An XML-based approach for the exchange of application defined algorithms in building information models

Julian Amann, julian.amann@tum.de
Technical University of Munich, Germany

André Borrmann, andre.borrmann@tum.de
Technical University of Munich, Germany

Abstract
This paper describes a dynamic XML-based approach for the exchange of application defined algorithms in building information models. The demonstrated approach is based on XML technology, in particular XQuery, that allows the exchange of methods (in the object oriented sense of the term) instead of mere plain data. One aim of this approach is to reduce the complexity of building information models. To prove this, we introduce an instance-level metric to measure the complexity of building information models. Using the example of transition curves for alignment design, we describe how complexity can be reduced using our approach.

Keywords: Building Information Modeling, Model Based Management Tools and Systems, Data, Information and Knowledge Management, Infrastructure, XML

1 Introduction

As the complexity of building information models increases, it becomes more and more difficult for software vendors to adapt the corresponding, fast evolving data models. In this paper, we propose a dynamic building model based on XML technology that reduces this burden.

In the first part of our paper we describe an approach with which object oriented software components can be embedded in a building information model (we call this a dynamic building information model). The demonstrated approach is based on XML technology (in particular XQuery (Chamberlin 2003)) that allows the exchange of methods (in the object oriented sense of the term) instead of mere plain data. With this approach we can increase the interpretability of a building information model since we are not exchanging simple data but intelligent methods. We decided to use XML here, because XML is widely established in computer science and can be seen as the de-facto standard for data modeling and exchange in almost all IT domains. Also, most data models from the infrastructure domain are XML-based (such as RoadXML, LandXML, upcoming encoding of InfraGML, OKSTRA, etc.) or have a corresponding valid mapping (for example, buildingSmart has defined a mapping from EXPRESS to XSD). Furthermore, XML has great software support. Almost every programming language offers XML parsers, schema validators and XQuery evaluation.

In the second part of the paper, we consider the complexity of existing building information models. The complexity of building information models can be measured using different software metrics borrowed from object oriented design. For instance, the average depth of inheritance of an IFC schema can be evaluated and compared to different IFC Schema versions. We also introduce a more sophisticated approach to measuring the complexity of a building information model by making comparisons at an instance level. We will show how to apply this complexity metric to several building information models such as LandXML (LandXML.org 2016), IFC (buildingSMART 2016) or OKSTRA (Bundesanstalt für Straßenwesen 2016) for a use case in the infrastructure domain.

Finally, we show a use case, from the infrastructure domain, comparing our approach to the conventional way and showing how implementation complexity can be significantly reduced using our approach.

2 Data exchange in BIMs today

When analyzing existing and widely used BIM data models such as IFC (buildingSMART 2016), LandXML (LandXML.org 2016), or OKSTRA (frequently used in German speaking countries), the following core concepts form the foundation of these models:

- Simple, well-defined types such as characters, strings, decimals, integers, reals
- Complex types such as lists, arrays, enumerations
- Self-defined composite data types that make use of simple, complex as well as other self-defined data types

Existing BIM data models also implement object-oriented concepts of inheritance, and some even support object-oriented methods and procedural functions that can compute derived attribute values from the available attributes. An example of this is shown in Listing 1. The attribute *P* in the listing is a derived attribute that is computed from the reference direction (*RefDirection*). The data model contains a description of how to compute this attribute using a function (*IfcBuild2Axes*). This function is also defined in the corresponding EXPRESS schema. Besides methods that can compute values, EXPRESS also permits one to define assertions to ensure data integrity, encoded in the corresponding *WHERE* block.

Listing 1 IFC 4 EXPRESS definition of the entity IfcAxis2Placement2D

```
1   ENTITY IfcAxis2Placement2D;
2       SUBTYPE OF (IfcPlacement);
3           RefDirection : OPTIONAL IfcDirection;
4       DERIVE
5           P : LIST [2:2] OF IfcDirection := IfcBuild2Axes(RefDirection);
6       WHERE
7           RefDirIs2D : (NOT (EXISTS (RefDirection))) OR (RefDirection.Dim = 2);
8           LocationIs2D : SELF\IfcPlacement.Location.Dim = 2;
9   END_ENTITY;
```

It should be noted that there is always a clear separation between a schema and the instance level in all of the previously mentioned data models. In the EXPRESS world, there is an EXPRESS schema and clear text encoding (or alternative an XML encoding) (Schenck & Wilson 1994) while in the XML world there is an XML schema and an XML instance file. There is a similar separation in the UML world between class and object diagrams or in object oriented programming languages where we have classes and objects.

For the purposes of data exchange, the schema is always known a-priori to both exchange partners and only the instance file is transmitted. Figure 1 shows this situation. The sender and receiver are each aware of the schema (e.g. schema contains classes *IfcAlignment* and *IfcTransition*) and only entity types, attribute values and entity relations are shared between both communication partners. The instance data always follows the static nature of the provided schema.

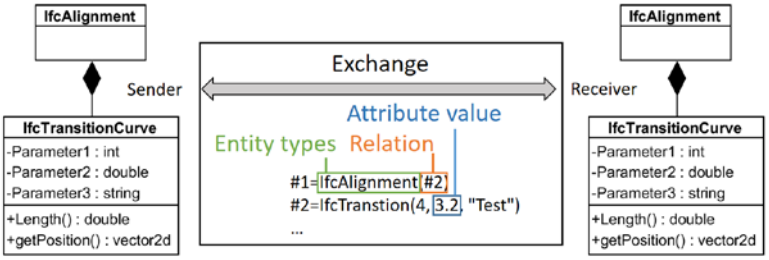


Figure 1: Data exchange in a building information model. Only entity types, attributes and relations are shared.

However, most state of the art BIM data models do not consider the exchange of methods (algorithms) or functions. This functionality is provided solely at the schema level (i.e. the *IfcBuild2Axes* function), but not at the instance level. The exchange of dynamic algorithms at the instance level can yield several advantages that have been discussed and proposed in (Amann, J. Flurl, et al. 2014) and (Amann & Borrmann 2016), and are briefly summarized in the next section.

3 The IFC Programming Language

In (Amann & Borrmann 2016) a high-level programming language IFCPL (IFC Programming Language) is introduced that can be used to formulate and share small programs at the instance level in IFC-based data models. Figure 2 shows a slightly modified version of the IFCPL environment. In the first step an interface is defined. For the purpose of interface definition, IFCPL itself can be used. In Figure 2 the interface named *Interface* is defined. It provides one method (*methodName1*). In the second step we define our interface in the EXPRESS schema as an entity with the same name as the interface. This entity gets one attribute named *Link* which is of type *InterfaceIfcPLRealizationLink*. The *Link* entity holds all the important information about an IFCPL class. An IFCPL class has a name and optional method names. When the host application (sender) wants to export the current data as an IFC file, a STEP file (clear text encoding) is exported with *InterfaceIfcPLRealizationLink* entities. In addition, an IFCPL class in source code form is written to a separate file. The grammar of IFCPL is shown in Figure 3. A snippet of an example program is shown in Listing 2.

Listing 2 IFCPL example program

```
1    import Math;
2    import IArbitraryTransitionCurve;
3    import Debug;
4    // …
5    class Clothoid : IArbitraryTransitionCurve {
6       public Clothoid(IfcPropertySet properties) { // injection
7          startPosition_ = properties.getVector2d("startPosition");  // special syntax for properties
8          startDirection_ = properties.getDouble("startDirection");
9          // …
10         startL_ = clothoidConstant_ * clothoidConstant_ * startCurvature_;
11
12         if (entry_)
13            endL_ = startL_ + length_;
14         else
15            endL_ = max(startL_ - length_, 0.0);
16      }
17
18      public double getLength() const override {
19         return abs(endL_ - startL_);
20      }
21   …
```

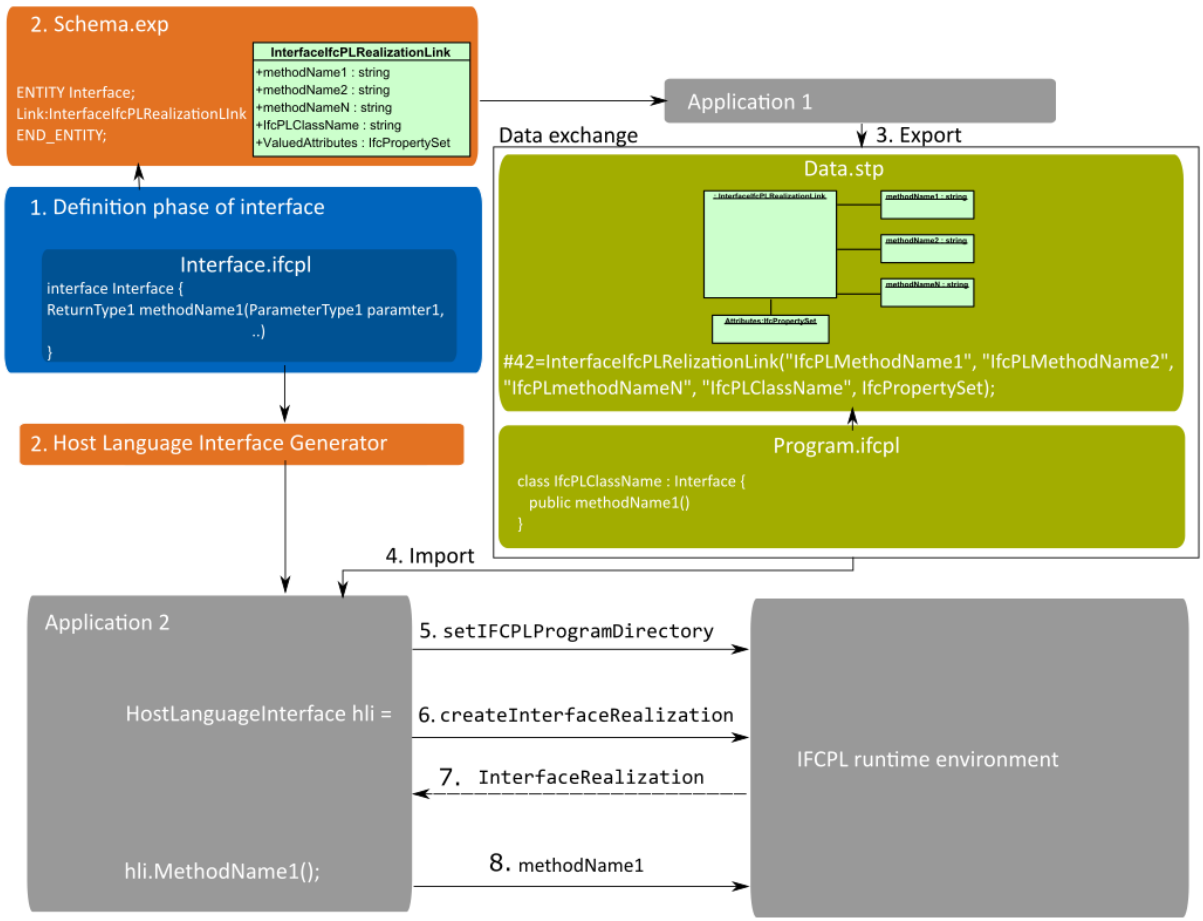


Figure 2: IFCPL overview

The Link object (of type *InterfaceIfcPLRealizationLink*) provides a property set for custom application- or user-defined attributes. In the example shown, the start position and direction of a clothoid transition curve are described by this property set. When the data is handed over to another application the application can read the data and must then interpret the IFCPL program to evaluate the provided methods. For instance, a method *getLength* or *getPosition* can be provided. Assuming that there is an IFCPL execution environment that can easily be used by both applications this approach has several advantages: Implementers can work on a higher abstraction level. For instance, they do not have to agree which transition curve types should be supported – if one of them can provide a program for it, then both application can compute it. Using the same program to interpret the data also ensures that both implementers interpret the data in the same way. Furthermore, we do not have to wait until different software vendors have integrated a certain transition curve, because it would now be possible to also share the implementation/interpretation of this data. The underlying EXPRESS/IFC data model does not have to be changed, no common international agreement on attribute data needs to be reached, and we do not have to come up with a new standard that needs to be accepted and implemented. This approach also permits different countries to have different attributes according to local legal regulations. Exchanging methods in conjunction with subtype polymorphism and generic property sets yields all the aforementioned advantages. The drawback of this approach is the need to have a defined syntax for describing methods and the need for an execution environment (interpreter) for IFCPL programs.

```
⟨program⟩ ::= ⟨stmts⟩

⟨stmts⟩ ::= ⟨stmt⟩
  | ⟨stmts⟩ ⟨stmt⟩

⟨import_stmt⟩ ::= 'import' ⟨ident⟩
  | 'import' ⟨ident⟩ '.' ⟨ident⟩ ';'

⟨using_stmt⟩ ::= 'using' ident ';'
  | 'using' ⟨ident⟩ '.' ⟨ident⟩ ';'

⟨stmt⟩ ::= var_decl ';'
  | ⟨class_decl⟩
  | ⟨if_stmt⟩
  | ⟨meth_decl⟩ ';'
  | ⟨extern_decl⟩ ';'
  | ⟨expr⟩ ';'
  | 'return' ⟨expr⟩ ';'
  | ⟨import_stmt⟩
  | ⟨using_stmt⟩
  | ⟨postfix_stmt⟩ ';'
  | ⟨for_stmt⟩
  | ⟨block⟩

⟨ternary_conditional_exp⟩ ::= ⟨expr⟩ '?' ⟨expr⟩ ':' ⟨expr⟩

⟨call_args⟩ ::= empty
  | ⟨expr⟩
  | ⟨call_args⟩ ',' ⟨expr⟩

⟨var_decl⟩ ::= ⟨ident⟩ ⟨ident⟩
  | 'const' ⟨ident⟩ ⟨ident⟩
  | ⟨ident⟩ ⟨ident⟩ '=' ⟨expr⟩
  | 'const' ⟨ident⟩ ⟨ident⟩ '=' ⟨expr⟩

⟨extern_decl⟩ ::= 'extern' ⟨ident⟩ ⟨ident⟩ '(' ⟨meth_decl_args⟩ ')'

⟨meth_decl⟩ ::= ⟨access_modifier⟩ ⟨ident ident⟩ '(' ⟨meth_decl_args⟩ ')' ⟨block⟩
  | ⟨access_modifier⟩ 'static' ⟨ident⟩ ⟨ident⟩ '(' ⟨meth_decl_args⟩ ')' ⟨block⟩
  | ⟨access_modifier⟩ ⟨ident⟩ ⟨ident⟩ '(' ⟨meth_decl_args⟩ ')' 'override' ⟨block⟩
  | ⟨access_modifier⟩ ⟨ident⟩ ⟨ident⟩ '(' ⟨meth_decl_args⟩ ')' 'const' 'override'
    ⟨block⟩
  | ⟨access_modifier⟩ ⟨ident⟩ ⟨ident⟩ '(' ⟨meth_decl_args⟩ ')' 'const' ⟨block⟩

⟨meth_decl_args⟩ ::= empty
  | var_decl
  | meth_decl_args ',' var_decl

⟨numeric⟩ ::= ⟨INTEGER_VALUE⟩
  | ⟨DOUBLE_VALUE⟩

⟨access_modifier⟩ ::= ⟨ACCESS_MODIFIER⟩

⟨unary_expr⟩ :: '-' ⟨numeric⟩
  | '!' ⟨expr⟩

⟨expr⟩ ::= ⟨ident⟩ '=' expr
  | ⟨ident⟩ '.' ident '(' ')' MULEQ expr
  | ⟨ident⟩ '.' ident '(' ')' '=' expr
  | ⟨ident⟩ ADDEQ expr
  | ⟨ident⟩ MULEQ expr
  | ⟨ident⟩ '(' call_args ')'
  | ⟨ident⟩
  | ⟨numeric⟩
  | ⟨stringx⟩
  | ⟨expr⟩ '
  | ⟨expr⟩ '*' ⟨expr⟩
  | ⟨expr⟩ '/' ⟨expr⟩
  | ⟨expr⟩ '+' ⟨expr⟩
  | ⟨expr⟩ '-' ⟨expr⟩
  | ⟨expr⟩ EQ ⟨expr⟩
  | ⟨expr⟩ '(' ¡expr⟩
  | '(' ⟨expr⟩ ')'
  | ⟨ternary_conditional_exp⟩
  | ⟨unary_expr⟩
  | ⟨ident⟩ '.' ⟨ident⟩ '(' ')'
  | ⟨ident⟩ '.' ⟨ident⟩ '(' ⟨call_args⟩ ')'

⟨postfix_stmt⟩ ::= ⟨ident⟩ '++'

⟨class_decl⟩ ::= 'class' ⟨ident⟩ '{' '}'
  | 'class' ⟨ident⟩ '{' ⟨class_body⟩ '}'
  | 'class' ⟨ident⟩ ':' ⟨ident⟩ '{' ⟨class_body⟩ '}'

⟨class_body⟩ ::= ⟨meth_decl⟩
  | ⟨class_body⟩ ⟨meth_decl⟩
  | ⟨access_modifier⟩ ⟨ident⟩ '(' ⟨meth_decl_args⟩ ')' ⟨block⟩
  | ⟨class_body⟩ ⟨access_modifier⟩ ⟨var_decl⟩ ';'

⟨for_stmt⟩ ::= 'for' '(' ⟨var_decl⟩ ';' ⟨expr⟩ ';' ⟨postfix_stmt⟩ ')' '{' ⟨stmts⟩ '}'
  | 'for' '(' ⟨var_decl⟩ ';' ⟨expr⟩ ';' ⟨postfix_stmt⟩ ')' '{' '}'

⟨if_stmt⟩ ::= 'if' '(' ⟨expr⟩ ')' ⟨stmt⟩ 'else' stmt
  | 'if' '(' ⟨expr⟩ ')' ⟨stmt⟩

⟨block⟩ ::= '{' ⟨stmts⟩ '}'
  | '{' '}'
```

Figure 3: IFCPL grammar

4 XML as a vehicle for dynamic methods

In (Amann & Borrmann 2016) we describe a prototype of an IFCPL implementation. But for real word adoption in the IFC context, there are nevertheless several hurdles. First of all, a standardized version of a programming language such as IFCPL needs to be established, and then an execution environment including an interpreter for this language has to be developed and distributed. Users also have to learn this new IFC Programming Language syntax and semantic. As an alternative, we investigate the use of the XQuery language (W3C 2014). XQuery is a well-defined, standardized language for which detailed documentation is available and a number of execution environments already exist for it. The main question is how it can be used as a vehicle for dynamic methods in an IFC-based data model.

To demonstrate how XQuery can be used for exchanging methods, a simple example is shown. Figure 4 shows an UML class diagram of an interface for transition curves. The interface is supposed to be implemented for a clothoid.

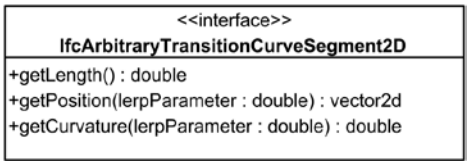


Figure 4: UML class diagram of an interface for transition curves

A clothoid is computed by computing its local x, y coordinates at a given length with Fresnel integrals shown for x in equation (1). The cosine can be computed using its series expansion.

$$x = \int_0^L \cos\frac{L^2}{2\cdot A^2}\,dL = L - \frac{L^5}{40\cdot A^4} + \frac{L^9}{3456\cdot A^8} - \frac{L^{13}}{599040\cdot A^{12}} + \frac{L^{17}}{175472640\cdot A^{16}} - + \cdots \qquad (1)$$

The implementation of the computation of the Fresnel integrals is straightforward and shown in Listing 3. XQuery statements must follow the FLWR schema. A FLWR (pronounced "flower") expression is constructed from FOR, LET, WHERE, and RETURN clauses, which must appear in that specific order. XQuery (3.0) is also Turing complete which means, somewhat simplified, that everything that can be expressed in another programming language can also be calculated by XQuery. The main purpose of XQuery is to query XML documents and, besides functional programming elements, it also has some declarative motivated elements. As a consequence, it is not always trivial to convert an iterative/procedural program to a valid XQuery.

Listing 3 Computation of the x-value of a clothoid. The successive terms of the series are computed by the flower expression. These values are then summed up by the fn:sum function. Summing up the value directly in the for loop does not work, since this does not fit in the FLWR or functional programming concept of XQuery.

```
declare function local:computeX(
   $L as xs:double, $A as xs:double, $iterations as xs:integer
) as xs:double {
  fn:sum(
    for $i in (1 to $iterations)
    let $sign := $i mod -1
    let $L_exponent := 5+ ($i - 1) * 4
    let $A_exponent := $i * 4
    let $factor := local:factorial(2 * $i) * local:pow(2, 2 * $i) * (5+($i - 1)*4) + 3
    let $tmp := local:pow($A, $A_exponent)
    let $term := $sign * local:pow($L, $L_exponent) div ($factor * $tmp)
    return $term
  ) + $L
};
```

The clothoid program described here depends on the following attributes: Start position, start direction, start curvature, orientation, the clothoid constant, its entry value and the corresponding length. The same parameters are also used for the description of clothoids in IFC Alignment 1.0 (Liebich et al. 2015). For further documentation on those parameters, the reader should refer to the IFC Alignment 1.0 documentation. The clothoid values are stored in the corresponding ifcXML document, as shown in Figure 5.

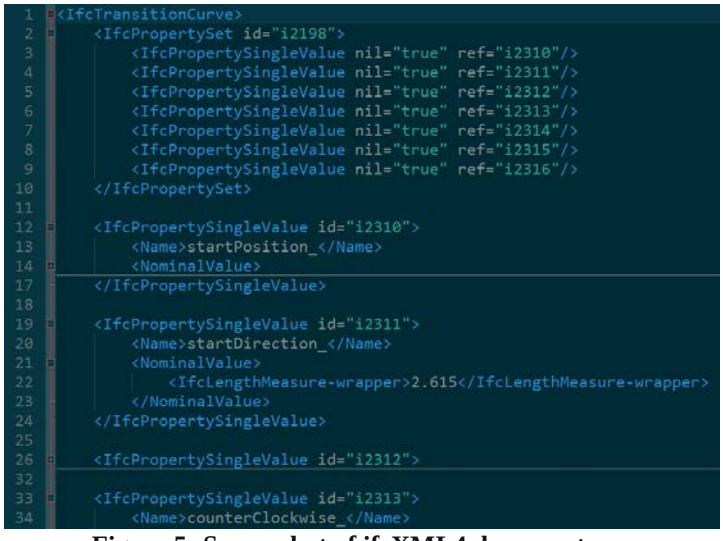


Figure 5: Screenshot of ifcXML4 document

To access these values with XQuery, an XPath can be used as shown in Listing 4.

Listing 4 XPath usage in XQuery to get the start x-position of a clothoid

```
1  declare variable $startPositionX_ :=
2  IfcTransitionCurve/IfcPropertySingleValue[Name="startPosition_"]/NominalValue/IfcLengthMea
3  sure-wrapper/data();
```

In this way we can define an XQuery program to compute the corresponding position value for a given clothoid and a certain station value. Furthermore, we can define programs to compute the length and curvature of the clothoid. This approach allows us to define our own attributes for use within a property set. This can be beneficial when different countries need support for different representations. For instance, some use cases or regulations may require the point of vertical intersection for vertical alignments. Furthermore, we can introduce other transition curve types that were not defined in the standardization process later. For instance, LandXML lacks a C-Clothoid curve type, OKSTRA lacks a sinusoid curve type, and RoadXML and IFC Alignment 1.0 lack the support of a Bloss transition curve type. Even if we were to accommodate a large amount of different transition curve types, it is still likely that we cannot cover them all. A further advantage of this proposal is that the interpretation is now also defined. Traditionally, every implementer writes their own code to interpret plain data and also introduces their own bugs.

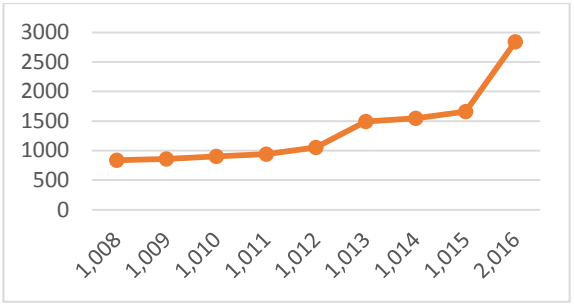
To make this XML-based approach work, the IFCPL environment would need to be modified as follows (see Figure 2):

- Programs should be expressed in the XQuery language instead of IFCPL
- The IFCPL runtime environment is replaced by an XQuery interpreter (a XQuery interpreter is offered for instance by the .Net Framework or Java Platform Standard Edition)

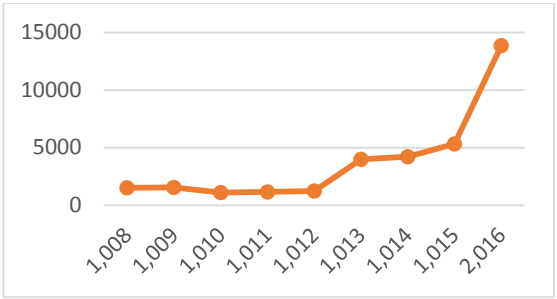
5 Measuring the complexity of Building Information Models: An instance level metric

(Amor et al. 2007; Amor 2015) analyzed the evolution of the IFC Schema using different metrics from object oriented software design. For instance, the total number of entities increased from 180 in IFC 1.5.0 to 750 in IFC 4.0. Also, the total number of attributes increased from 150 to 1400 and the number of types rose from 100 to 400 as well. There was also a growth in the average inheritance depth, in the number of introduced property sets and the number of functions, rules and where-clauses in the IFC EXPRESS schema.

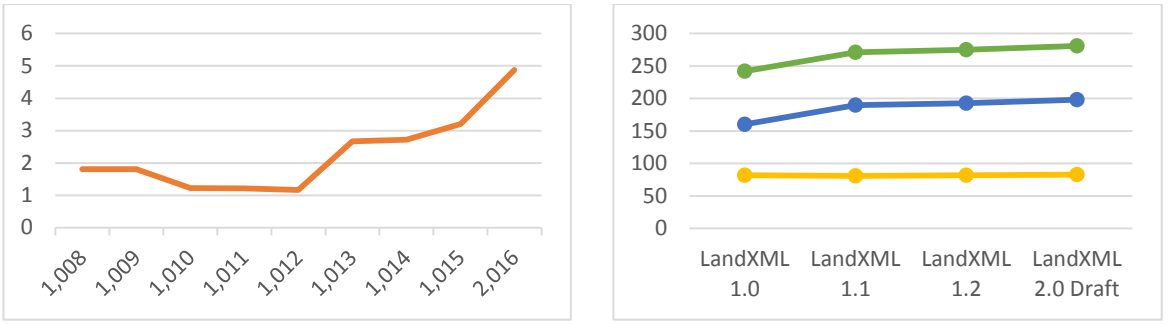
If we apply similar software metrics to successive versions of the OKSTRA or LandXML standards, we can also observe an increase in the number of complex types (corresponds to entities), attributes and average attributes (see Figure 6).



Number of complex types in the OKSTRA standard



Number of attributes in the OKSTRA standard



Average number of attributes in the OKSTRA standard

Evolution of LandXML (types, classes, enumerations)

Figure 6: OKSTRA and LandXML as an example of growing BIM standards

In software engineering, we can use metrics such as depth of inheritance tree, number of children of a class or coupling between object classes to obtain an idea of how complex a data model is (McConnell 2004). But while those metrics provide an indication of how a schema has evolved, it is not well-suited to comparing, for instance, OKSTRA with LandXML or IFC. Comparing OKSTRA 2.016 to LandXML 1.2 and IFC Alignment 1.0 (Liebich et al. 2015) by just counting the number of entities in each standard, reveals that OKSTRA 2.016 is four times more complex than IFC Alignment 1.0 and ten times more complex than LandXML 1.2. But this comparison is not really appropriate, since OKSTRA covers many more use cases than LandXML or IFC Alignment 1.0. For a better metric, we propose using an instance level metric. Instead of counting classes at the schema level we count classes used at the instance level. To apply this metric, at least one instance file is required that serves as a test case/use case scenario. The test case shown in Figure 7 has been created for this study.

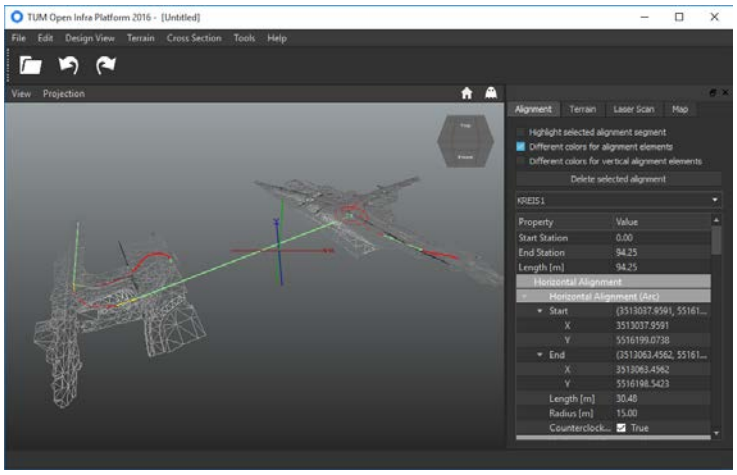


Figure 7: Test case. A simple alignment with an digital elevation model

The test case has been stored in the file formats IFC Alignment 1.0, OKSTRA 2.016 and LandXML 1.2. All three files contain the same data: a digital elevation model and an alignment that consists of a horizontal and vertical alignment. The horizontal alignment is composed of arcs, clothoids and straight lines. The vertical alignment is composed of parabolas and straight line segments. From each file, we can generate every other file without any data loss. For converting from one file format to another we used the TUM Open Infra Platform (Amann et al. 2016). For our metric, we count all unique occurrences of tags in the XML-based instance documents. For the EXPRESS file we counted all unique occurrences of entity names. Table 1 shows the computed results. Listing 4 shows the algorithm used to count the individual occurrences of element tags in XML documents.

Table 1 Number of unique occurrences of tags and entity names

Building Information Model	Test case 1
LandXML 1.2	23 (unique element tags)
IFC Alignment 1.1	32 (unique entity names)
OKSTRA	42 (unique element tags)

Listing 4 Algorithm in C# pseudo syntax for counting unique element tags in XML instance file

```
1    private static int countUniqueXMLElements(string filename) {
2       List<string> withDupes = new List<string>();
3
4       XmlReaderSettings settings = new XmlReaderSettings();
5       settings.IgnoreWhitespace = true;
6
7       using (XmlReader reader = XmlReader.Create(filename, settings)) {
8          while (reader.Read()) {
9             switch (reader.NodeType) {
10               case XmlNodeType.EndElement:
11                  withDupes.Add(reader.Name);
12                  break;
13
14               default: break;
15            }
16         }
17      }
18
19      List<string> noDupes = withDupes.Distinct().ToList();
20      return noDupes.Count;
21   }
```

In this specific use case, OKSTRA does not exhibit ten times as many entities as LandXML, for example, providing us with a better basis for comparing different building information models. The conversion of data between the different data models is also discussed in (Amann, M. Flurl, et al. 2014).

With the instance level metric described here, the complexity of a BIM model is defined by the number of entities used in an instance file. In the transition curve example, we used a generic interface that can be implemented for different curves. This means we do not increase the count of the instance level metric when introducing new transition curve types. The demonstrated approach can support different curve types without increasing the complexity of the BIM model, as rated by the instance level metric. The addition of new transition curve types does not affect model complexity since no new entities are introduced in the schema and instance file. In a traditional model without dynamic methods we would have to modify and extend the schema to support other transition curve types and by doing this we would increase the instance level metric complexity.

6 Results and Conclusion

In this paper we have demonstrated an approach for exchanging method definitions in building information models. XML and XQuery in particular are very widespread and supported by many programming libraries for different programming languages. In contrast EXPRESS or IFCPL have very little support among different programming languages. More documentation is also available for XML. Nevertheless, XQuery has also some minor drawbacks: It has cumbersome access patterns and some users do not find the functional programming paradigm as nice as a procedural or iterative programming styles. Furthermore, it does not fit so well in the traditional STEP/CTE world, but in the long run the IFC world may move completely to XML. So far ifcXML does not have any counterparts of EXPRESS

functions, rules or where-clauses in the XML world. As such, XQuery would also be a good choice here.

At the same time, we should be aware that not every implementer will be happy about sharing implementation details in the form of XQuery programs. But some parts of the standard do not present a mystery and it is more important that every implementer computes the same values – for instance for positions of offset alignments. The adaption of a BIM standard can also be accelerated by using this approach as an implementer does not need to waste time implementing 15 different transitions curve types. The standards committee can provide a standard implementation as free software to accelerate adoption.

Interoperability of an application and a model is defined as the capability of the application to process the respective model. For instance, IFC4 provides the abstract entity type *IfcCurve*. To be fully compatible with this curve concept, an application needs to implement all variants of *IfcCurve*, such as *IfcLine*, *IfcCircle*, *IfcEllipse*, *IfcOffsetCurve2D/3D*, *IfcBSplineCurve* and others. Many software applications and frameworks support only a subset of the different entities provided by IFC4, resulting in a reduced interoperability of models and applications. Considering the ongoing evolution of the IFC schema, more and more object types are introduced and the situation gets even worse. Using our approach many task could be improved, e.g. the visualization of the different *IfcCurve* subtypes could be improved. By introducing a curve interface with a flexible property set and a possibility for the exchange of methods instead of plain data, the burden of implementing all curve types could be shifted from the application level to the instance level of the model. This way, every application could support, for instance, the visualization of every *IfcCurve* subtype. The results in an improved interoperability between the model and the application. Of course, it comes at the cost of setting up an IFCPL runtime environment. However, if we also integrate use cases for surface and solids (a whole geometry kernel could be created) the effort would eventually pay off. Besides the demonstrated approach could also be used to describe parametric properties. For instance, it allows to have smart BIM objects, e.g. a bridge abutment that adapts to the underlying surface model. Furthermore, rules and regulations could be implemented.

References

Amann, J., Flurl, M., et al., 2014. An alignment meta-model for the comparison of alignment product models. *eWork and eBusiness in Architecture, Engineering and Construction: ECPPM 2014*, p.7.

Amann, J., Flurl, J., et al., 2014. An Approach to Describe Arbitrary Transition Curves in an IFC Based Alignment Product Data Model.

Amann, J. et al., 2016. TUM Open Infra Platform 2016. Available at: https://www.cms.bgu.tum.de/oip [Accessed February 8, 2016].

Amann, J. & Borrmann, A., 2016. Embedding Procedural Knowledge into Building Information Models: The IFC Procedural Language and Its Application for Flexible Transition Curve Representation. *Journal of Computing in Civil Engineering*.

Amor, R., 2015. Analysis of the Evolving IFC Schema. In *Proceedings of the CIB W78 conference 2015*. Eindhoven, Netherlands.

Amor, R., Jiang, Y. & Chen, X., 2007. BIM in 2007–are we there yet. *… of CIB W78 conference on Bringing …*. Available at: http://www.cs.auckland.ac.nz/~trebor/papers/AMOR07B.pdf.

buildingSMART, 2016. Industry Fondation Classes. Available at: http://buildingsmart.org/.

Bundesanstalt für Straßenwesen, 2016. Objekt katalog für das Straßen- und Verkehrswesen. Available at: http://www.okstra.de/.

Chamberlin, J., 2003. XQuery: A Query Language for XML.

LandXML.org, 2016. LandXML. Available at: http://landxml.org/.

Liebich, T. et al., 2015. IFC Alignment. , (1). Available at: http://www.buildingsmart-tech.org/downloads/ifc/ifc5-extension-projects/ifc-alignment/ifcalignment-projectpresentation-cs-1.

McConnell, S., 2004. *Code Complete, Second Edition*, Redmond, WA, USA: Microsoft Press.

Schenck, D.A. & Wilson, P.R., 1994. *Information Modeling: The EXPRESS Way*, New York, NY, USA: Oxford University Press, Inc.

W3C, 2014. XQuery 3.0: An XML Query Language. Available at: https://www.w3.org/TR/xquery-30/.