procedures requires VCCL to provide elements of control flow. Most of the existing imperative programming languages provide at least the most rudimentary elements of control flow – conditional branching and iterations. In textual programming languages, conditional branching is typically realized by means of *if* statements while iterations are implemented using *loops*. These constructs have shown to be very effective, easy to understand and sufficient for many application scenarios

(Böhm and Jacopini, 1966). Accordingly, VCCL provides these two control flow elements using a graphical notation.

These flow controls are basically implemented as derivations of nested methods. In this way, each VCCL program can be tested for validity separately. Special control methods would lead to the use of multiple output ports and therefore ambiguous programs, which can lead to invalid VCCL programs. FIG. 12 depicts the notation of an If-Else node. For this node, two nested VCCL subprograms are defined, of which finally only a single one is decisive depending on the result of a test case. So the prerequisite for such an If-Else node is the Boolean result (*True* or *False*) of a check which is evaluated before. If the Boolean value is evaluated as *True*, the corresponding area, which is defined for this case, is executed. If it is evaluated as *False*, the other sub-program is executed. In order to re-align the control flow after the conditional branch, it is necessary that the global input and output ports are of the same type for both sub-programs. At least one case of the logical expression (*True* or *False*) must defined – the definition of the second one is optional and can be left blank.

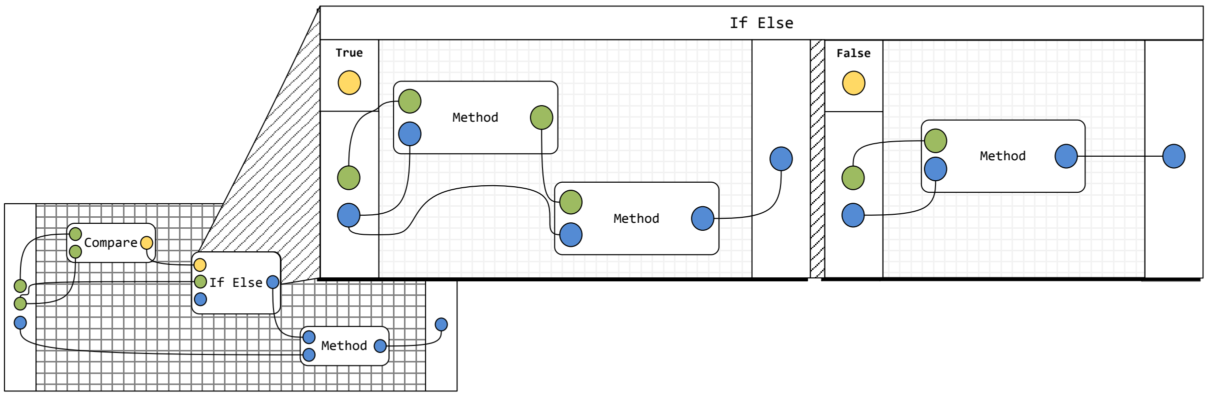

FIG. 12: Example showing the graphical notation for an If-Else node

FIG. 13 depicts the notation of the iteration element, the Foreach node. This node is used if a certain process is applied for each element of a collection. Therefore, the first input port must be multi-dimensional. The nested subprogram in turn defines a process for a single instance of the same data type, so that this process can be performed for each element, which is part of the input set. Since the nested subprogram produces a result for each element, the result of this method node is a relation, which holds a tuple with each element and the corresponding result. In this way, the results can be assigned to the respective values later on.

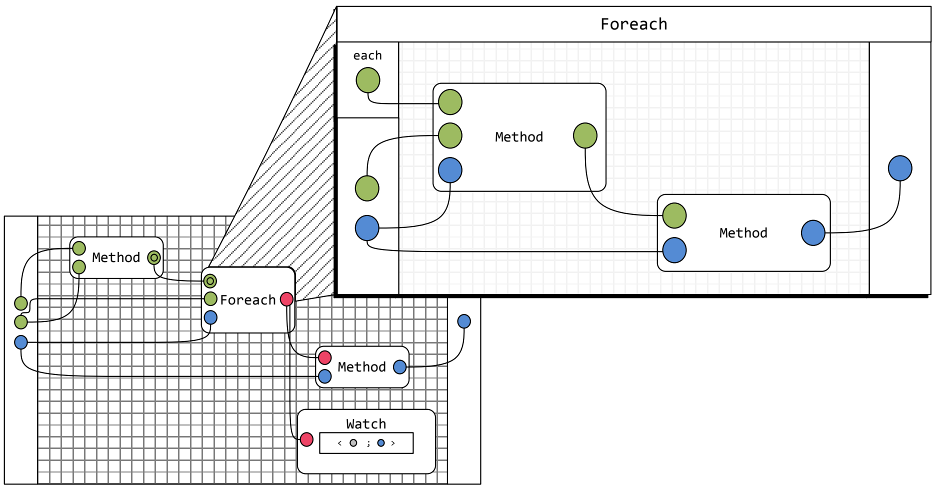

FIG. 13: Example showing the graphical notation for an Foreach node

A simple exemplary application of both VCCL control elements is shown in FIG. 14. In this example, each element within a set of walls is checked for a specific height. If the wall fulfils the criterion, which is described by the logical expression within the nested Foreach statement, it is added to the resulting relation. In this case the *Else* case is blank, since the object is not forwarded if the criterion is not fulfilled.

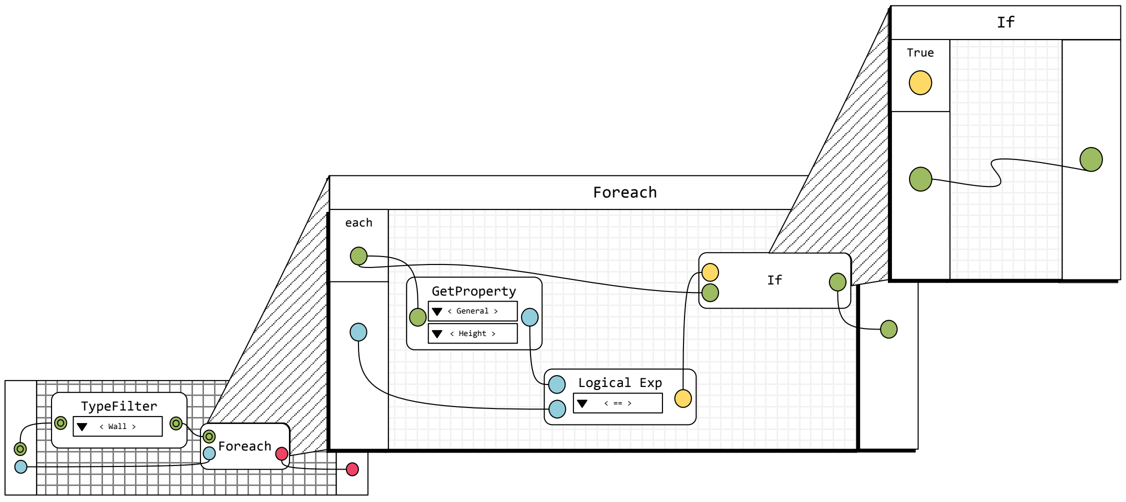

FIG. 14: Exemplary application of the VCCL control elements

4. APPLICABILITY

4.1 German fire code DIN 18232-2

In order to demonstrate the potential and versatility of the VCCL approach, a semi-automated compliance check for the German standard DIN 18232-2 (DIN, 2007) is shown. The standard addresses the design of buildings in terms of smoke and fire protection: Depending on the height of the room, the height of the smoke layer caused by the fire and the strength of the fire, the code defines a minimal smoke ventilation area (FIG. 15).

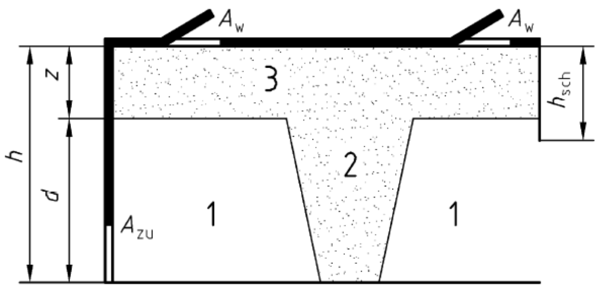

FIG. 15: Illustration of the smoke distribution created by a fire according to the German standard DIN 18232-11:2007 (DIN, 2007). Depending on the height of the room h, the height of the smoke layer z and the strength of the fire, the code requires a minimal smoke escape area A_w.

The specific input and output values are provided by means of a table, which is shown in FIG. 16.

Room Height [m]	Height of the Smoke Layer [m]	Required Ventilation Area [m²]				
		Fire Classification				
		1	2	3	4	5
3	0,5	4,8	6,2	8,2	11	15,4
3,5	1	3,4	4,4	5,8	7,8	10,9
	0,5	3	8,7	11,3	15	20,4
4	1,5	2,5	3,6	4,7	6,4	8,9
	1	3	6,2	8	10,6	14,4
4,5	2	2,5	3,1	4,1	5,5	7,7
	1,5	3	5	6,5	8,7	11,8
	1	3,5	8,4	10,7	13,9	18,6

FIG. 16: Excerpt of the data table for the required smoke ventilation area in m² (DIN, 2007)

The encoding of this rule as a VCCL graph is depicted in FIG. 17 In this processing graph a single room is identified and afterwards its attributes are used to capture the target as well as the actual area. The final step of this checking procedure is the comparison of these values - to check whether the limit value is met or not. The result is a corresponding Boolean value, which can be used afterwards for the further processing of the checking results such as assigning the requirements which are not fulfilled to a responsible project stakeholder.

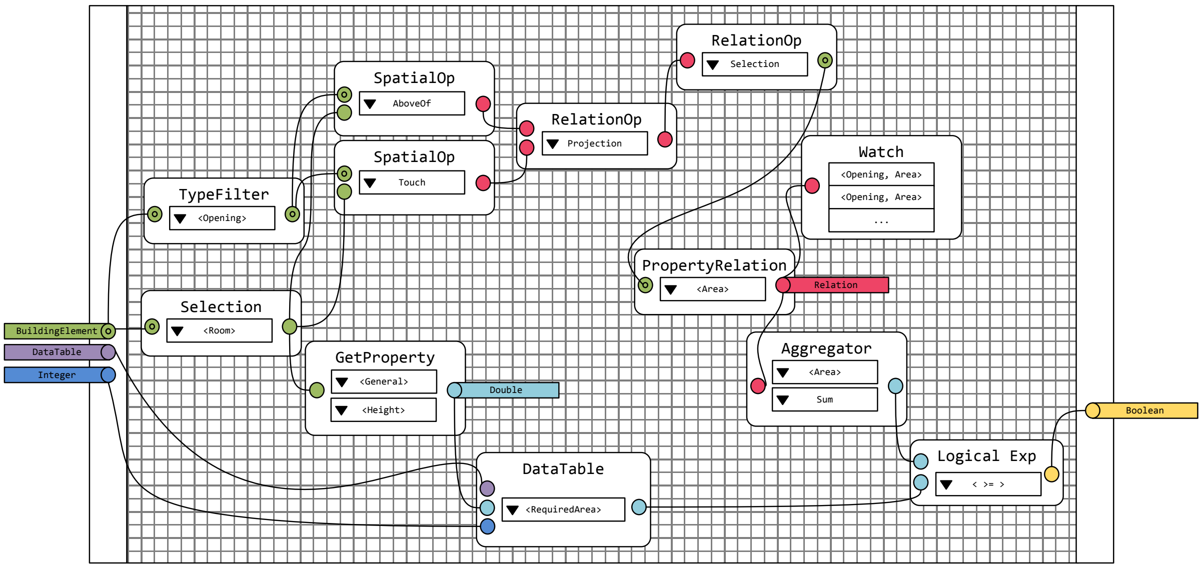

FIG. 17: VCCL program encoding the smoke ventilation area rules of the DIN 18232-2:2007-11

4.2 Korean Building Act: Article 34 Clause 1

Another application the VCCL was applied for the translation of Article 34 Clause 1 of the Korean Building Act (Korea Legislation Research Institute, 2008), which is shown in FIG. 18.

"On each floor of a building, direct stairs leading to the shelter floor or the ground other than the shelter floor shall be installed in the way that the walking distance from each part of the living room to the stairs is not more than 30 meters: Provided, that in cases of a building of which main structural part is made of a fireproof structure or non-combustible materials, the walking distance of not more than 50 meters may be established, and in cases of a factory prescribed by Ordinance of the Ministry of Land, Infrastructure and Transport, which is equipped with automatic fire extinguishers, such as sprinklers, in an automated production facility, the walking distance of not more than 75 meters may be established."

FIG. 18: Content of Article 34 Clause 1 of the Korean Building Act (Korea Legislation Research Institute, 2008)

Since the regulation is quite complex, it can be divided into different parts with the help of the VCCL. Afterwards the single VCCL programs can be represented as nested modules, so that a perpetual clearness can be guaranteed for the user.

First, all exit staircases must be identified for each floor. Therefore, the staircases (rooms, which have stairs inside) have to be checked, if they can be used as exits (rooms, which have exit doors). Subsequently, a single floor can be checked, if it is connected with such an exit staircase. The encoded VCCL graph is shown in FIG. 19.

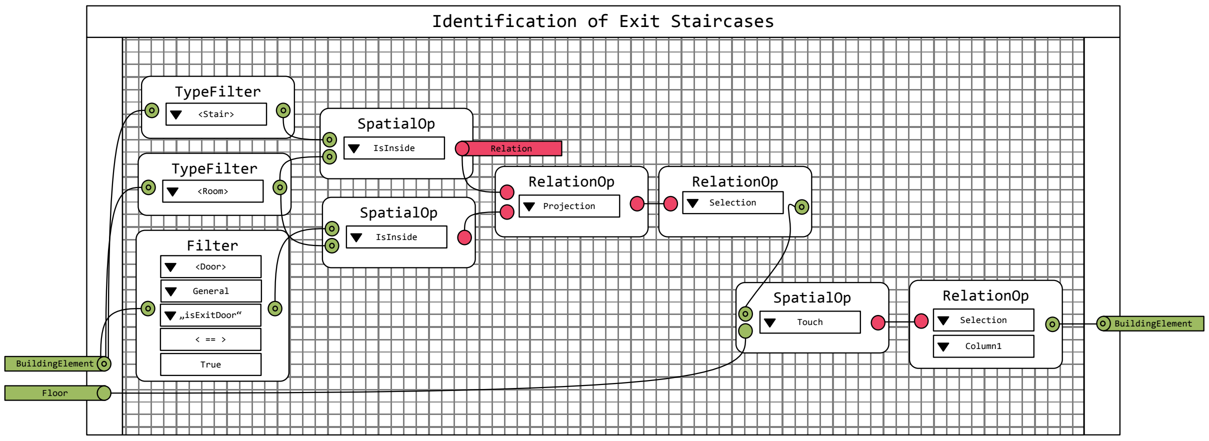

FIG. 19: Identification of exit staircases for a selected floor with the VCCL

As described in Section 3.2, we assume, that there is a black box level in each checking domain that describes the user's acceptance of working with not-accessible methods. In the present case the computation of the walking distance represents a complex process, which the user most likely has no interest to have an insight. Therefore, we assume at this point, that this evaluation is given as a method, which calculates the maximum walking distance from the floor plan and a respective target point.

FIG. 20 shows the final composed VCCL translation of the guideline. In this program the already presented VCCL method for the identification of the stairways as well as the atomic walking distance method is reused. In the present case, we assume, that the indication whether the main structure of the building can be classified as fireproof or is equipped with automatic sprinklers highly depends on the way this information is modelled in the building information model. So this information can be obtained either as a user input or evaluated in a higher VCCL program from the information of a building model. Both values are introduced as input variables for the VCCL program. In order to set the relevant value for the maximum walking distance, the program uses *If* elements. Finally, the relevant maximum value can be determined using simple mathematical methods.

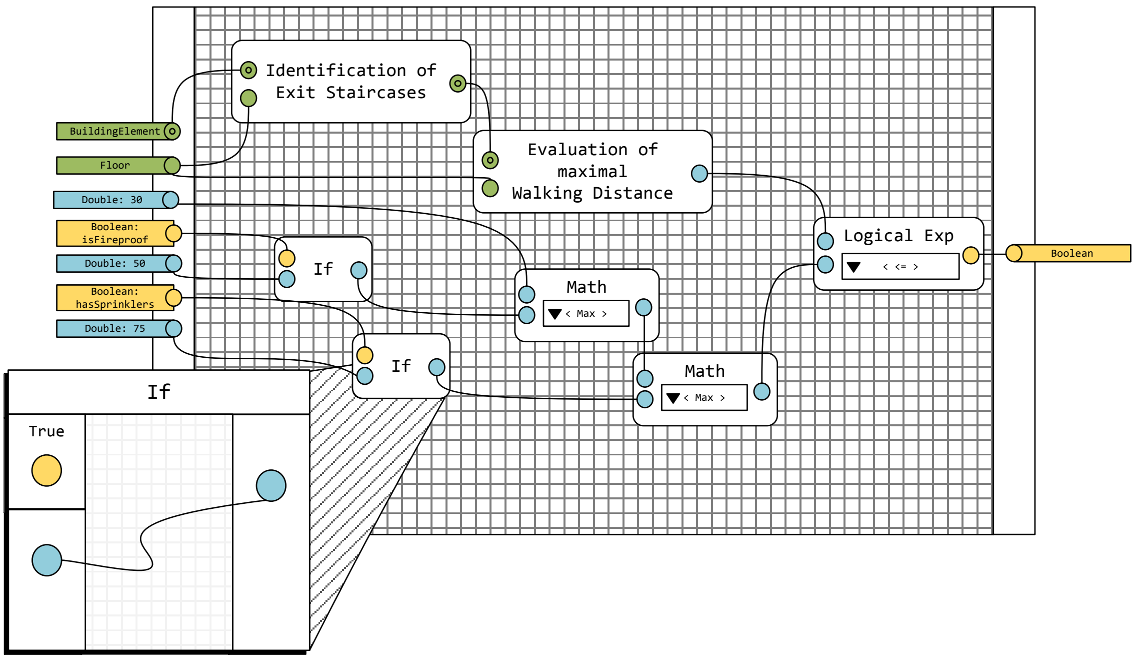

FIG. 20: Encoded VCCL representation of Article 34 Clause 1 of the Korean Building Act

5. PROTOTYPE IMPLEMENTATION

The application of the VCCL was developed and designed in close cooperation with the German software company Nemetschek Group. The VCCL was implemented within a prototypical standalone application, which is closely connected with bim+ (Allplan GmbH, 2016), a central platform for managing of building information models (FIG. 21). Next to a large number of basic functionalities, such as a web project management and a web viewer, the platform provides an open API, which enables developers to use the existing functionalities for their own purposes. The platform follows the openBIM approach and therefore building model data can be uploaded in various data formats including IFC files. The information is mapped onto a native internal data model, which is closely related to the IFC standard and well documented (bim+, 2016).

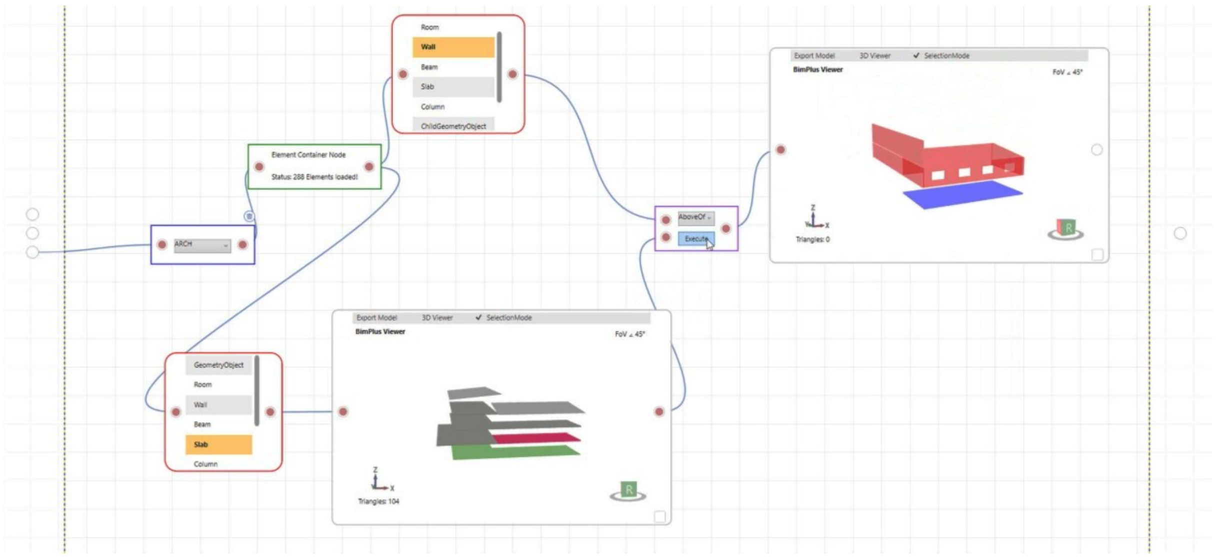

FIG. 21: User interface of the VCCL application

The prototype allows the user to build up a VCCL graph using a library of elementary nodes. Since bim+ is used as a database, which allows a fast loading and switching of different models, especially the principle of generality of a VCCL-graph comes into effect. As each graph is supposed to be applied to any valid building model, the model can be switched on the fly and the graph is subsequently re-processed. Most importantly, the application focuses particular on involving the user in the checking process. Therefore, most of the nodes are able to display intermediate results of the processing procedure by connecting the UI nodes with the output ports. In this way, the user is able to check if the processed result meets his expectations and requirements. Any defined VCCL graph can be stored in a native XML-format, which basically contains a list of the used methods, ports and edges. In this way the stored graphs can be exchanged with different project stakeholders.

The developed tool was examined for its practical applicability by domain experts. To this end, the central regulations of DIN 18232-2 discussed in Section 4 were translated into VCCL and successful semi-automated. Exemplarily, the user interface as well as a result of a geometric checking, which stated an intermediate result of the VCCL processing, is shown in FIG. 22.

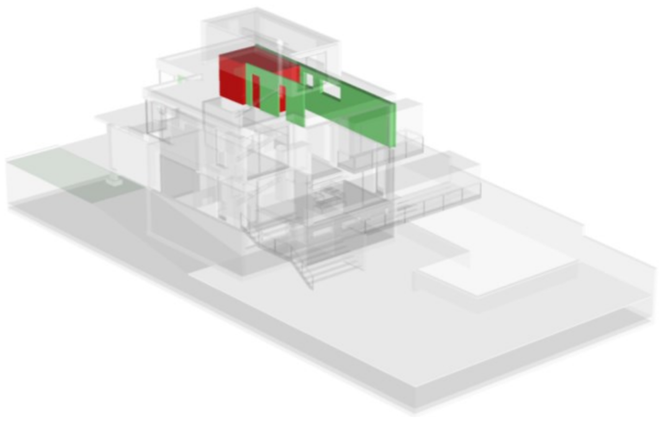

FIG. 22: Visualized intermediate result of a checking program: Identification of the building elements containing an opening and bounding to room with defined properties

6. CONCLUSIONS AND OUTLOOK

There is an extraordinary relevance and importance of methods for Automated Code Compliance Checking for the construction industry. However, the approaches developed so far have a number of shortcomings. The most advanced commercial tools typically realize a black-box approach where the code checking procedures are hidden from the user and remain widely unmodifiable. For many building authorities, the non-transparency involved is not acceptable. On the other hand, the scientific approaches aiming at representing rules in an open computer-processable format typically use notations which are hardly accessible by AEC domain experts without profound programming skills.

To overcome these limitation, we developed an approach which is based on using a visual programming language for encoding code checking procedures. We defined a dedicated language, the visual code checking language (VCCL) and presented the syntax and semantics of its major components. In this paper, we presented the first steps towards proving the practical viability of the approach. Future publications will provide more extensive applications for a large set of different application domains.

Besides, there are a number of challenges, which have to be tackled in the future work. A general criticism of visual programming languages is that complex translation tasks quickly lead to a very large and unclear programs, which can no longer be interpreted by the user. In our approach we implemented various mechanisms like the nesting approach, the embedded UI controls or the control flow elements to counteract overly complex visual programs. However, further application examples are needed to investigate if these mechanisms are sufficient to address the challenges.

Although it is one of the principles of the VCCL to make each individual process step visible to the user, it becomes clear that there are application examples where the user might not be interested to have a too deep insight in expert routines. In order to define this level of “black-box acceptance”, further test cases must be examined. It is also likely that this level is different for the various application areas and depending on the experience of the user. In any case, care must be taken to ensure that too specific methods which relate exclusively to a single application case are not defined as atomic methods, because this would violate the principle of genericity.

In building practice, there is a variety of codes and many different ways of presenting information. Therefore, it is necessary to develop more VCCL elements in general, which are able to represent this information within a node and finally in a VCCL graph. Based on an analysis of other standards, these representations can be identified and serve as a basis for further development. In this way, a library of VCCL elements progressively arises, which captures gradually more different applications.

Last but not least, the information described in a VCCL program has to be exchanged between different project stakeholders. Currently the contents of a VCCL program can only be stored as a native XML-file but not in a generic format. As shown in Section 2 currently there is no common data standard for the representation of the guideline knowledge, but BuildingSmart (2016) recently founded the working group *Regulatory Room*, which aims to find an open format that is able to represent this knowledge. Although it will presumably take some time until first results will be available, it is planned to align VCCL developments with the outcomes of this working group and make the VCCL content independently accessible.

To put it in a nutshell, the introduction of a visual language for Automated Code Compliance Checking opens up a variety of new opportunities to improve the process by involving the user in the process.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support by the Nemetschek Group as well as ALLPLAN GmbH for the research project presented in this paper.

REFERENCES

Allplan GmbH (2016), “bim+”, available at: https://www.bimplus.net/de/ (accessed 8 March 2016).

Anton, I. and Tănase, D. (2016), “Informed Geometries. Parametric Modelling and Energy Analysis in Early Stages of Design”, *Energy Procedia*, Vol. 85, pp. 9–16.

bim+ (2016), “Documentation”, available at: https://doc.bimplus.net/ (accessed 15 May 2016).

Böhm, C. and Jacopini, G. (1966), “Flow diagrams, turing machines and languages with only two formation rules”, *Communications of the ACM*, Vol. 9 No. 5, pp. 366–371.

BuildingSmart (2016), “Regulatory Room”, available at: http://buildingsmart.org/standards/standards-organization/rooms/regulatory-room/ (accessed 25 August 2016).

Catarci, T. and Santucci, G. (1995), “Are Visual Query Languages Easier to use than traditional ones? an Experimental Proof”, in Kirby, M.A.R. (Ed.), *People and computers X: Proceedings of HCI '95, Huddersfield, August 1995*, Cambridge Univ. Pr, Cambridge.

Codd, E.F. (1970), “A relational model of data for large shared data banks”, *Communications of the ACM*, Vol. 13 No. 6, pp. 377–387.

Codd, E.F. (1991), *The relational model for database management: Version 2,* Reprinted with corr, Addison-Wesley, Reading, Mass.

DIN (2007), *18232:2007-11 - Smoke and heat control systems - Part 2: Natural smoke and heat exhaust ventilators; design, requirements and installation*, Beuth, available at: https://www.beuth.de/de/norm/din-18232-2/101330298.

Ding, L., Drogemuller, R., Rosenman, M. and Marchant, D. (2006), “Automating code checking for building designs - DesignCheck”, *Cooperative Research Centre (CRC) for Construction Innovation*, pp. 1–16.

Eastman, C. (2009), “Automated Assessment of Early Concept Designs”, *Architectural Design*, Vol. 79 No. 2, pp. 52–57.

Eastman, C., Lee, J.-m., Jeong, Y.-s. and Lee, J.-K. (2009), “Automatic rule-based checking of building designs”, *Automation in Construction*, Vol. 18 No. 8, pp. 1011–1033.

Hils, D.D. (1992), “Visual languages and computing survey. Data flow visual programming languages”, *Journal of Visual Languages & Computing*, Vol. 3 No. 1, pp. 69–101.

Hjelseth, E. (2012), “Converting performance based regulations into computable rules in BIM based model checking software”, in Gudnason, G. (Ed.), *eWork and eBusiness in Architecture, Engineering and Construction: ECPPM 2012*, CRC Press, Hoboken, pp. 461–469.

Hjelseth, E. and Nisbet, N. (2011), “Capturing normative constraints by use of the semantic mark-up RASE methodology”, *Proceedings of the 28th International Conference of CIB W78*, pp. 26–28.

ISO (2011), *Building construction - Accessibility and usability of the built environment* No. 21542:2011, available at: http://www.iso.org/iso/catalogue_detail?csnumber=50498 (accessed 10 October 2016).

Kerrigan, S. and Law, K.H. (2003), “Logic-based regulation compliance-assistance”, in Zeleznikow, J. (Ed.), *Proceedings of the 9th international conference on Artificial intelligence and law*, ACM, New York, NY, pp. 126–135.

Kim, H. and Grobler, F. (2009), “Design Coordination in Building Information Modeling (BIM) Using Ontological Consistency Checking”, in Caldas, C.H. and O'Brien, W.J. (Eds.), *Computing in civil engineering: Proceedings of the 2009 ASCE International Workshop on Computing in Civil Engineering ; June 24 - 27, 2009, Austin, Texas*, ASCE, Reston, Va, pp. 410–420.

Korea Legislation Research Institute (2008), *Article 34 Clause 1*, available at: https://elaw.klri.re.kr/kor_service/jomunPrint.do?hseq=33006&cseq=926869.

Lange, M. (2008), *Semantische Integration von Bauplanungsinformationen am Beispiel des vorbeugenden Brandschutzes*, Zugl.: Darmstadt, Techn. Univ., Diss., 2007, *Berichte des Instituts für Numerische Methoden und Informatik im Bauwesen*, Vol. 2008,1, Shaker, Aachen.

Lê, M., Mohus, F., Kvarsvik, O.K. and Lie, M. (2006), “The HITOS Project - A Full Scale IFC Test”, in Martínez, M. and Scherer, R. (Eds.), *eWork and eBusiness in architecture, engineering and construction: Proceedings of the 6th European Conference on Product and Process Modelling [ECPPM 2006], 13 - 15 September 2006, Valencia, Spain*, Taylor& Francis, London.

Lee, J.K. (2011), “Building Environment Rule and Analysis (BERA) Language”, Architecture, Georgia Institute of Technology, 2011.

Lee, J.-K., Eastman, C.M. and Lee, Y.C. (2015), “Implementation of a BIM Domain-specific Language for the Building Environment Rule and Analysis”, *Journal of Intelligent & Robotic Systems*, Vol. 79 No. 3-4, pp. 507–522.

Myers, B.A. (1990), “Taxonomies of visual programming and program visualization”, *Journal of Visual Languages & Computing*, Vol. 1 No. 1, pp. 97–123.

Nisbet, N., Wix, J. and Conover, D. (2008a), “The Future of Virtual Construction and Regulation Checking”, in Brandon, P.S. and Kocatürk, T. (Eds.), *Virtual futures for design, construction & procurement*, Blackwell Pub, Oxford, Malden, MA, pp. 241–250.

Nisbet, N., Wix, J. and Conover, D. (2008b), “The future of virtual The future of virtual construction and regulation checking”, paper presented at AEC-ST Conference Presentation, Anaheim, USA.

Park, S. and Lee, J.-K. (2016), “KBimCode-based Applications for the Representation, Definition and Evaluation of Building Permit Rules”, in *2016 Proceedings of the 33rd ISARC*, Auburn, USA, pp. 720–728.

Preidel, C. and Borrmann, A. (2015), “Automated Code Compliance Checking Based on a Visual Language and Building Information Modeling”, in *Connected to the future: 32nd International Symposium on Automation and Robotics in Construction and Mining (ISARC 2015) Oulu, Finland, 15-18 June 2015, Oulu, Finland*, Curran Associates Inc, Red Hook, NY.

Preidel, C. and Borrmann, A. (2016), "Integrating Relational Algebra into a Visual Code Checking Language for Information Retrieval from Building Information Models", in Yabuki, N. and Makanae, K. (Eds.), *Proceedings of the 16th International Conference on Computing in Civil and Building Engineering. Osaka, Japan: ICCCBE*, Osaka, Japan.

Preidel, C., Borrmann, A. and Beetz, J. (2015), "BIM-gestützte Prüfung von Normen und Richtlinien", in Borrmann, A., König, M., Koch, C. and Beetz, J. (Eds.), *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*, *VDI-Buch*, Springer Vieweg, Wiesbaden.

Schiffer, S. (1998), *Visuelle Programmierung: Grundlagen und Einsatzmöglichkeiten*, Addison-Wesley, Bonn.

Solihin, W. (2004), *Lessons learned from experience of code-checking implementation in Singapore, BuildingSMART Conference*, Singapore.

Uhm, M., Lee, G., Park, Y., Kim, S., Jung, J. and Lee, J.-K. (2015), "Requirements for computational rule checking of requests for proposals (RFPs) for building designs in South Korea", *Advanced Engineering Informatics*, Vol. 29 No. 3, pp. 602–615.

United States Access Board (2014), *ADA and ABA Accessibility Guidelines*, available at: https://www.access-board.gov/attachments/article/412/ada-aba.pdf (accessed 10 October 2016).

Wilson, P. (1988), "STEP and EXPRESS", in Patrikalakis, N. (Ed.), *NSF Workshop on Distributed Information, Computation and Process Management for Scientific and Engineering Environments: 15 -16 May*, Herndon, Virginia.

Yurchyshyna, A., Faron-Zucker, C., Le Thanh, N. and Zarli, A. (2008), "Towards an ontology-enabled approach for modeling the process of conformity checking in construction", *CEUR Workshop Proceedings*, Vol. 344, pp. 21–24.

Yurchyshyna, A. and Zarli, A. (2009), "An ontology-based approach for formalisation and semantic organisation of conformance requirements in construction", *Automation in Construction*, Vol. 18 No. 8, pp. 1084–1098.

Zhang, S., Teizer, J., Lee, J.-K., Eastman, C.M. and Venugopal, M. (2013), "Building Information Modeling (BIM) and Safety. Automatic Safety Checking of Construction Models and Schedules", *Automation in Construction*, Vol. 29, pp. 183–195.