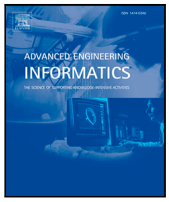
Advanced Engineering Informatics 58 (2023) 102132

Contents lists available at ScienceDirect

Advanced Engineering Informatics

journal homepage: www.elsevier.com/locate/aei



MVDLite: A fast validation algorithm for Model View Definition rules

Han Liu [a,b], Ge Gao [a,c,*], Hehua Zhang [a,c], Yu-Shen Liu [a,c], Yan Song [b], Ming Gu [a,c]

[a] Tsinghua University, Haidian, Beijing, China
[b] Digital Horizon Technology Co., Ltd., Shenzhen, China
[c] Beijing National Research Center for Information Science and Technology (BNRist), Haidian, Beijing, China

ARTICLE INFO

Keywords:
Model View Definition (MVD)
Industry Fundamental Classes (IFC)
Building Information Model (BIM)
Automated compliance checking

ABSTRACT

Model View Definition (MVD) is the mainstream method for formally representing the knowledge about the data exchange of building information models (BIMs). The MVD ruleset can be used in automatically validating the compliance of the BIM data against the exchange requirements and rule constraints. At present, MVD validation is performed with the "matching-checking" process according to the data structure templates and the rule statements respectively. However, there are redundant visit of edges and nodes in a process with separated subgraph matching and value checking, and the efficiency of MVD validation on large real-world models is still a challenge. In this paper, the MVDLite algorithm is proposed for the fast validation of MVD rules. By integrating the separated templates and statements into a "rule-chain" structure, a fast round-trip searching can be performed, which remarkably speeds up the MVD validation in the experiments. Discussions on the complexity of the MVD validation algorithms and the applicable scope of the proposed algorithm are also provided, which intends to bring some new ideas to the community about the future evolution of the MVD technology.

1. Introduction

The Building Information Models (BIMs) are digital expressions of physical and functional characteristics of objects in the process of design, construction, operation and maintenance of assets in the architectural, engineering and construction (AEC) field. A BIM system consists of a variety of software with different functions, including BIM modeling tools, data platforms and analysis tools, and many stakeholders are involved in it. All kinds of BIM software tools form a pipeline in the whole production process. The upstream software needs to provide BIM data with the correct format and sufficient required information to support the normal operation of the downstream software.

BIM data exchange is the process of transferring data between heterogeneous BIM software tools, and it is the key link for realizing BIM-based collaboration among stakeholders. The interoperability of BIM data exchange means that within a specific requirement boundary, the results of data exchange can guarantee the normal operation of software supported by BIM.

The buildingSMART international organization has proposed several standards for supporting the interoperability of BIM data exchange. The Industry Foundation Classes (IFC) [1] is a standard open-source data schema for BIMs, in which the object types and data structure are specified. The Information Delivery Manual (IDM) [2,3] is a standard method for defining data exchange process, requirements and constraints based on IFC. The Model View Definition (MVD) [4,5] is a standard method for the digitalization of IDM, for supporting software implementation and automatic data validation.

The IFC schema provided basic object types for supporting BIM data exchange, including product types, geometry representations, properties, relationships, and so on. Each row in an IFC file corresponds to a data node, and each type of node has a list of named attributes. The value of attributes can be literals (such as strings or integer numbers) or references pointing to other nodes. All the nodes and references form a directed acyclic graph (DAG) structure. A DAG is a graph with directed edges and no cycled references, which is a common data structure to ensure that every node corresponds to determined data by a subgraph rooted with the node. In addition, several named inverse attributes are defined in the IFC schema for accessing the nodes that reference the current node, and such inverse attributes are usually used to link "IfcRelationship" nodes.

IFC is a standard with compatibility for general-purpose data exchange in the construction industry, in which the data nodes are in coarse-grained classes, and the data types and properties are extensible. The IDM-MVD methodology is proposed for defining fine-grained domain concepts corresponding to different nodesets in the IFC data, and for representing the data exchange requirements as rule constraints [2,

* Corresponding author at: Tsinghua University, Haidian, Beijing, China.
E-mail addresses: liuhan@digital-h.cn (H. Liu), gaoge@tsinghua.edu.cn (G. Gao).

https://doi.org/10.1016/j.aei.2023.102132
Received 16 December 2022; Received in revised form 4 June 2023; Accepted 7 August 2023
Available online 21 August 2023
1474-0346/

3,6,7]. The digitalized MVD rules are commonly represented as an mvdXML ruleset [8,9], which is recommended by the buildingSMART. The mvdXML rules can be parsed by computers, which can support the software implementation of IFC-based data exchange, and can be used for automatically validating whether IFC models conform to the MVD rules.

The digitalized MVD rules in the mvdXML ruleset are mainly in two parts. The first part is the subgraph templates in the header part as nested XML tags, which define the node types and edge names that form a subgraph structure based on the IFC schema. The subgraph templates are used to find matched subgraphs in the whole graph structure of an IFC file. The second part is the rule statements, written in "mvdXML Rule Grammar" in the body part, which define the value constraints and logical interconnections for checking each matched subgraph.

Compared with the BIM code-checking methods with enriched geometry calculation and semantic inferencing [10–13], the MVD validation task focuses on fast finding data nodes and fast validation of data structures and values in the IFC raw data. MVD validation can be used as a fast pre-checking step for various applications that use IFC as input data, including the enriched code-checking tasks [14,15]. The applicability rules in MVD can be used to quickly find fine-grained nodesets according to the conditions. The constraint rules in MVD can be used to check the conformance of data and to fetch required attributes from the data.

At present, there have been several implementations of validation algorithms for mvdXML rulesets [16–19]. In accordance with the separation of templates and rule statements in mvdXML, the current validation algorithms usually follow the two-step "matching-checking" process: first matching the template to find subgraphs from the data, and then checking the rule statements on each found subgraph. However, in a process with separated subgraph matching and value checking, there may be redundant visits of edges in finding subgraphs, and the data values also tend to be checked repeatedly in the process. Hence the efficiency of MVD validation of large rulesets on real-world size models is still a challenge. The validation of an MVD ruleset with thousands of rules on a large-scale model may take hours, which is an obstacle in the promotion of MVD applications.

Aiming at the efficiency issue of MVD validation, the MVDLite algorithm is proposed in this paper for the fast validation of MVD rules. The main contributions are listed as follows.

- A "rule chain" structure is introduced to combine and reorganize the separated templates and rule statements from an input mvdXML ruleset.
- The MVDLite algorithm is proposed for fast MVD validation by performing round-trip searching on the rule chain structure.
- The complexity and applicable scope of the proposed algorithm are discussed.

2. Related work

In this section, first, the related studies on IDM-MVD method are reviewed (see Section 2.1). Next, the templates and rule statements in the mvdXML ruleset are introduced (see Section 2.2). Finally, the automated MVD validation methods are reviewed (see Section 2.3).

2.1. Research on the IDM-MVD method

The IFC schema is defined based on the 10303-11 "EXPRESS" method [20]. The ISO 10303 series provide the methodology for defining data schemas and performing validation on data to ensure conformance to the schema. Based on the schema-level validation in ISO 10303, the IDM method is introduced in ISO 29481 [21] and ISO 15926-11 [22] for specifying user-defined data delivery process, exchange requirements, and constraint rules, in order to support trusted information exchange in the industry.

The MVD method is introduced in ISO 29481 as a technology to bind the requirements and constraints to a certain data schema [21]. MVD supports fine-grained validation and filtering of data, which intend to realize meaningful IFC implementations for software developers [4]. According to ISO 29481, each rule in an MVD ruleset corresponds to an "information unit" in the IDM document. An information unit in IDM is about either a domain entity (e.g. a window) or an attribute of a domain entity (e.g. the height of a window). One information unit may have several attached constraints that are described in natural language. As the technical solution of IDM, MVD rules are created to represent the information units and constraints in computer-executable rules. A domain entity is bound to a root nodeset in the data model, and an attribute is bound to a path starting from a root node to reach the required attribute node. The rule constraints are validated on the subgraphs composed of the root entities and the attribute paths.

There have been several studies on implementing the IDM-MVD method from different aspects. The Extended Process to Product Modeling (**xPPM**) [23] focuses on the representation of IDM exchange requirements in MVD. The xPPM is based on the formal definition of exchange requirements and functional parts in IDM. A tool is provided to build up the process maps with reusable functional parts. The functional parts are mapped to IFC element types for generating MVD rules, and the links between the functional parts and the exchange requirements are kept in the MVD ruleset. The Semantic Exchange Modules framework (**SEM**) [24,25] focuses on the definition of domain concepts and their mapping to the IFC data structure. The SEM method defines object-oriented and reusable modules for representing entities, attributes and relationships. By mapping the domain concept modules to the IFC schema, the data structure templates in the MVD ruleset can be generated accordingly. The Generalised Model Subset Definition (**GMSD**) [26] focuses on the rule constraints for specifying the partial models. The GMSD method proposes the specification of rules for querying a subset of entities from the IFC model. The entities can be filtered by the range of values. The boolean operations between the entity collections are also supported.

Based on the previous research about MVD, an integrated IDM-MVD process for IFC data exchange is proposed and recommended by buildingSMART [3], which combines the strengths of the previous research. The mvdXML format [9] is used for this integrated IDM-MVD process, which involves abundant information about exchange requirements, domain concepts, and rule constraints. The recommended IDM-MVD process mainly includes:

- Defining the domain concepts (entities, properties, and geometries), relationships and rule constraints in an Exchange Requirements Model (ERM) as forms and diagrams.
- Binding the concept definitions in the ERM into a specific IFC Release (such as IFC4).
- Writing the relationships and rule constraints into a specific IFC Release as MVD Implementation Guidance forms and diagrams.
- Implementing the mvdXML according to the concept bindings, forms, and diagrams.

In addition to mvdXML, there are other studies on specifying and validating exchange requirements and constraints on industrial data models. Many of these studies are based on semantic web technologies. The data models defined with the 10303-11 "EXPRESS" method [20] can be represented as RDF graphs, including the IfcOWL [27] and the ISO 15926 data models [28]. The Linked Building Data (LBD) is another recent topic for modularized representation of semantic models in the construction industry [29]. SHACL (Shapes Constraint Language) [30] is recommended in ISO 15926-10 as a validation method for data models [31]. SHACL is also used in validating the data structures and constraints of IfcOWL and LBD models [14,32–34]. SHACL has much in common with mvdXML since they both represent the rules on subgraphs composed of root nodesets and attribute paths.

```
<?xml version="1.0"?>
<mvdXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
uuid="ca533756-fc05-4f56-b091-04b953e362ed" status="sample" xsi:schemaLocation="http://www.buildingsmart-
tech.org/mvd/XML/1.1 http://www.buildingsmart-tech.org/mvd/XML/1.1/mvdXML_V1.1_add1.xsd"
xmlns="http://buildingsmart-tech.org/mvd/XML/1.1">
  <Templates>
    <ConceptTemplate uuid="d22c5deb-95d9-4686-800b-a23d292062c9" status="sample" applicableSchema="IFC4"
        applicableEntity="IfcWall">
      <Rules>
        <AttributeRule RuleID="IsTypedBy_0" AttributeName="IsTypedBy">
          <EntityRules>
            <EntityRule EntityName="IfcRelDefinesByType">
              <AttributeRules>
                <AttributeRule RuleID="RelatingType_1" AttributeName="RelatingType">
                  <EntityRules>
                    <EntityRule EntityName="IfcTypeObject">
                      <AttributeRules>
                        <AttributeRule RuleID="HasPropertySets_2" AttributeName="HasPropertySets">
                          <EntityRules>
                            <EntityRule EntityName="IfcPropertySet">
                              <AttributeRules>
                                <AttributeRule RuleID="Name_3" AttributeName="Name">
                                  <EntityRules>
                                    <EntityRule EntityName="IfcLabel" />
                                  </EntityRules>
                                </AttributeRule>
                                <AttributeRule RuleID="HasProperties_4" AttributeName="HasProperties">
                                  <EntityRules>
                                    <EntityRule EntityName="IfcPropertySingleValue">
                                      <AttributeRules>
                                        <AttributeRule RuleID="Name_5" AttributeName="Name">
                                          <EntityRules>
                                            <EntityRule EntityName="IfcIdentifier" />
                                          </EntityRules>
                                        </AttributeRule>
                                        <AttributeRule RuleID="NominalValue_6" AttributeName="NominalValue">
                                          <EntityRules>
                                            <EntityRule EntityName="IfcValue" />
                                          </EntityRules>
      ......
  </Templates>
  <Views>
    <ModelView uuid="d305b4eb-b668-437a-a9ef-a9ce0d55aa2b" status="sample">
      <Roots>
        <ConceptRoot uuid="9ebbc41e-b172-474a-a45e-b862ddd423b8" name="IfcWall" status="sample"
            applicableRootEntity="IfcWall">
          <Concepts>
            <Concept uuid="309fe979-2673-414a-a85b-b34b3b22fd03" name="Rule 1" status="draft" override="true">
              <Template ref="d22c5deb-95d9-4686-800b-a23d292062c9" />
              <TemplateRules operator="and">
                <TemplateRule Parameters="Name_3[Value]='Pset_WallCommon' AND Name_5[Value]='IsExternal' AND
                NominalValue_6[Type]='IfcBoolean'"/>
              </TemplateRules>
      ......
  </Views>
</mvdXML>
```

Templates about subgraph structures

Rule statements

Fig. 1. An example mvdXML ruleset.

The above-mentioned studies focus on the fast validation of raw data structures and value constraints to meet data exchange requirements. Compared with the code checking methods with enriched geometry calculation and semantic inferencing [10–13,35–38], the validation methods for data exchange requirements tend to be more lightweight. The rules are with limited functions concentrating on the information units and constraints in exchange requirements. The validation tasks are supposed to be performed on the receiving of data models to check the correctness of the data structure and the sufficiency of the content, and to ensure that the data meets the exchange requirements of subsequent automatic services [39]. The validation of raw data structures and values can be a fast pre-checking process before the enriched rule checking tasks to ensure the conformance of input data [14,15].

The mvdXML is currently accepted as a mainstream ruleset format that is widely used in BIM data validation and BIM software certification. The mvdXML rules have close connections with the exchange requirements and use cases in the IDM method, and can provide useful information for both domain engineers and software developers. In this paper, the proposed method uses mvdXML as input, and tries to find speeding-up strategies for the validation of data structures and value constraints of industrial data models.

2.2. The mvdXML ruleset

Currently, the most widely used version of mvdXML is V1.1 [9]. In this paper, all the analyses and experimental comparisons are based on mvdXML V1.1. An example mvdXML ruleset is shown in Fig. 1.

In an mvdXML ruleset, each MVD rule is composed of two parts: the subgraph templates in the header part, and the rule statements (with the logical interconnections between the statements) in the body part.

The subgraph templates are contained in the “<Templates>” part at the beginning of the file. Each subgraph template is defined as a “<ConceptTemplate>” structure, which indicates a certain version of IFC schema with the “applicableSchema” field, and indicates a root entity node type with the “applicableEntity” field. Inside a “<ConceptTemplate>” structure, the nested “<AttributeRules>” and “<EntityRules>” form the subgraph structure of the template. Each item in the “<EntityRules>” is a named type of node, including the entity node types and the literal data types. Each item in the “<AttributeRules>” is a named type of edge that starts from the current entity node type and points to the next node type. The names of the edges are defined in the IFC schema, including both the attributes directly recorded in the IFC data nodes, and the inverse attributes for accessing the nodes where the current node is referred to. One node can have several internal edge definitions, and also one edge can point to several different node types. Each edge may have a short “RuleID” string, so that the edge can be referred to in the body part of the ruleset.

The rule statements are contained in the “<ModelView>” parts at the body of the file. Inside of which, one “<ConceptRoot>” is an executable rule consisting of several rule statements with logical interconnections. The rule statements are in two groups: the applicability rules and the constraint rules, both of which must refer to a “<ConceptTemplate>” tag by a UUID for obtaining the subgraph structure information. The applicability rules are used for defining the conditions to filter a fine-grained subset of the root entities of a certain type, and the constraint rules are about the requirements for the correctness and sufficiency of data structures and values, which are applied to the subset nodes in the validation tasks.

The rule statements are written in the “mvdXML Rule Grammar”. A statement is the logical combination of several clauses, and each clause is usually composed of the following four parts.

- RuleID reference: a reference to an edge in a "<AttributeRules>" structure in the header part, standing for the data nodes that the edge points to.
- Metric: the measures for making judgments on the nodes, including "[Type]", "[Value]", "[Size]", "[Exists]", and "[Unique]".
- Operator: symbols for the comparison between the metric of nodes and the values, including "=", ">", "<", ">=", "<=", "!=".
- Value: the target value for comparison, which can be either a string value, boolean value, numeric value, or a regular expression.

The logical combinations of the rule statements are in two forms. Inside a rule statement, the interconnections (including "AND", "OR", "XOR", "NAND", "NOR", and "NXOR") together with the brackets are used to link the clauses. Outside of a rule statement, a nested XML structure composed of "<TemplateRules>" with operators ("and", "or", and "not") is used for assembling the rule statements. The two forms of logical combination are used in different phases in the MVD validation, as shown in Section 2.3.

In addition to the subgraph templates and rule statements for automatic MVD validation, there is also information in natural language in an mvdXML ruleset for describing the exchange requirements, specifying the severity level, and grouping the rules. Such human-readable information is helpful in providing documents and instructions for the MVD users in the data exchange process.

2.3. Automated MVD validation

Currently, automatic MVD validation has been implemented on several software platforms, such as Solibri [40], EXPRESS Data Manager (EDM) [41], Simplebim [42], Xbim [16,43], BIMserver [17,44], and IfcOpenShell [18,19]. The tools can be roughly divided into two groups: modularized MVD validation and generalized MVD validation.

The modularized MVD validation [15,40,42,45–47] is to implement program modules for the commonly-used MVD rule types, so that the users can perform the checking task by assembling the modules and configuring the parameters. The modularized MVD validation is based on the classification of MVD rule types. The mvdXML developer team categorized the mvdXML-based automated validation tasks [8], such as the existence of attributes, the size of collections, and the uniqueness of values. Similar categorizations are also proposed in other studies [6,45–47]. The modularized MVD validation is easy to understand by users and is commonly adopted by commercial BIM checking tools [40,42]. However, it cannot support the validation of arbitrary MVD rules for the IFC schema.

Generalized MVD validation [16–19] performs subgraph template matching and rule constraint checking directly on the graph structure of IFC data, and it is supposed to be able to support arbitrary mvdXML rulesets. In performing the generalized MVD validation, the applicability rules are checked before the constraint rules, and the filtered nodeset from the applicability checking is used as the input nodeset of the constraint checking.

In addition to mvdXML, there are also implementations of generalized model view checking based on other computer languages. The EDM tool supports user-defined EXPRESS rules for model view validation [41]. Several tools use SHACL for validating the linked data models [14,32–34]. It is available to use SHACL to validate a linked data model composed of several different schemas [33], for example, a model using both IfcOWL [27] and BOT (Building Topology Ontology) [48]. A comparison of the reviewed tools for automatic BIM data conformance validation tools is listed in Table 1.

Due to the separation of subgraph templates and rule statements in the mvdXML ruleset, the current implementations of generalized MVD validation usually follow the matching-checking process. The major steps of the matching-checking process are as follows.

(a) **Root nodeset acquisition.** According to the root entity type of the subgraph template, a root nodeset is found in the IFC data.
(b) **Subgraph matching.** Starting from each root entity in the root nodeset, the subgraphs that match (or partially match) the template are found and recorded.
(c) **Rule statement checking on subgraphs.** For each subgraph, the rule statement is checked for obtaining the boolean result of this subgraph. (When the metrics "[Size]", "[Exists]" and "[Unique]" are used, the rule statement is checked on the aggregation of several subgraphs.)
(d) **Merging results to the root entity.** One root entity can usually have multiple matched subgraphs, and the boolean result of the root entity is decided by the existence of a subgraph that returns "TRUE" in the rule statement checking.
(e) **Logical combination of the root entity results.** For each root entity, the boolean results from multiple rule statements are combined into the final output boolean result according to the logical operators in the nested "<TemplateRules>" structure.

In this validation process, two different types of logical operators are used. The logical operators inside each statement are used in step (c) for combining the clauses for different attributes in the same subgraph. While the logical operators in "<TemplateRules>" tags are used in step (e), which is the combination of boolean results for the root entities.

The complexity of the matching-checking process is mainly determined by the subgraph matching step and the rule statement checking step. In the subgraph matching step, for a single root entity, there are usually multiple subgraphs that can match the same template. For example, if one root entity has multiple properties and each property is an "IfcProperty" node, then for this root entity, the number of matched subgraphs equals the number of properties. Typically, in checking a rule statement on a root nodeset with n entities, when the total number of matched subgraphs for all the root entities is nm (i.e. each root entity can match m subgraphs in average), and the subgraph template has p attributes, then the complexity of the subgraph matching step is $O(nmp)$. In the rule statement checking step, let p' be the number of attributes in the statement, then the complexity is $O(nmp')$. Usually, only part of the attributes will appear in a rule statement, and some attributes may appear multiple times with logical interconnections, so p and p' are considered to be in the same magnitude, and then $O(nmp)$ can be considered as the complexity of the whole matching-checking process.

The matched subgraphs are usually cached in some kinds of data structures, in order that the found subgraphs can be reused in checking multiple rule statements. For example, one implementation on the Xbim [16] caches the found subgraphs for each root entity set in a DataTable with size nm×p, in which each column stands for an attribute in the concept template, and each row contains the node values of a matched (or partially matched) subgraph. Some other implementations [17,18] cache the edges in found subgraphs for each root entity in a HashMap with size $n \times m \times p$, in which each root entity points to a subgraph set, and each subgraph records the key–value pairs of the attributes. The DataTable and HashMap structures may have different speeds in reading and writing, but mathematically, they have the same $O(nmp)$ complexity in storing the subgraphs.

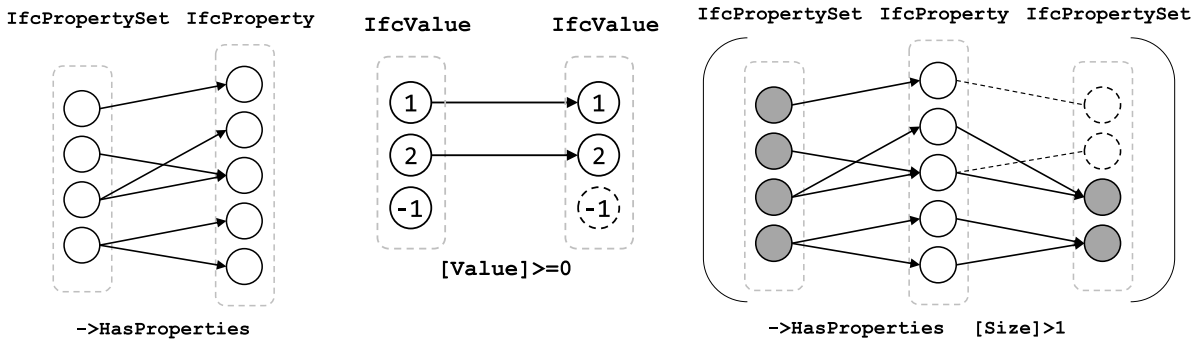
However, since the IFC model of a real-world project usually exceeds millions of nodes, with hundreds of megabytes of data, the efficiency of MVD validation of large rulesets on real-world size models is still a challenge. An MVD ruleset may contain thousands of rule statements, and in checking one rule statement, the total number of matched subgraphs nm may exceed several hundreds of thousands.

The matching-checking process faces the challenge of efficiency, and the motivation of the MVDLite algorithm proposed in this paper is based on the observations of the separated subgraph matching and rule statement checking steps. First, there are usually common nodes in multiple subgraphs with different root entities, which are visited multiple times in the matching-checking process. For example, if multiple

Table 1
The comparison of 13 BIM data conformance validation tools.

Tool name	Ruleset format	Model schema	Data storage dependency	Algorithm type
Solibri Model Checker [40]	(GUI configurations)	IFC	–	Modularized
SimpleBIM [42]	mvdXML-Excel	IFC	–	Modularized
Modularized rule-based validation [45]	Excel	IFC	–	Modularized
IfcDoc instance validation [46]	mvdXML	IFC	–	Modularized
EXPRESS Data Manager [41]	EXPRESS	IFC	–	Generalized
Xbim mvdXML plugin [16]	mvdXML	IFC	Xbim.Essentials[a]	Generalized
mvdXML Checker [17]	mvdXML	IFC	BIMserver[b]	Generalized
mvdXML Checker [18]	mvdXML	IFC	IfcOpenShell[c]	Generalized
python-mvdXML [19]	mvdXML	IFC	IfcOpenShell[c]	Generalized
Linked-data based constraint-checking [32]	SHACL	IfcOWL	RDFLib[d]	Generalized
LBD Checker [33]	SHACL	IfcOWL, BOT	Apache Jena[e]	Generalized
Validation of IfcOWL datasets using SHACL [34]	SHACL	IfcOWL	Apache Jena[e]	Generalized
ICDD SHACL validation [14]	SHACL	IfcOWL	dotnetRDF[f]	Generalized

[a] https://github.com/xBimTeam/XbimEssentials.
[b] https://github.com/opensourceBIM/BIMserver.
[c] https://github.com/IfcOpenShell/IfcOpenShell.
[d] https://github.com/RDFLib/rdflib.
[e] https://jena.apache.org/.
[f] https://github.com/dotnetrdf/dotnetrdf.



(a) Attribute segment. (b) Single metric segment. (c) Collection metric segment.

Fig. 2. Examples of attribute segment and metric segments.

instances refer to the same resource node, this resource node (as well as the property nodes linked to this resource node) would appear in the matched subgraphs of all instances. Second, if the constraints in rule statements are considered in the subgraph matching step, many false branches can be pruned at early stages in node searching, which may reduce the number of edges to visit.

Based on the above observations, the idea of the MVDLite algorithm is to reduce the complexity by reorganizing the separated templates and rule statements into an integrated searching process.

3. The MVDLite algorithm

In this section, the MVDLite algorithm is proposed for the fast validation of MVD rules. First, the "rule chain" structure with several types of "rule segments" is introduced (see Section 3.1). Second, the algorithm for parsing the mvdXML ruleset and composing the rule chain structure is proposed (see Section 3.2). Then, the round-trip searching and filtering on the rule chain structure is introduced (see Section 3.3). Finally, the deep-caching strategy is introduced to further speed up the checking tasks on a large ruleset (see Section 3.4).

3.1. The rule chain structure

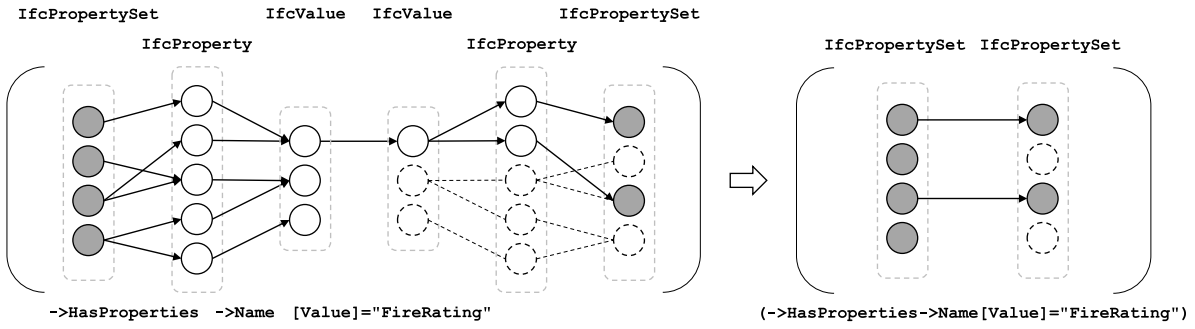
The rule chain is the rule structure used in the MVDLite algorithm to integrate the subgraph templates, rule statements and logical interconnections. A rule chain is composed of a sequence of rule segments starting from a root nodeset. One rule segment is the representation of a mapping from the source nodeset to the target nodeset. In the sequence of rule segments, the target nodeset of a former rule segment is the source nodeset of the latter rule segment.

There are three types of rule segments: attribute segment, metric segment and compound segment.

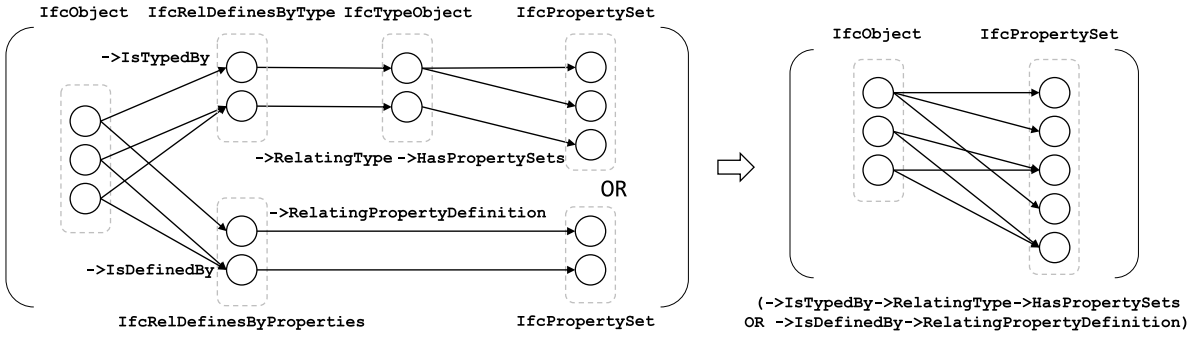
Attribute segment. An attribute segment is a mapping defined by an attribute in the IFC data. Each attribute segment has an attribute name and a target node type, corresponding to the "AttributeRule" and "EntityRule" tags in mvdXML. Fig. 2(a) shows an example attribute segment.

Metric segment. A metric segment is a mapping from the source nodeset to itself, which works as a filter for the source nodes. Each metric segment is with a metric ([Type], [Value], [Size], [Exists], and [Unique]), an operator (=, >, <, >=, <=, and !=) and a value constraint (string, boolean, or numeric value), corresponding to the components in the mvdXML Rule Grammar. Among the metric segments, the [Type] and [Value] metrics are "single metric segments", which can be evaluated by every single node in the nodeset, and act as filters for the nodeset itself. Fig. 2(b) shows an example single metric segment. The [Exists], [Size] and [Unique] are "collection metric segments", which can only be evaluated by a collection of nodes. A collection metric segment is calculated together with its previous attribute segment (or a compound attribute segment) and acts as a filter for the source nodeset of the previous attribute segment. Fig. 2(c) shows an example collection metric segment.

Compound segment. A compound segment encapsulates one or more rule chains into brackets, so that it can be embedded into a higher-level rule chain and can act as a single rule segment. There are two different types of compound segments: compound attribute



(a) Compound metric segment.



(b) Compound attribute segment.

Fig. 3. Examples of compound rule segments.

segment and compound metric segment. A compound metric segment encapsulates the filter result of the source nodeset, and acts as a single metric segment, as shown in Fig. 3(a). A compound attribute segment encapsulates the paths between the source nodeset and the target nodeset, and acts as a single attribute segment, as shown in Fig. 3(b). Each sub-chain in a compound metric segment must end with a metric segment, and each sub-chain in a compound attribute segment must end with an attribute segment. A compound segment can also be nested inside another compound segment, and it acts as a corresponding attribute segment or metric segment, so that the searching and filtering on subgraphs with branches can be supported.

Inside a compound segment, several sub-chains can be combined with logical interconnections (AND, OR, NOT), which correspond to the intersection, union and complement operations of the target nodesets. Specifically, since each metric segment can be viewed as a filter of its source nodeset, the AND operation of metric segments can also be represented as the series connection of metric segments.

Each rule segment is named with a string, which indicates the operation of the rule segment, and performs as a command to the program for seaching and filtering. As a result, a rule chain can be uniquely identified by linking the strings of all segments. An attribute segment is named with "->" followed by the attribute name, and the optional entity type rule can be linked afterward with a ":", such as "->Name:IfcLabel". A metric rule is named with the metric name, an operator and a value constraint in order, such as "[Value]=TRUE". A compound segment is named with exterior brackets, inside which are the strings of the interior rule chains and logical interconnections.

Fig. 4 shows a rule chain structure generated from an mvdXML rule statement. The meaning of this rule is that if a property "IsExternal" in property set "Pset_WallCommon" is assigned to an "IfcWall" instance, then the value type should be "IfcBoolean".

3.2. Composing the rule chain from the mvdXML ruleset

The MVDLite algorithm uses mvdXML as the input ruleset, and each rule statement is parsed into a rule chain structure according to the referred subgraph template, as shown in the pseudo-code of the "GenerateRuleChain" function in Algorithm 1. Fig. 5 shows the steps for composing the rule chain from the mvdXML ruleset.

The input of the "GenerateRuleChain" is the subgraph template **T** and the rule statement **R**. The rule statement is composed of tuples of a logical interconnection u and a sub-clause **S**. Each sub-clauses can either be a tuple of a RuleID v and a value constraint w, or an internal rule statement encapsulated in brackets. Let the first logical interconnection be NULL, since the number of sub-clauses is always larger than the number of logical interconnections by one.

The algorithm starts with an empty rule chain **c**, and iterates through all sub-clauses in the rule statement. When the sub-clause is a tuple $\langle v, w \rangle$, the RuleID v corresponds to a series of attribute segments by the function "GetAttributeSegChain", and the value constraint w corresponds to a metric segment by the function "GetMetricSeg". The attribute segments are the instructions to search from the root nodeset to the target nodeset, and the metric segment is added to the end of the attribute segments for filtering the target nodeset, to form a branch of the rule chain. When the sub-clause is an internal rule statement encapsulated in brackets, the algorithm recursively calls the "GenerateRuleChain" function to generate a sub-chain for the internal rule statement, which is also regarded as a branch of the rule chain.

In the iteration of the sub-clauses, the lowest-common-ancestor attribute between the current rule chain and the newly-returned branch is calculated with the function "GetLowestCommonAncestor". Then the two branches are merged into one rule chain by the function "MergeRuleChains". The merged chain has a shared prefix, and the suffix branches with interconnections are wrapped into compound segments. Specifically, when several compound metric segments are combined with "AND", they can also be connected as a series of compound metric segments, which means that the nodes should pass through all the filters.

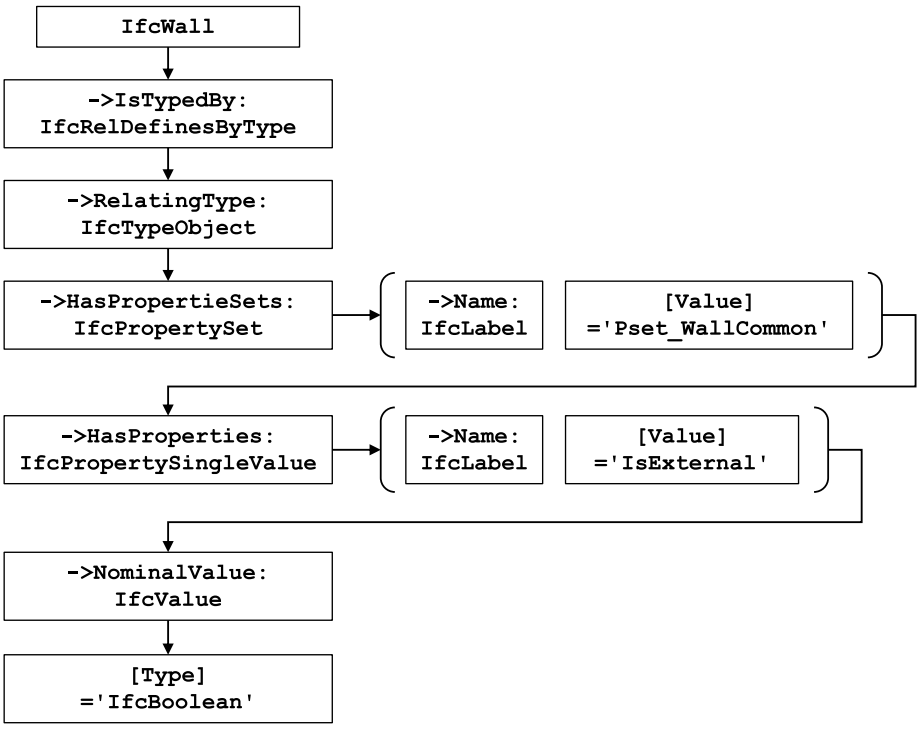


Fig. 4. An example rule chain structure with its string identification.

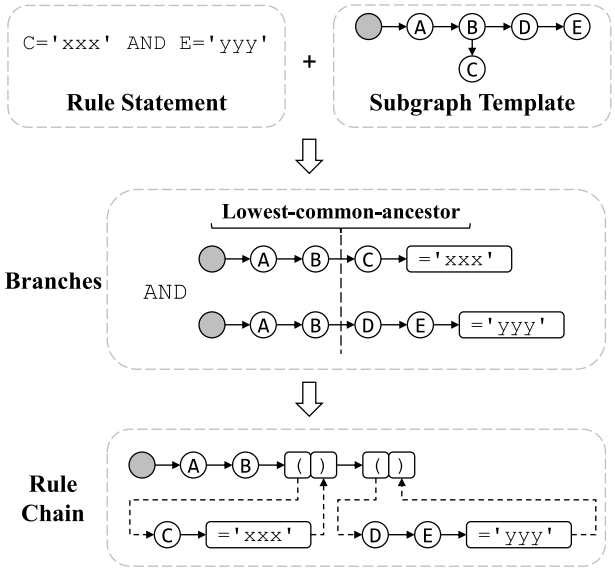


Fig. 5. Steps for composing the rule chain from the mvdXML ruleset.

Algorithm 1 GenerateRuleChain

Input: template **T**, rule statement **R**
rule chain **c** ← NULL
for tuple ⟨ interconnection u, sub-clause **S** ⟩ in **R** **do**
  if S is another rule statement **R′** in brackets **then**
    c′ ← GenerateRuleChain(**T**, **R′**)
  else
    S is tuple ⟨ RuleID v, value constraint w ⟩
    c′ ← GetAttributeSegChain(v, **T**)
    c′ ← **c′**+ GetMetricSeg(w)
  end if
  if c = NULL **then**
    c ← **c′**
  else
    x ← GetLowestCommonAncestor(**c**, **c′**)
    c ← MergeRuleChains(**c**, **c′**, x, u)
  end if
end for
Output: **c**

3.3. Performing validation on the rule chain

In Algorithm 2, a round-trip searching is performed based on the rule chain to get the validation results. A rule chain is a series of instructions for finding and filtering the nodes. Starting from the root entity set, a round-trip searching is performed: first go forward through the chain to find the existence of the paths which can pass all the rule segments, and then trace back to find the root entity nodeset where these paths started.

According to the instructions on each rule segment, the result of each segment is a mapping between two nodesets, either a forward mapping to the succeeding nodeset by an attribute segment, or a filter pointing to a subset of the current nodeset by a metric segment. When a segment is a compound segment, each inner branch is a sub-chain starting from the current nodeset, so "SearchOnRuleChain" is recursively called for each branch, and the combined result acts as a mapping between two nodesets in the host chain.

The behaviors of the sub-processes in Algorithm 2 are as follows.

- "FindSucceedingNodes": Given a current nodeset $\mathbf{n_i}$, an attribute segment $\mathbf{s_i}$, and the whole data graph **G**, the function searches starting from the current nodeset to find the succeeding nodeset according to the attribute name and target entity type defined in the attribute segment, and returns the found nodeset.
- "FilterNodes": Given a current nodeset $\mathbf{n_i}$, and a metric segment $\mathbf{s_i}$, the function filters the current nodeset and returns a subset that conforms to the constraint.

Algorithm 2 SearchOnRuleChain

Input: root nodeset $\mathbf{n_1}$, rule chain $\mathbf{c}$, data graph $\mathbf{G}$
for segment $\mathbf{s_i}$ in $\mathbf{c}$ **do**
  if $\mathbf{s_i}$ is an attribute segment **then**
    $\mathbf{n_{i+1}} \leftarrow$ FindSucceedingNodes($\mathbf{n_i}, \mathbf{s_i}, \mathbf{G}$)
  else if $\mathbf{s_i}$ is a metric segment **then**
    $\mathbf{n_{i+1}} \leftarrow$ FilterNodes($\mathbf{n_i}, \mathbf{s_i}$)
  else if $\mathbf{s_i}$ is a compound metric segment **then**
    $\mathbf{n_{i+1}} \leftarrow \emptyset$
    for tuple $\langle$ interconnection u, branch $\mathbf{c'}$ $\rangle$ in $\mathbf{s_i}$ **do**
      $\mathbf{n'} \leftarrow$ SearchOnRuleChain($\mathbf{n_i}, \mathbf{c'}, \mathbf{G}$)
      $\mathbf{n_{i+1}} \leftarrow$ CombineNodeset($\mathbf{n_{i+1}}, \mathbf{n'}, u$)
    end for
  end if
end for
if $\mathbf{s_{last}}$ is a metric segment **then**
  $\mathbf{n_{out}} \leftarrow$ Backtrack($\mathbf{c}, \mathbf{n_{last}}, \mathbf{n_1}$)
else
  $\mathbf{n_{out}} \leftarrow \mathbf{n_{last}}$ //only inside of a compound attribute segment
end if
Output: $\mathbf{n_{out}}$

- "CombineNodeset": Given two nodesets $\mathbf{n}$ and $\mathbf{n'}$, and an interconnection u, the function returns the corresponding boolean result of the nodesets.
- "Backtrack": Given a rule chain $\mathbf{c}$, the last nodeset $\mathbf{n_{last}}$ in its searching result, and the root nodeset $\mathbf{n_1}$, the function traces back through the mappings found by the rule segments, to find the root entities as the starting points of the paths that can pass all the rule segments, and returns the found subset of the root nodeset.

Fig. 6 compares the round-trip searching algorithm on the rule chain with the matching-checking process in MVD validation.

Fig. 6(a) shows the matching-checking process in checking a rule statement, as introduced in Section 2.3. Starting from the root entity set, in the matching step, each root entity is likely to have multiple matched subgraphs. The rule is checked on each subgraph, and the existence of a subgraph that can pass the rule constraint decides the validation result of the one root entity.

In comparison, Fig. 6(b) shows the round-trip searching on the rule chain in the MVDLite algorithm. Rather than traversal subgraph-by-subgraph according to the template matching result, the MVDLite algorithm is able to find the results of all subgraphs through one single turn of round-trip searching starting from the root nodeset. The nodes and edges tend to appear in multiple subgraphs, and be checked multiple times. While in the round-trip searching, each edge is visited at most twice, which tends to be more efficient in the calculation. Detailed discussions about the complexity of the algorithm are presented in Section 5.1.

3.4. The deep-caching strategy

For the checking task on a large ruleset with multiple rules, the efficiency of the MVDLite algorithm can be further improved by applying the "deep-caching" strategy. Based on the correspondence between the rule chain structure and a prefix string, the checking results can be reused across multiple rules.

By naming each rule segment with a string, the whole rule chain corresponds to a string record by linking the strings of all segments, and then each prefix of a rule chain corresponds to a prefix of the string, as shown in Fig. 4. The deep-caching strategy records the correspondence so that a previously visited nodeset can be re-found according to the prefix string of a rule chain.

As a result, rather than starting from the root nodeset every time, the algorithm can start from a visited nodeset with the longest common prefix. In a large ruleset, there are usually multiple rules with a common prefix. For example, all rules for checking the type properties of an "IfcWall" instance should have the common prefix "IfcWall ->IsTypedBy: IfcRelDefinesByType ->RelatingType: IfcTypeObject ->HasPropertieSets: IfcPropertySet", so the nodeset of the corresponding "IfcPropertySet" nodes can be reused across the rules. In our experiments, the deep-caching strategy is effective in speeding-up without significantly increasing memory usage.

4. Experiments

4.1. Experiments setup

In this section, the performance of the proposed MVDLite algorithm is compared with several matching-checking tools on models and rulesets in different sizes.

(1) The compared tools.

Among the four generalized MVD checking tools supporting mvdXML listed in Table 1, the mvdXML Checker on IfcOpenShell [18], the python-mvdXML [19] and the Xbim MVD plugin [16] are selected for comparison.

The mvdXML Checker on IfcOpenShell [18] and mvdXML Checker on BIMserver [17] have similar core algorithm codes in Java that are based on HashMap caching, and the mvdXML Checker on IfcOpenShell is selected because it is newer that supports the released mvdXML version. For each root entity, several HashMaps are used to cache the matched subgraphs, and each HashMap records the values of the attributes in a subgraph.

The python-mvdXML [19] is another implementation based on IfcOpenShell, and the core algorithm codes are in Python. The python-mvdXML tool also uses HashMap (dictionary) for subgraph caching.

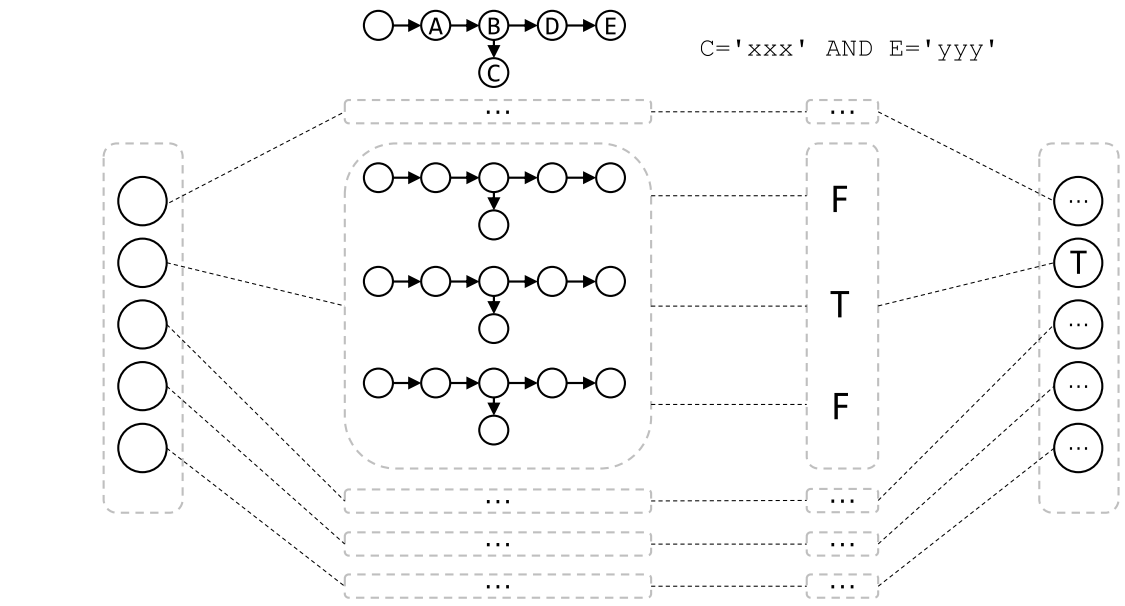
The Xbim mvdXML plugin [16] is implemented in C# with DataTable caching for matched subgraphs. One root entity corresponds to several rows in the DataTable, and each row records the values of the attributes in a subgraph. The rule statements are validated by querying the cached rows in the DataTable.

The MVDLite tool is implemented in C# based on an IFC model reader named "STEParser". The MVDLite algorithms with and without the deep-caching strategy are compared, which tests the effect of the deep-caching strategy on rulesets with multiple rules. For a fair comparison, our own matching-checking implementation with HashMap caching on the same STEParser tool is also tested.

On testing the tools, the time used for loading and parsing IFC data is excluded. For all the tools, the functions to export complex results (such as generating detailed reports with screenshots) are closed, and each tool just exports a plain text report or a simple JSON result file, so that the time used in checking can be fairly compared.

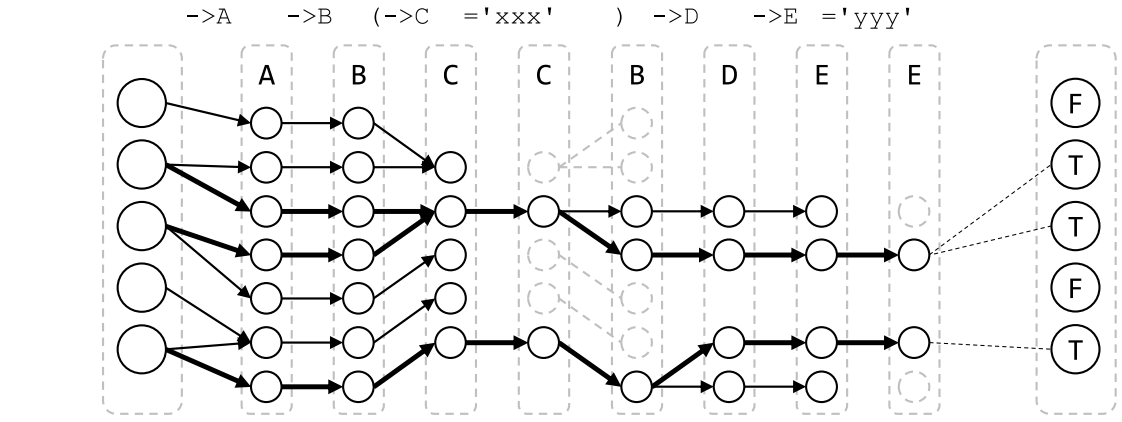
(2) Models. Several IFC4 models in different sizes are used. The models are shown in Fig. 7, and the sizes of these IFC models are listed in Table 2.

- **Duplex.ifc** A small sample model of a duplex building provided by NIBS [49]. This model is converted and merged from two IFC2X3 models to one IFC4 model, including an architectural model and an MEP model.
- **Office.ifc** A medium-size sample model of an office building provided by NIBS [49]. This model is converted and merged from three IFC2X3 models to one IFC4 model, including an architectural model, a structural model, and an MEP model.
- **B01.ifc** A model of the B01 underground storey of a commercial complex building. This model is exported from Autodesk Revit, which includes building elements in architecture, structure, and MEP disciplines.



(1) Get the root entity set. (2) For **each** root entity, list all subgraphs according to the template. (3) Validate the rule on **each** subgraph. (4) Get the result of **one** root entity.

(a) The matching-checking process.



(1) Get the root entity set. (2) Filter the nodes through the rule chain. (3) Backtrack to get the results of **all** root entities.

(b) The round-trip searching on the rule chain in the MVDLite algorithm.

Fig. 6. The comparison of two types of MVD rule validation algorithms.

Table 2
The size of models in the experiments.

	Duplex.ifc	Office.ifc	B01.ifc
File size	52 MB	193 MB	841 MB
"IfcElement"s	1170	7174	57,344
"IfcProperty"s	76,118	476,900	1,434,337
"IfcShapeModel"s	2029	11,095	89,735
Data lines	864,327	3,024,817	11,652,719

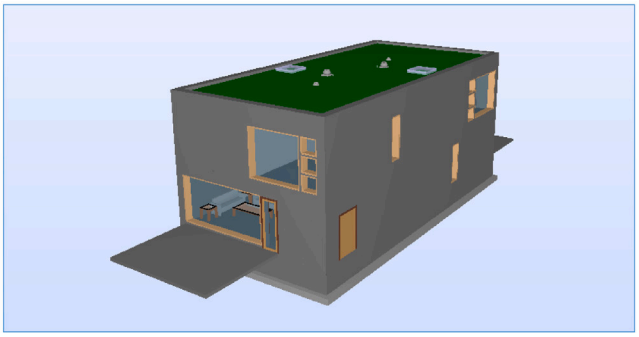
(3) Rulesets. Three mvdXML rulesets based on IFC4 are used in the validation experiments.

- **UnitTest.mvdxml** A small unit test ruleset with 58 statements about walls, which is used as the ruleset for unit test in the Xbim mvdXML validation tool [16].
- **RV.mvdxml** The IFC4 Reference View V1.1 draft with 1770 statements, provided by buildingSMART [50].
- **DTV.mvdxml** The IFC4 Design Transfer View V1.1 draft with 1791 statements, provided by buildingSMART [50].

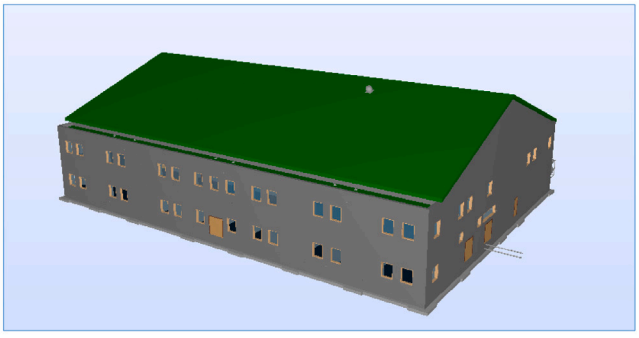
The rulesets are chosen in the experiments because they are open-accessible and cover various features of MVD rules. The "UnitTest" ruleset is small but with various types of rules for testing different checking functions in the Xbim mvdXML tool. The "RV" and "DTV" rulesets are relatively large and among the most-used MVD rulesets provided by buildingSMART.

Table 3
Time usage in MVD validation tasks (in seconds).

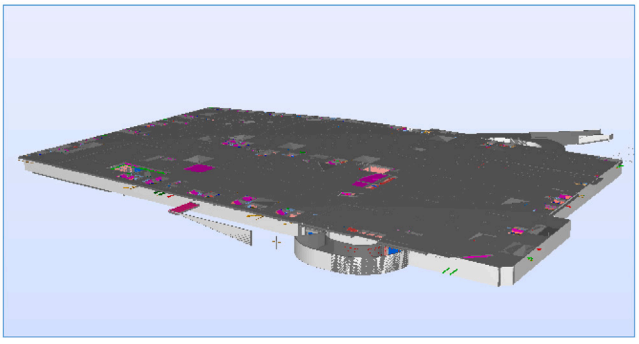
Models	Rulesets	Tools					
		(a) MVDLite	(b) MVDLite without deep-caching	(c) matching-checking on STEParser	(d) mvdXML-Checker IfcOpenShell [18]	(e) python-mvdXML IfcOpenShell [19]	(f) Xbim mvdXML plugin [16]
Duplex.ifc	UnitTest	**0.4**	0.5	1.3	2.5	2.0	2.4
	RV	**1.3**	2.0	11.2	21.1	170.1	7.1
	DTV	**1.4**	2.0	11.5	21.6	118.5	6.0
Office.ifc	UnitTest	**0.8**	1.5	5.8	8.6	14.1	177.3
	RV	**6.0**	9.7	88.5	97.0	658.8	185.7
	DTV	**6.0**	10.3	84.5	107.9	1137.0	178.3
B01.ifc	UnitTest	**5.0**	13.1	49.8	55.8	351.6	10,745.1
	RV	**73.2**	126.7	388.9	480.6	5798.1	8,722.8
	DTV	**68.6**	123.9	393.5	613.7	Fail	8,707.5



(a) Duplex.ifc



(b) Office.ifc



(c) B01.ifc

Fig. 7. The models used in the experiments.

4.2. Experimental results

Table 3 shows the time usage of the tools in MVD validation tasks, with minimum values in bold. All the experiments are performed on a PC with a 3.60 GHz processor and 32 GB of physical memory. All the algorithms run on a single core of the processor. In column (e), the python-mvdXML fail to finish the checking task of "DTV" on the "B01.ifc" model, because of the out-of-memory error.

For all the compared matching-checking tools, the number of entries in the data structures to cache found subgraphs and attributes are $O(nmp)$, which decides the basic time and space complexity in subgraph matching. The differences in the algorithm implementations to access data and process filter rules also contribute to the difference in executing time. The experimental results show that the efficiency of the Xbim tool in column (f) declines remarkably with the increase of the model size, and the efficiency of the python-mvdXML tool in column (e) declines remarkably with the increase of the ruleset size. The other two matching-checking tools shown in columns (c) and (d) have more balanced performance in tasks with different sizes of models and rulesets, and the time usages of the two tools are of the same magnitude in each task. Considering that the implementation of the MVDLite algorithm is based on the STEParser tool, it is proper to regard the result in column (c) as a baseline of performance.

The experimental results show that the MVDLite algorithm is significantly faster than the matching-checking tools in all tasks. Comparing the results in columns (a) and (c) in Table 3, the speed-up scales between 3 times to 14 times are achieved by the MVDLite algorithm. The results in columns (a) and (b) show that the deep-caching strategy can improve the performance of MVD validation on rulesets with multiple rules. By saving and reusing the visited prefixes on the rule chain in checking multiple rules, the time consumption can be reduced by around 40%.

5. Discussions

In this section, the discussions about the MVDLite algorithm are presented. First, the complexity of the MVDLite algorithm is provided, and compared with the matching-checking algorithm (see Section 5.1). Second, the applicable scope of the MVDLite algorithm is discussed, which is about the scope where the speeding-up strategies can be used (see Section 5.2). The discussions are supposed to explain the efficiency of the MVDLite algorithm, and to provide some ideas for keeping the fast validation of MVDs in future updates of the MVD technology.

5.1. The complexity of MVDLite algorithm

The complexity of the round-trip searching in the MVDLite algorithm is $O(e)$, in which e is the number of visited edges in the searching, and each edge is visited at most twice (forward and backward in a round-trip). In the rule chain, the number of nodes is not greater than e, and the number of metric segments on each nodeset is a small constant integer, so $O(e)$ can be considered as the complexity of the whole MVDLite algorithm.

In comparison, the complexity of the matching-checking method is $O(nmp)$, as shown in Section 2.3. In this section, the DataTable structure is used to describe the process of subgraph matching, and

to explain the comparison between the complexity of the matching-checking method and that of the MVDLite algorithm. The formulas to describe the subgraph matching process are shown as follows.

The subgraphs stored in a DataTable can be regarded as a matrix. In the matrix, each column stands for an attribute (with the first row as the root entities), and each row stands for a matched (including partially matched) subgraph. Since the template is a tree structure, the attributes can be sorted with topological sorting, so that all prefixes of any attribute are ranked before itself. For a template with p attributes (regarding the root entity set as the first attribute), let $f(i)$ be the index of the direct prefix of the ith attribute, $1 \le f(i) < i \le p$.

With the sorted attributes, the subgraph matching process can be regarded as the expansion of the DataTable starting from a one-column DataTable containing the root entities. An attribute stands for an instruction to find the edges from a source nodeset to a target nodeset. When scanning a new attribute in the subgraph matching process, the source nodeset comes from a column in the current DataTable, and the found target nodeset will be appended to the DataTable as a new column.

Let $\mathbf{K}^{(i)}$ be the recorded DataTable after scanning the ith attribute, which is a matrix with the size $k_i \times i$, and k_i is the number of matched subgraph prefixes. In particular, $\mathbf{K}^{(1)}$ is with size $n \times 1$, and $\mathbf{K}^{(p)}$ is with size $nm \times p$, in which $k_1 = n$ is the size of root entity set, and $k_p = nm$ is the total number of subgraphs. In scanning the ith attribute, the source nodeset is from the $f(i)$th column, and each current subgraph is to be expanded to one or more new subgraphs according to the number of target nodes linked to the corresponding source node.

The following two sequences of arrays are defined to describe the attribute scanning process, which is useful in the following analysis of complexity. Let $\mathbf{n}^{(i)}$ be the non-repeated nodeset of the ith attribute. Specifically, the size of the root nodeset $|\mathbf{n}^{(1)}| = n$. Let $\mathbf{q}^{(i)}$ be the number of succeeding edges in scanning the ith attribute for each node in $\mathbf{n}^{(f(i))}$, with $|\mathbf{q}^{(i)}| = |\mathbf{n}^{(f(i))}|$.

Specifically, if some source nodes do not have succeeding edges, the subgraphs are kept in the DataTable as partially-matched subgraphs by appending a null node. This means that the minimum value in $\mathbf{q}^{(i)}$ is 1.

Fig. 8 is a diagrammatic example of the subgraph matching process.

- Fig. 8(a) shows a template with 3 attributes. In this case, the prefix indices are $f(2) = 1$ and $f(3) = 2$.
- Fig. 8(b) shows the graph data to be matched.
- Fig. 8(c) shows the change of DataTable $\mathbf{K}^{(i)}$ in the subgraph matching process.
- Fig. 8(d) shows the arrays for describing the attribute scanning process. It is seen that each $\mathbf{n}^{(i)}$ is a description for the target nodeset of the ith attribute. Each $\mathbf{q}^{(i)}$ is a description for the edges found by the ith attribute, which is between the nodesets $\mathbf{n}^{(f(i))}$ and $\mathbf{n}^{(i)}$.

Based on the above definitions, the comparison of the complexity of the two algorithms is provided in Theorem 1 below.

Theorem 1. $e \le nmp$.

Proof. In the matching-checking method, let $\mathbf{k}^{(i,j)}$ be the jth column of $\mathbf{K}^{(i)}$, in which each item is from $\mathbf{n}^{(j)}$ and may repeat several times. Let $\mathbf{r}^{(i,j)}$ be the array of repeating times for the nodes in $\mathbf{k}^{(i,j)}$, and the size $|\mathbf{r}^{(i,j)}| = |\mathbf{n}^{(j)}|$. In scanning the ith attribute, for each node in $\mathbf{k}^{(i-1,f(i))}$, every succeeding edge corresponds to a new matched subgraph prefix, so the number of rows k_i is

$$k_i = \mathbf{r}^{(i-1,f(i))} \cdot \mathbf{q}^{(i)}. \tag{1}$$

The partially-matched subgraphs are kept in the DataTable, which keeps $k_{i-1} \le k_i$.

In the MVDLite algorithm, since the value constraints are evaluated during the searching, some of the nodes are excluded before scanning the next attribute. Let $\mathbf{h}^{(i)}$ be a mask array for $\mathbf{n}^{(i)}$, in which kept nodes are marked with 1 and excluded nodes are marked with 0. In scanning the ith attribute, the number of visited edges e_i is

$$e_i = \mathbf{h}^{(f(i))} \cdot \mathbf{q}^{(i)}. \tag{2}$$

Specifically, with regarding the root entity set as the first attribute, $e_1 = k_1 = n$. Since every item in $\mathbf{r}^{(i-1,f(i))}$ is not less than 1, and every item in $\mathbf{h}^{(f(i))}$ is not greater than 1, then $e_i \le k_i$. As a result,

$$e = \sum_{i=1}^{p} e_i \le \sum_{i=1}^{p} k_i \le \sum_{i=1}^{p} k_p = nmp. \quad \square \tag{3}$$

Theorem 1 indicates that the complexity of the MVDLite algorithm is not greater than the matching-checking method. The two "$\le$" signs in Eq. (3) imply that in the worst cases, the complexities are equal. The conditions that the two "$\le$" signs get equal can be summarized as the following points.

(a) Each root entity matches only one subgraph.
(b) No repeated nodes among multiple matched subgraphs.
(c) No excluded nodes by any value constraint before the last segment of a path.

The first "$\le$" sign gets equal only when every item in $\mathbf{r}^{(i-1,f(i))}$ and $\mathbf{h}^{(f(i))}$ equals 1, which corresponds to conditions (b) and (c), respectively. The second "$\le$" sign gets equal only when every $k_i = k_p = n$, which corresponds to condition (a).

In real-world models, the above three conditions are often violated. Three corresponding examples are given as follows.

(a) One root IfcObject node refers to multiple properties, and each property corresponds to a matched subgraph.
(b) Many root IfcObject nodes refer to a single IfcTypeObject node, and the single node would appear in multiple matched subgraphs.
(c) One root IfcObject node refers to multiple properties in different property sets, but a filter rule keeps only the properties in a certain property set.

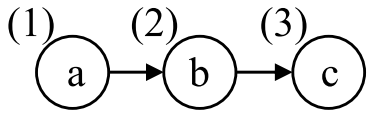
As a result, the two "$\le$" signs usually make significant differences in real-world MVD validation tasks, which remarkably speeds up the calculation.

5.2. The applicable scope of the MVDLite algorithm

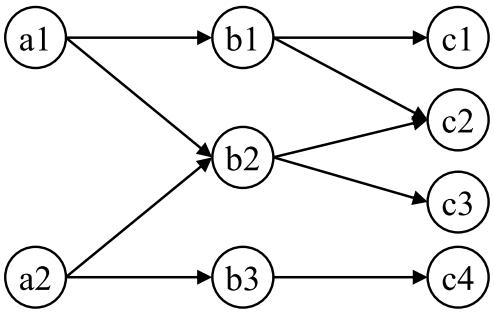
The speeding-up strategies in the MVDLite algorithm can be summarized as the following three points.

(a) **Reordered node searching and filtering operations.** The value constraints are evaluated during the searching so that some nodes can be excluded before scanning the next attribute.
(b) **Logical interconnections on lowest-common-ancestors.** The interconnections of attributes are performed as the corresponding intersection, union, or complement operations on the lowest-common-ancestor nodeset of the branches.
(c) **Merged nodeset.** The redundant nodes in multiple subgraphs are merged into one nodeset in the rule chain, which reduces the calls for node filtering.

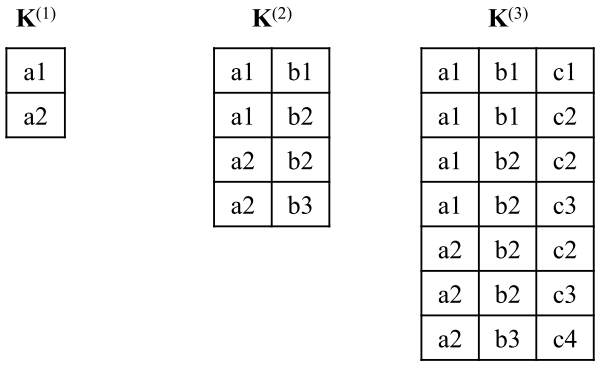
Strategies (a) and (b) rely on the tree structure of the subgraph template in the mvdXML ruleset, and the fact that the checking results of all the subgraphs need to be pushed back to get the result of each root entity. A tree-shaped template ensures the following conditions that support the round-trip searching on the rule chain. In the forward-searching stage, a tree-shaped template ensures that the branch pruning can be performed. In the backward tracing stage, when a node has succeeding paths that can pass all the rule segments, it must be on the path tracing back to the root entities. For two attributes, the



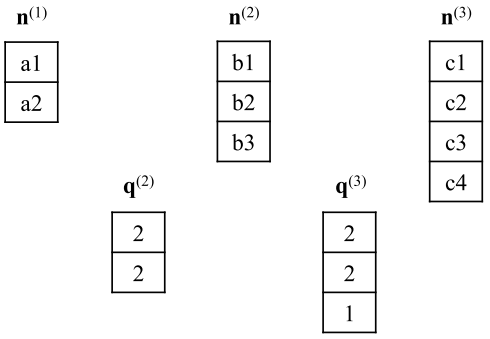
(a) The subgraph template.



(b) The graph data.



(c) Change of the DataTable $\mathbf{K}^{(i)}$ in the subgraph matching process.



(d) The arrays for describing the attributes and edges.

Fig. 8. A diagrammatic example of the subgraph matching process.

interconnection can be performed on the sub-subgraphs rooted with the lowest-common-ancestor nodeset, and the result can then be pushed back to the upper root entities.

Strategy (c) relies on the current fact that the MVD rules ignore the order of nodes in a node collection, and ignore the repeating times of a node in different matched subgraphs. In the node searching according to an attribute name in the mvdXML template, an attribute that can match multiple succeeding nodes in different subgraphs may be in either of the following cases.

- An attribute defined as a "SET" or a "LIST" of nodes in the IFC schema.
- An attribute defined as a nested collection of nodes in IFC schema, such as "LIST OF LIST OF ...".
- An inverse attribute in IFC schema, and the current node can be referred to in multiple source nodes.

The speeding-up strategies of MVDLite closely rely on the above-mentioned characteristics of the mvdXML ruleset, and such characteristics are based on the origin of MVD rules from the IDM information units. An information unit is the natural language description of the data exchange requirement of a domain entity or an attribute of a domain entity [21]. The domain entity corresponds to a root nodeset in MVD rules, and the attribute corresponds to a path starting from the root node to reach the required attribute node. Hence the path patterns are naturally combined into a tree-structured subgraph template. In subgraph matching, the order of searching different branches is not important, and the found subgraphs are without an order. Hence the merged unordered nodeset is applicable in the algorithm.

The mvdXML ruleset format is designed to fit the validation tasks of the IDM information units. The nested XML structure in "<ConceptTemplate>" makes the template be in a tree structure in most cases. However, the two exception cases "RuleID overloading" and "multi-RuleID clause" may result in a non-tree structure.

The "RuleID overloading" allows that one same RuleID may be assigned to different branches in the template, which represents several alternative paths in searching a nodeset. Some operations can be used to handle this case. First, a complex template with RuleID overloading can be split into a collection of optional templates in the tree shape. Second, the alternative paths in searching a nodeset may also be represented with the compound attribute segment in the rule chain structure. The first operation requires that the metrics can be independently evaluated in the collection of matched subgraphs for each optional template. The second operation requires that the node searching and filtering on each branch of the compound attribute segment are independent.

The "multi-RuleID clause" allows that on the right side of the operator, not only the values but also the RuleID-metric pairs can be used, such as "Inst_ObjectType[Value] = Type_ObjectType[Value]" for checking that the "ObjectType" of an instance node is the same as that of its type node. This can be handled by regarding the clause as a filter for the lowest-common-ancestor nodeset of the branches, but only when the node searching and filtering on each branch are independent.

Currently, although some extreme cases can be constructed, most of the domain concept definitions and data constraints can be represented in or converted into tree-shaped templates. The rules about ordered repeatable node collections are not involved in the current mvdXML format, and the searching and filtering results on the subgraphs are always pushed back to get the result of each root node, hence the fast searching and filtering strategies in MVDLite are applicable. The mvdXML method is still in evolution. The discussions in this section intend to provide some ideas for the future development of MVD technology, so that the fast validation of industrial data models can keep being supported when new features are added.

6. Conclusion and future work

The MVDLite algorithm proposed in this paper uses the rule chain structure to integrate the data templates and the value constraints from the mvdXML ruleset. By performing the round-trip searching on the rule chain structure, the MVDLite algorithm remarkably speeds up MVD validation. In a typical task, the time usage may reduce from minutes to seconds, or from around an hour to several minutes. The comparison between the complexity of the MVDLite algorithm and that of the matching-checking method is provided. The complexity analysis supports that the MVDLite algorithm is faster in most real-world cases with fewer redundant visits of edges.

The speeding-up strategies of the MVDLite algorithm are based on the characteristics of the mvdXML ruleset, which can be summarized as rooted tree-shaped subgraph templates, and ignoring the order and repetition of nodes in subgraph matching. Such characteristics of the mvdXML ruleset fit the validation tasks for the IDM information units, in which domain concepts and attributes are mapped to root nodesets and attribute paths, respectively. The speeding-up of the MVD validation tasks is beneficial in promoting the IDM-MVD method in the industry. A fast checking of data structure and information sufficiency is helpful in the data exchange to support downstream BIM-based applications.

Based on the discussions of the proposed fast MVD validation algorithm, there are several potential research topics for future work.

The first topic is to explore the applicability of the MVD technology in more flexible scenarios. The MVD ruleset plays a role as a bridge between the domain concepts in natural language and the raw data structure in the IFC schema. With the ability of fast finding the nodeset according to the applicability rules and fast accessing the required attributes according to the paths, the MVD validation algorithm may also be used in some more flexible scenarios, such as information query and partial data extraction from the model. Such applications may be helpful in the data preparation for the downstream BIM-based software tools.

The second topic is to extend the algorithm to linked building data applications. With an observation on the SHACL rules, it has a similar subgraph template structure as the mvdXML ruleset with root nodes and attribute paths. It may be a potential research topic whether a formal mapping can be found between mvdXML and SHACL, in order to apply the speeding-up strategies on the validation of linked building data.

In addition, the discussions on the speeding-up strategies intend to bring some new ideas to the community about the future evolution of the MVD technology, as well as other technologies for fast validation of industrial data models. The borderline between fast data conformance validation algorithms and powerful enriched checking and inferencing algorithms may be defined in future research, so that the speeding-up strategies can be kept, and the two types of algorithms can collaborate to support various applications.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Except for part of the confidential data, the links to other data resources have been added. The code is not permitted to be shared for now, but would be ready for open source in the future

Acknowledgments

This work was supported in part by the 2019 MIIT Industrial Internet Innovation and Development Project “BIM Software Industry Standardization and Public Service Platform” and the National Key Research and Development Program of China (2021YFB1600303).

References

[1] buildingSMART, Industry foundation classes IFC4 official release, 2013, Available from: https://standards.buildingsmart.org/IFC/RELEASE/IFC4/FINAL/HTML/ (accessed January 2022).
[2] J. Wix, J. Karlshoej, Information Delivery Manual: Guide to components and development methods, 2010, Available from: https://standards.buildingsmart.org/documents/IDM/IDM_guide-CompsAndDevMethods-IDMC_004-v1_2.pdf (accessed January 2022).
[3] R. See, J. Karlshoej, D. Davis, An integrated process for delivering IFC based data exchange, 2012, Available from: https://standards.buildingsmart.org/documents/IDM/IDM_guide-IntegratedProcess-2012_09.pdf (accessed January 2022).
[4] J. Hietanen, IFC Model View Definition Format, Technical document, International Alliance for Interoperability, 2006.
[5] buildingSMART, MVD database, 2022, Available from: https://technical.buildingsmart.org/standards/ifc/mvd/mvd-database/ (accessed January 2022).
[6] Y.-C. Lee, C.M. Eastman, W. Solihin, An ontology-based approach for developing data exchange requirements and model views of building information modeling, Adv. Eng. Inform. 30 (3) (2016) 354–367.
[7] S. Son, G. Lee, J. Jung, J. Kim, K. Jeon, Automated generation of a model view definition from an information delivery manual using idmXSD and buildingsmart data dictionary, Adv. Eng. Inform. 54 (2022) 101731.
[8] M. Weise, mvdXML requirements and examples: review of a standardized format to define and exchange model view definitions with exchange requirements and validation rules, 2014, Available from: https://github.com/BuildingSMART/mvdXML/tree/master/mvdXML1.1 (accessed January 2022).
[9] T. Chipman, T. Liebich, M. Weise, mvdXML: Specification of a standardized format to define and exchange model view definitions with exchange requirements and validation rules, 2016, Available from: https://standards.buildingsmart.org/MVD/RELEASE/mvdXML/v1-1/mvdXML_V1-1-Final.pdf (accessed January 2022).
[10] P. Pauwels, D. Van Deursen, R. Verstraeten, J. De Roo, R. De Meyer, R. Van de Walle, J. Van Campenhout, A semantic rule checking environment for building performance checking, Autom. Constr. 20 (5) (2011) 506–518.
[11] P. Pauwels, S. Zhang, Semantic rule-checking for regulation compliance checking: An overview of strategies and approaches, in: 32rd International CIB W78 Conference, 2015.
[12] T.H. Beach, Y. Rezgui, H. Li, T. Kasim, A rule-based semantic approach for automated regulatory compliance in the construction sector, Expert Syst. Appl. 42 (12) (2015) 5219–5231.
[13] H. Zhang, W. Zhao, J. Gu, H. Liu, M. Gu, Semantic web based rule checking of real-world scale BIM models: a pragmatic method, in: International Congress and Conferences on Computational Design and Engineering (I3CDE), 2019, pp. 130–137.
[14] P. Hagedorn, M. König, Rule-based semantic validation for standardized linked building models, in: Proceedings of the 18th International Conference on Computing in Civil and Building Engineering: ICCCBE 2020, Springer, 2021, pp. 772–787.
[15] C. Eastman, J.-m. Lee, Y.-s. Jeong, J.-k. Lee, Automatic rule-based checking of building designs, Autom. Constr. 18 (8) (2009) 1011–1033.
[16] xBimTeam, XbimMvdXML, 2016, Available from: https://github.com/xBimTeam/XbimMvdXML (accessed January 2022).
[17] opensourceBIM, mvdXMLChecker, 2014, Available from: https://github.com/opensourceBIM/mvdXMLChecker (accessed January 2022).
[18] J. Oraskari, C. Zhang, mvdXML Checker, 2021, Available from: https://github.com/jyrkioraskari/mvdXMLChecker (accessed January 2022).
[19] opensourceBIM, python-mvdxml, 2019, Available from: https://github.com/opensourceBIM/python-mvdxml (accessed May 2023).
[20] International Standards Office, ISO 10303-11:2004 Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 11: Description Methods: The EXPRESS Language Reference Manual, ISO, Geneva, Switzerland, 2019.
[21] International Standards Office, ISO 29481-1:2016 Building Information Models - Information Delivery Manual - Part 1: Methodology and Format, ISO, Geneva, Switzerland, 2019.
[22] International Standards Office, ISO/TS 15926-11:2023 Industrial Automation Systems and Integration - Integration of Life-Cycle Data for Process Plants Including Oil and Gas Production Facilities - Part 11: Simplified Industrial Usage of Reference Data Based on RDFS Methodology, ISO, Geneva, Switzerland, 2023.
[23] G. Lee, Y.H. Park, S. Ham, Extended Process to Product Modeling (xPPM) for integrated and seamless IDM and MVD development, Adv. Eng. Inform. 27 (4) (2013) 636–651.

[24] M. Venugopal, C. Eastman, R. Sacks, Configurable model exchanges for the precast/pre-stressed concrete industry using semantic exchange modules (SEM), in: R.R. Issa, I. Flood (Eds.), International Conference on Computing in Civil Engineering, ASCE, 2012, pp. 269–276, http://dx.doi.org/10.1061/9780784412343.0034.

[25] M. Venugopal, C.M. Eastman, J. Teizer, An ontology-based analysis of the industry foundation class schema for building information model exchanges, Adv. Eng. Inform. 29 (4) (2015) 940–957.

[26] M. Weise, P. Katranuschkov, R.J. Scherer, Generalised model subset definition schema, in: CIB W78's 20th International Conference on Construction IT, CIB, ISBN: 0908689713, 2003, pp. 440–448.

[27] J. Beetz, J. Van Leeuwen, B. De Vries, IfcOWL: A case of transforming EXPRESS schemas into ontologies, Artif. Intell. Eng. Des. Anal. Manuf. 23 (1) (2009) 89–101.

[28] International Standards Office, ISO 15926-2:2003 Industrial Automation Systems and Integration - Integration of Life-Cycle Data for Process Plants Including Oil and Gas Production Facilities - Part 2: Data Model, ISO, Geneva, Switzerland, 2003.

[29] P. Pauwels, D. Shelden, J. Brouwer, D. Sparks, SahaNirvik, T.P. McGinley, Building and Semantics: Data Models and Web Technologies for the Built Environment, CRC Press, 2022, pp. 106–107.

[30] H. Knublauch, D. Kontokostas, Shapes constraint language (SHACL), 2017, Available from: https://www.w3.org/TR/shacl/ (accessed January 2022).

[31] International Standards Office, ISO 15926-10:2019 Industrial Automation Systems and Integration - Integration of Life-Cycle Data for Process Plants Including Oil and Gas Production Facilities - Part 10: Conformance Testing, ISO, Geneva, Switzerland, 2019.

[32] R.K. Soman, M. Molina-Solana, J.K. Whyte, Linked-Data based Constraint-Checking (LDCC) to support look-ahead planning in construction, Autom. Constr. 120 (2020) 103369.

[33] J. Oraskari, M. Senthilvel, J. Beetz, SHACL is for LBD what mvdXML is for IFC, in: Proc. of the Conference CIB W78, Vol. 2021, 2021, pp. 11–15.

[34] S. Stolk, K. McGlinn, Validation of IfcOWL datasets using SHACL, in: Proceedings of the 8th Linked Data in Architecture and Construction Workshop, 2020, pp. 91–104.

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

[36] W. Mazairac, J. Beetz, BIMQL - An open query language for building information models, Adv. Eng. Inform. 27 (4) (2013) 444–456.

[37] E. Tauscher, H.-J. Bargstädt, K. Smarsly, Generic BIM queries based on the IFC object model using graph theory, in: Proceedings of the 16th International Conference on Computing in Civil and Building Engineering, ICCCBE2016 Organizing Committee, ISBN: 9784990737122, 2016, pp. 905–912.

[38] C. Preidel, A. Borrmann, Integrating relational algebra into a visual code checking language for information retrieval from building information models, in: Proceedings of the 16th International Conference on Computing in Civil and Building Engineering, ICCCBE2016 Organizing Committee, ISBN: 9784990737122, 2016, pp. 454–461.

[39] J. Werbrouck, M. Senthilvel, M.H. Rasmussen, Building and Semantics: Data Models and Web Technologies for the Built Environment, CRC Press, 2022, pp. 150–151.

[40] Solibri, Solibri office, 2023, Available from: https://www.solibri.com/solibri-office (accessed May 2023).

[41] EXPRESS Data Manager, Jotne, 2023, Available from: https://jotneconnect.com/products/express-data-manager/ (accessed May 2023).

[42] Simplebim, Support site for Simplebim users - mvdXML, 2022, Available from: https://www.simplebim.com/support/addon-mvdxml.html (accessed January 2022).

[43] M. Weise, T. Liebich, N. Nisbet, C. Benghi, IFC model checking based on mvdXML 1.1, in: S.E. Christodoulou, R. Scherer (Eds.), EWork and EBusiness in Architecture, Engineering and Construction: ECPPM, CRC Press, ISBN: 9781315386898, 2016, pp. 19–26.

[44] C. Zhang, J. Beetz, M. Weise, Model view checking: automated validation for IFC building models, in: A. Mahdavi, B. Martens, R. Scherer (Eds.), EWork and EBusiness in Architecture, Engineering and Construction: ECPPM, CRC Press, ISBN: 9781315736952, 2014, pp. 123–128.

[45] Y.-C. Lee, C.M. Eastman, W. Solihin, R. See, Modularized rule-based validation of a BIM model pertaining to model views, Autom. Constr. 63 (2016) 1–11.

[46] Y.-C. Lee, C.M. Eastman, W. Solihin, Logic for ensuring the data exchange integrity of building information models, Autom. Constr. 85 (2018) 249–262.

[47] W. Solihin, C. Eastman, Y.-C. Lee, Toward robust and quantifiable automated IFC quality validation, Adv. Eng. Inform. 29 (3) (2015) 739–756.

[48] M.H. Rasmussen, M. Lefrançois, G.F. Schneider, P. Pauwels, BOT: the building topology ontology of the W3C linked building data group, Semant. Web 12 (1) (2021) 143–161.

[49] NIBS, Common building information model files and tools, 2012, Available from: https://www.wbdg.org/bim/cobie/common-bim-files (accessed January 2022).

[50] buildingSMART, IfcDoc/IfcKit/exchanges/, 2018, Available from: https://github.com/buildingSMART/IfcDoc/tree/master/IfcKit/exchanges (accessed January 2022).