
Extended Imperative Model Checking – A visual programming approach for a user-friendly MVD generation and validation

Felix Sirtl¹, Jimmy Abualdenien²

¹Chair of Computational Modeling and Simulation · Technical University of Munich · Arcisstraße 21 · 80333 Munich · E-Mail: felix.sirtl@tum.de

²Chair of Computational Modeling and Simulation · Technical University of Munich · Arcisstraße 21 · 80333 Munich · E-Mail: jimmy.abualdenien@tum.de

Abstract

Digital building models play a key role in the digitalisation of the building industry. In the Building Information Modeling (BIM) methodology, geometric and semantic information of the entire life cycle are mapped to a digital building model. This extended amount of information makes automatically controlling quality of the building information possible and essential. Industry Foundation Classes (IFC), the international standard in the field of digital building with OpenBIM, utilizes the Model View Definition (MVD) method for the purpose of automated quality assurance. Progressive extension of the MVD method and technical implementation by means of the digital format mvdXML led to the fact that domain experts have difficulties in understanding and using MVDs. This work aims to enable domain experts to create MVD functionalities on their own through a user-friendly approach. Therefore, a concept that makes use of the widely established visual programming is presented. The visual programming approach is based on a general concept called Extended Imperative Model Checking (EIMC). The concept allows to analyse, filter, and validate data of an IFC model. To prove the concept a prototype was developed. The prototype is called EIMC-VP and makes use of visual programming. The application is intended to make it easier for users to create and execute a query, respectively functionalities of an MVD, in EIMC. It employs the imperative query language QL4BIM to interact with a building model.

Keywords: Building Information Modeling (BIM), Model View Definition (MVD), Automated Quality Assurance

1 Introduction

Building Information Modeling (BIM) is a methodology that holistically attempts to map analogue processes of the construction industry digitally [1]. BIM is being established in more and more construction projects worldwide. Increasingly, Germany is also requiring BIM to be integrated into public construction projects [2]. Nevertheless, this establishment still presents companies with major challenges; small companies in particular are often overwhelmed by complex standardisations [3]. The implementation for schema-based reduction in form of Model View Definition (MVD) is one of these complex practices. The highly complicated standard results in a lack of software solutions that could make the application practicable for domain users.

MVDs have several tasks. In summary, they are the machine-readable notation of the information gathered in the Information Delivery Manual [1]. The Information Delivery Manual describes processes and information flows of a construction project [1]. For example, it specifies which group of people may have access to which information at what time. The requirements of this exchange information can then be technically specified by MVDs and enable an application-based reduction of information of the IFC schema. MVDs are designed in such way that all-encompassing functionalities of the IFC schema can be retained. For specifying and exchanging MVD information, the mvdXML format was developed. However, the universal approach leads to complex implementations through the mvdXML format. Zhang and colleagues [5] formulate this with the accusation that "mvdXML is a semi-structured description method rather than a logic-based strictly formatted method". The two-part structure of mvdXML files, through *Templates* and *Views*, attempts to enable reusability of individual templates, thereby causing a representation in which many different expressions can achieve similar results. Zhang and colleagues [5] further criticise the lack of clear and comprehensible documentation for creating mvdXMLs, which in turn makes access to the topic difficult. The complex methodology combined with the lack of software that supports the creation of MVDs in a user-friendly and sufficient way, leads to the fact

that by now only high-level experts are able to create MVDs. The consequence is that an additional hurdle has been placed on the path to the digitalisation of the construction industry. This paper presents a new concept which offers a user-friendly approach for the development of MVD functionalities through visual programming. It is called Extended Imperative Model Checking (EIMC).

2 Model View Definition - mvdXML

Nowadays, the conceptual idea of filtering the universal IFC schema to an application-specific required subset is carried out by means of the mvdXML format [4]. The goals of the format are, in addition to the pure determination of IFC subsets, the documentation of these, as well as the filtering and validation of IFC instance files. MvdXML files consist of two parts. In the first part, *Templates*, general and thus reusable concepts are stored. These contain structures of the IFC schema of required entities. The second part, *Views*, contains the case-related exchange requirements. In this part, restrictions and validations are made. The following Listing 1 shows an excerpt from a mvdXML file. This is structured hierarchically and after reference has been made to the previous *ConceptTemplates*, a more refined filtering of the schema can be carried out in the *Applicability* part. The concrete rules have been left out for the purpose of clarity (line 12).

```
1 <Views>
2   <ModelView uuid="72dad5df-6f61-49f2-ba8c-baccf24a6ce5" name="Sensor signal view" applicableSchema="Ifc4" code="Sensor">
3     <ExchangeRequirements>
4       <ExchangeRequirement uuid="a70f764-938b-4cf7-9814-c29a47f56b0e" name="Distribution signal" code="ERM1"
5         applicability="export"></ExchangeRequirement>
6     </ExchangeRequirements>
7     <Roots>
8       <ConceptRoot uuid="8b949664-a5df-4bfc-922c-4a486c41d756" name="Sensor" applicableRootEntity="IfcSensor">
9         <Applicability></Applicability>
10        <Concepts>
11          <Concept uuid="a4fa348c-a025-4a02-abfd-c42fd0901540" name="Port Assignment">
12            <!--Checking Task -->
13          </Concept>
14        </Concepts>
15      </ConceptRoot>
16    </Roots>
17  </ModelView>
18 </Views>
```

Listing 1: Section of the *View* part of an MVD (adopted from [4])

3 Extended Imperative Model Checking

In the scope of this work, the automatic quality assurance of building models was examined. The underlying paradigm of the work was always to enable the highest possible usability and intuitive application. For this purpose, we decided to analyse models using the imperative open-source query language QL4BIM [6].

3.1 QL4BIM as programming basis

QL4BIM makes it possible to analyse and filter a building model with just a few lines of code [6]. The language uses statements to declare variables with particular entities [6]. These variables can then be reused in subsequent statements to make further restrictions. Listing 2 shows how a model can be filtered for external walls in QL4BIM. In the first line, the `ImportModel` operator allows access to the building model. All entities of the building are stored in the variable `entities`. In line two, the `TypeFilter` operator filters this set of entities to those that belong to the type `IfcWall` and subtypes. The obtained entities are then filtered again in line three to select only those with the property assignment of `IsExternal` to get all external walls of the model.

```
1 entities = ImportModel ("C:\Testmodel.ifc ")
2 walls = TypeFilter ( entities is IfcWall )
3 ExternalWalls = PropertyFilter ( walls.IsExternal )
```

Listing 2: QL4BIM-code that filters an IFC model for external walls (adopted from [6])

3.2 EIMC Concept

Based on the intuitive language QL4BIM, we have developed a concept that aims to allow end users an easy generation of functionalities of an MVD. In the context of this work, we focus on filtering and validating the filtered elements of an IFC model. Figure 1 represents the schematic concept of EIMC. It starts with the import of an instance file. Afterwards, the imported file can be filtered with various operators. A separate validation of the filtered elements is possible. Then the results are output in a report.

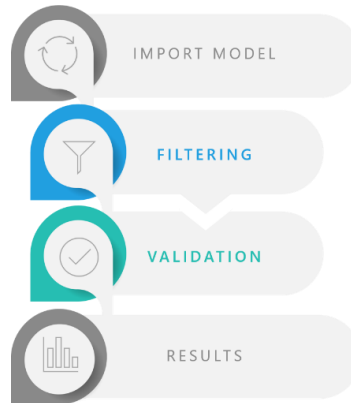


Figure 1: Schematic representation of the EIMC concept

3.3 Validation Operators

To implement the concept, we have implemented new validation operators into the language. These make it possible to check the values of attributes and properties for restrictions. In addition to the `AttributeCheck` and `PropertyCheck`, we have developed a combination of these two operators, the `XCheck` operator, for more simplification. This operator first performs a query comparison for attributes and then a query comparison for properties, which means that users have not to worry about IFC-specific contexts. The following Listing 3 first filters all entities of the model to only walls. Then, in line 3, the `XCheck` operator is applied. It allows access to an attribute or property of the selected variable using the dot operator. In the example, the attribute `Description` is checked. The query checks whether each wall has a description called `TestDescription`.

```
1 entities = ImportModel ("C:\Testmodel.ifc")
2 wall = Typefilter ( entities is IfcWall )
3 a = XCheck ( wall. Description = "TestDescription" )
```

Listing 3: EIMC code that filters an IFC model for walls and then checks if the filtered elements have an attribute or property called `Description` with the content `'TestDescription'`

Further overloads allow for more diverse usages and thus more flexible checks. For example, numerical values can be checked. These can be constrained to precise values as well as to numerical ranges. Listing 4 contains a boundary that makes it possible to define a range of values up to a certain number. In line 3, the thermal transmittance is checked to see if it is less than 1.3 units. The selection of elements to be checked was initially restricted to all walls by lines 1 and 2. In addition, it is possible to narrow the examination to a specific property set. The property set is enumerated by a second string-based parameter set. In line 3, the argument of the `XCheck` states out that only properties in the property set `PSet_WallCommon` are to be checked.

```
1 entities = ImportModel ("C:\Testmodel.ifc")
2 wall = Typefilter ( entities is IfcWall )
3 a = XCheck ( wall. ThermalTrancemittance < 1.3, "PSet_WallCommon" )
```

Listing 4: EIMC code that filters an IFC model for walls and then checks if the filtered elements have a property `ThermalTrancemittance` provided by the property set `PSet_WallCommon` with a value less than 1.3 units

3.4 EIMC compared to mvdXML

Due to the user-friendliness and thus deliberate restriction of EIMC in various purposes, there are advantages compared to an MVD created in `mvdXML` format. Likewise, the limitations also lead to a number of functionalities that can no longer be performed or can only be performed to a different extent.

Within the mvdXML schema the reusability of templates and concepts is of great importance due to a structure that can be used in many different ways. This leads to the fact that for a domain expert who has knowledge of IFC schema but no programming skills, it is almost impossible to understand the basic schema of an MVD created with mvdXML. In contrast, EIMC, with its imperative and clear style from QL4BIM, provides an easy entry point for domain experts. The Application EIMC-VP aims to simplify the usage even more and tries to give an entry point for domain experts with no programming skills. The complexity of the mvdXML schema inevitably means that the creation of new MVDs as well as the extension of existing MVDs is hardly possible for non-experts. EIMC and the resulting visual programming application EIMC-VP, which will be discussed in the following section, attempt to simplify the creation of features of an MVD. Besides, relationships within the IFC schema can relate to entities in a variety of ways. For example, a property set can be attached to an IFC entity via both occurrence and type. The mvdXML schema requires the specification of all possible relationships to determine whether certain entities are linked to other entities. In the EIMC all possible mappings are searched by using operators and users do not have to enter all possible mappings of a property set manually, due to implementations in QL4BIM. Furthermore, within a mvdXML rule only one path between entities can be executed. Therefore, only a set selection is available and geometrical queries that refer to multiple objects of different classes are not possible. Within EIMC queries can be created due to geometrical operators, which allow a wider range of geometric queries. Further information on geometrical operators can be found in chapter 6.5.5 in [6].

4 Extended Imperative Model Checking – Visual Programming

To ensure the basic objective of usability, we have created an application in which the concept of EIMC can be applied to an instance model using visual programming. The application is designed to enable domain experts to create automated quality checking. In order to offer an appealing and intuitive design, we decided to use the Blockly software development kit (SDK) [7]. The great adaptability of the SDK allows a detailed adjustment to the needs of the EIMC concept. Figure 2 contains an exemplary representation of a query assembled in a Blockly demo workspace.

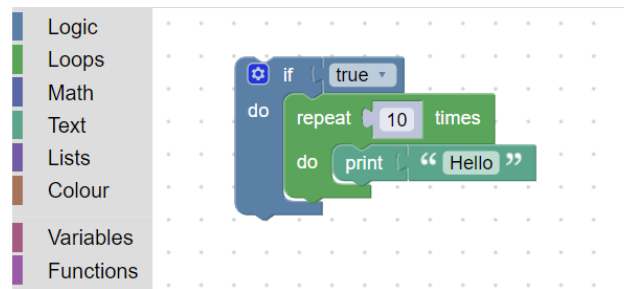


Figure 2: Blockly demo workspace with an assembled query

The individual blocks and structures can be adapted as required in Blockly's source code. We have implemented our own workspace and our own blocks to incorporate the concept in an effective way. To make it easier to use the language, we have also decided to use variables, which occur in EIMC, only indirectly in the visual programming language. This works by building representations of QL4BIM statements with visual blocks in horizontal rows. Statements always vertically build on each other and use the previous variable as input for the next statement until a new block of statements is introduced. By default, the resulting block structures are used for each check. Figure 3 illustrates the conceptual structure.

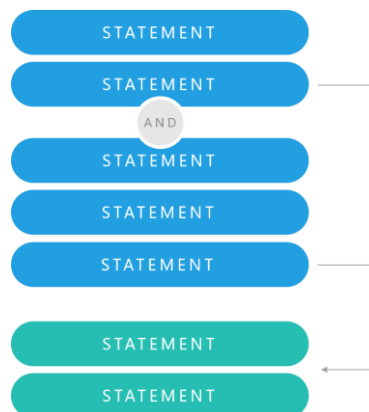


Figure 3: Schematic representation of the EIMC-VP concept

4.1 Detailed look at EIMC-VP

The layout of the prototype is divided into two parts. On the one hand, there is the toolbox area (Figure 4 part 1), in which various tabs contain blocks sorted by category, and on the other hand, the workspace area (Figure 4 part 2), in which the blocks can be added via drag and drop. The workspace is the place where queries are created. In the following Figure 4, the Start tab is already opened (Figure 4 part 1). It contains structural blocks for a query. A query always starts with the ImportModel block. Thereby, the IFC model is loaded into the context of the query interpreter. Subsequently, one or more Applicability blocks can be placed below each other, in which the filtering of the model takes place. An Applicability block always represents a self-contained analysis of the current IFC model. If, for example, several types are to be examined separately from each other, several Applicability blocks are required. In the workspace area (Figure 4 part 2) a simple query is assembled. It checks whether all windows of the model have a thermal transmittance that is less than 1.3 units. After a query has been created and a model has been loaded via the ImportModel block, the query can be executed via the Execute-button. A JSON report is then automatically generated, listing all entities that did not pass the check.

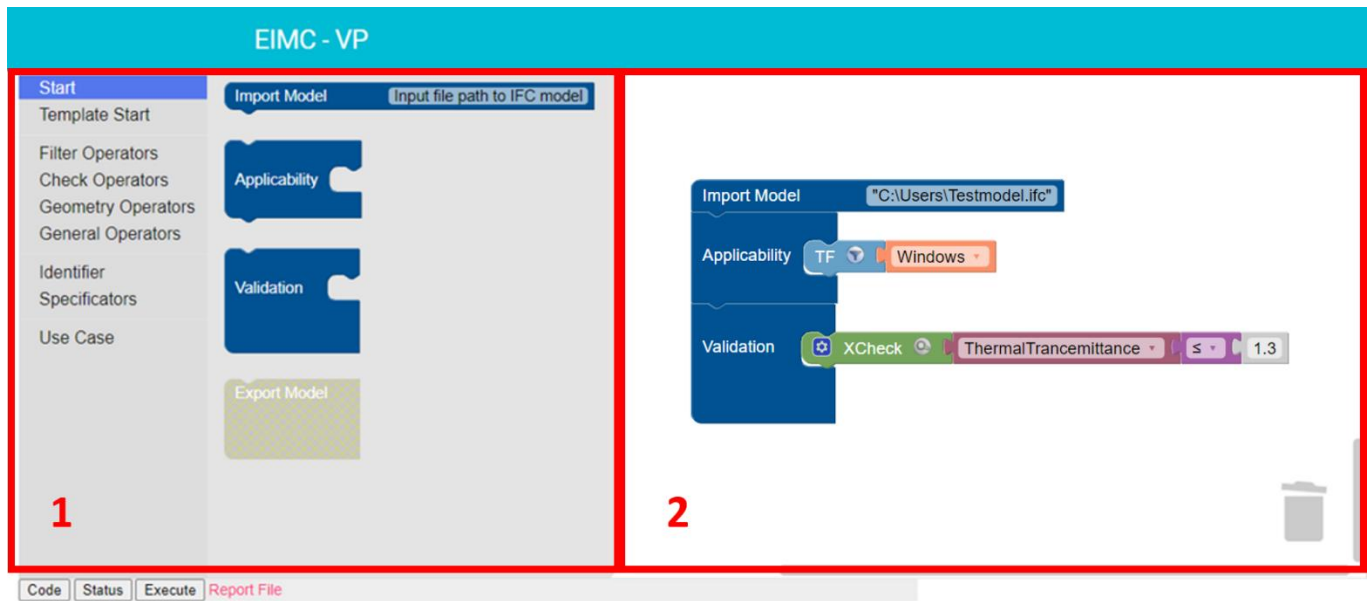





Figure 4: Toolbox and workspace area of EIMC-VP

The structural blocks (ImportModel, Applicability, Export Model and Validation) provide the basic framework for the query. Within these blocks, statements are built up to define how a model should be analysed. Each statement starts with an operator that evaluates the model for specific actions. The argument of the operator is constructed horizontally with Identifiers and logical Specification blocks to be able to specify the action of the operator. A selection of different blocks, their functionalities, and corresponding names in EIMC is listed in the following table 1. Operator blocks are additionally highlighted in which part they can be used, whereas Identifier blocks and logical Specification blocks can be used in both the *Applicability* parts and the *Validation* part.

Block in EIMC-VP	Name	Description
Operators		
	TypeFilter	Filters entities from previous statement to specific types. The types are declared in the horizontally next block. Utilisation in <i>Applicability</i> parts.
	PropertyFilter	Filters the inputted entities from previous statement to specific properties or property sets. The properties are declared in the horizontally connected blocks. Utilisation in <i>Applicability</i> parts.
	AttributeFilter	Filters the inputted entities from previous statement to specific attributes. The properties are declared in the horizontally connected blocks. Utilisation in <i>Applicability</i> parts.











	XCheck	Checks the last variable of each <i>Applicability</i> part. The gearwheel button enables specification for certain <i>Applicability</i> parts. Both properties and attributes can be checked. The check is determined by horizontally connected blocks. Utilisation in the <i>Validation</i> part.
	TouchOperator	The TouchOperator is used as a representative example for various geometric operators. It combines touching entities from previous statements into a relation. Utilisation in <i>Applicability</i> parts.
Identifiers		
	TypeLiteral	Selection of different IFC entities in a drop-down menu. Last selection allows to enter own string into the field. Usually attached horizontally to the TypeFilter to specify a type.
	AttributeLiteral	Selection of different IFC attributes in a drop-down menu. Last selection allows to enter own string into the field.
	PropertyLiteral	Selection of different IFC properties in a drop-down menu. Last selection allows to enter own string into the field.
	PSetLiteral	Selection of different IFC property sets in a drop-down menu. Last selection allows to enter own string into the field.
Specificators		
	BooleanSpec	Drop down menu with the choice of <i>true</i> or <i>false</i> . Besides true and false, it is possible to select the value <i>exists</i> . Concludes a statement.
	ComparisonSpec	Drop down menu with a selection of comparison operators.
	IntegerSpec	Drop down menu with a selection of integers. Concludes a statement.
	Separator	Allows to specify input variables for operators that need more than one input source. The output variable of the statement above the Separator is used as input variable for the next multiple input operator. The statement below a Separator begins, like a statement at the beginning of the <i>Applicability</i> part, with all entities of the IFC model as input variables.

Table 1: Selection of important blocks in EIMC-VP

4.2 Use Case

Figure 5 shows a query assembled in EIMC-VP. First, all entities of an IFC model are imported by using the ImportModel block. These entities are passed to both *Applicability* parts and act as input variables of each first statement. The first *Applicability* part filters all entities to all instances of IfcWindow using the TypeFilter combined with the TypeLiteral block (1). The second *Applicability* part starts again with all entities of the IFC model as input variable for the first statement. The first operator restricts these entities to all instances of IfcWall using the TypeLiteral block. The following PropertyFilter in the next statement specifies the selection of the instances further. The output of the PropertyFilter contains all instances of IfcWall that have a property LoadBearing with the corresponding value true (2). The following Separator block indicates that an operator, which needs multiple input sources, appears in the sequence. In the following example, this is the TouchOperator. The output variable of the statement directly above the Separator block is stored in a hidden background variable (2).

The following statement starts again with all entities of the IFC model as input variable. From these, all instances of IfcSlab are selected, the output variable is stored in a background variable as well (3). The TouchOperator now uses the two stored background variables as input variables and creates a relation of entities touching each other (4). The created relation lists pairs of touching walls and slabs. The generated relations are stored in the output variable of the second *Applicability* part (5). The following *Validation* part contains two XCheck statements. These two are processed independently of each other. The first check has one integer node attached. Therefore, the check is just applied to the *Applicability* part declared within the integer block (6). The first XCheck is only applied to the first *Applicability* part. The first *Applicability* part provides all instances of the class IfcWindow. It is checked whether the property FireRating exists by the instances of IfcWindow. Then, the second XCheck follows. It performs two individual checks, since two attached integer nodes declare that both the first and the second *Applicability* part are checked (the same result would be achieved if no integer node were attached at all/ for a better explanation of the integer nodes, the more detailed variant was chosen in this example). The first of the two performed checks refers to *Applicability* part one and thus checks all windows whether the property ThermalTrancemittance exists (7). The second performance of the second check refers to the entities obtained from *Applicability* part two. This provided variable in *Applicability* part two is a relation and not a set of entities. In the case of the given relation, both the selected load bearing walls and the selected slabs are checked (5). So, the load bearing walls and the touching slabs are checked for the existence of the ThermalTrancemittance property.

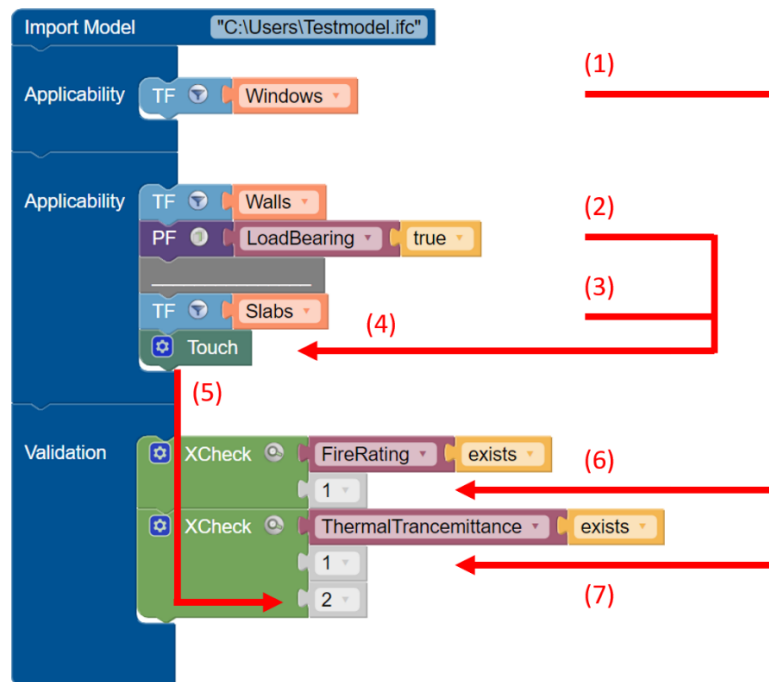


Figure 5: Assembled query in EIMC-VP

5 Technical implementations

The technical processes and implementations of the EIMC-VP application are briefly described below. The entire code is open source and can be found on GitHub [8]. In the frontend users assemble blocks in EIMC-VP to a query and validation. To execute this, the data entered by users is first sent to the backend as intermediate code. The intermediate code is a JSON format. In the backend the intermediate code is converted into EIMC code. Then the code is forwarded to the QL4BIM Extended server, where the validation is executed on the IFC model. A check report is sent back to the frontend with a validation report. Users can eventually download the report file via a download link in the frontend. The individual blocks are programmed in JavaScript. Every block has an own code generator. This converts the entire query into a textual intermediate code. In the backend the textual representation of the intermediate code is converted back into an object-orientated representation by using Python's JSON-to-Object conversion. This is done in the programmed QL4BIM-Extended-Code-Generator. The algorithm of the QL4BIM-Extended-Code-Generator traverses the tree structure. Certain objects, which are nodes in the tree structure, trigger certain generation of QL4BIM Extended statements. The result is a valid QL4BIM Extended code. This code is then forwarded to the QL4BIM Extended Server. In order to be able to reproduce full functionality, the check operators were implemented in the existing QL4BIM server. These are programmed using C# and are based on AttributeFilter and PropertyFilter operators. By facilitating the filter operators, specific entities can be checked. The selection of the failed elements as well as the integration of these into the check report were also implemented. Listing 5 shows the C# programming for achieving this.

```

69 private void AttributeCheckLocal(QEntity entity, PredicateNode predicateNode)
70 {
71     var checkResult = AttributeSetTestLocal(entity, predicateNode);
72     if (!checkResult.Item1)
73     {
74         //return tuples: result of check, prop does exit, value preset with [0]=type, [1]=value, [2]=unnested
75         var checkResultMeta = checkResult.Item3;
76         var value_report = $"property exists: {checkResult.Item2}, actual type {checkResultMeta[0]},
77             actual value: {checkResultMeta[1]}, unnested: {checkResultMeta[2]}";
78         reportWriter.AddEntityIdMessageToCheckEntry(entity, predicateNode.ToString(), value_report);
79     }
80 }

```

Listing 5: Adding results of the check to the report file of the AttributeCheck

6 Summary and Outlook

In practice the amount of information stored in building models is constantly increasing. Therefore, the relevance of automated quality assurance is rising as well. The MVD method of buildingSMART establishes a formal specification of model filtering and validation. However, due to the complexity of the format and lack of intuitive software it is not possible for domain experts to create and use MVDs with reasonable effort for a certain use case. To overcome this gap, the EIMC-VP concept provides an intuitive approach to filter and validate IFC models. The described approach is based on EIMC, an imperative style of model checking and combines it with visual programming. Two subsequent strategies exist to further improve EIMC-VP and to simplify the creation of advanced model checks. The first strategy is to include IFC schema information into the EIMC-VP system. Thereby, the domain expert can examine the appropriate IFC schema version directly in the application. The second strategy is to automatically guide the user while creating model checks so that all IFC class/attribute names and enum values are presented to the user. This can be extended as well, so that the current query/validation is considered. For example, after a reduction to walls, only attributes of that type and subtype can be chosen in a subsequent AttributeFilter.

Bibliography

- [1] Borrmann, A., König, M., Koch, C., & Beetz, J. (Hrsg.) (2015). Building Information Modeling Technologische Grundlagen und industrielle Praxis. Wiesbaden: Springer-Verlag.
 - [2] Bundesministerium für Verkehr und digitale Infrastruktur. (2015). Stufenplan Digitales Planen und Bauen. [Online] Available: https://www.bmvi.de/SharedDocs/DE/Publikationen/DG/stufenplan-digitales-bauen.pdf?__blob=publicationFile
 - [3] Mittelstand-Digital. (2018). Digitalisierung der mittelständischen Bauwirtschaft in Deutschland - Statusevaluation und Handlungsempfehlungen. [Online]. Available: https://kommunikation-mittelstand.digital/content/uploads/2018/10/Status-Quo_Digitalisierung_Bauwirtschaft.pdf.
 - [4] Chipman, M., Liebich, T., & Weise, M. (2016). mvdXML: Specification of a standard-ized format to define and exchange Model View Definitions with Exchange Re-quirement and Validation Rules. [Online]. Available: https://standards.buildingsmart.org/MVD/RELEASE/mvdXML/v1-1/mvdXML_V1-1-Final.pdf
 - [5] Zhang, C., Beetz, J., & Weisen, M. (2015). Interoperable validation for IFC building models using open standards. Journal of Information Technology in Construction (ITcon), 20(2), 24-39.
 - [6] Daum, S. (2018). Konzeption einer raum-zeitlichen Anfragesprache für die Analyse und Prüfung von 4D-Gebäudeinformationsmodellen. (Doctoral dissertation, Technische Universität München).
 - [7] Blockly. (2020). Introduction to Blockly. [Online]. Available: <https://developers.google.com/blockly/guides/overview>
 - [8] Sirtl, F., Daum, S., & Abualdenien, J. (2018). EIMC. [Online]. Available: <https://github.com/FelixSirtl/EIMC>
-