Embedding Procedural Knowledge into Building Information Models: The IFC Procedural Language and Its Application for Flexible Transition Curve Representation

Julian Amann[1] and André Borrmann[2]

Abstract: Building information modeling (BIM) refers to the continuous use of semantically rich three-dimensional (3D) building models throughout the entire lifecycle of a facility. BIM data models capture the geometry as well as the semantics of buildings and its constituent parts in an object-oriented manner. They have been developed to achieve high-quality data exchange between software applications, reduce data loss, and increase interoperability. Across the architecture, engineering, and construction (AEC) industry, the open-data model industry foundation classes (IFC) has become a well-accepted standard. The main contribution of this paper is the introduction of a procedural language called the IFC procedural language (IFCPL) that can be easily embedded into an IFC-based building information model. This enables software developers to exchange procedural programs between different software applications in a platform-independent way using a neutral data format. IFCPL programs describe algorithms that operate on a set of input parameters and generate a set of output parameters (return values). The EXPRESS language, which is part of standard for the exchange of product model data (STEP), provides the concept of functions and rules for representing algorithmic knowledge. However, EXPRESS operates on the schema level, i.e., the rules and algorithms defined apply to all instances of the respective entity type in the same manner. IFCPL shifts this concept from the schema (class level) to the instance level and is not limited to realizing data integrity or attribute derivation. The paper describes in detail the features and the design of the IFCPL language. To illustrate its applicability, the language is used to demonstrate how transition curves of road or railway alignments can be described in a very flexible manner: IFCPL allows the definition and exchange of algorithms for computing the curve coordinates from general curve parameters. In doing so, software developers can dynamically define and exchange new transition curve types without modifying the IFC data model. At the same time, this approach helps avoid misinterpretations of informal curve descriptions. The procedural language provides a powerful option for adding enhancements and reduces software development costs by allowing semiautomated integration. Besides the alignment use case, there are many other application areas in which IFCPL can be used and where software developers as well as software users can benefit from it. They are discussed extensively throughout the paper. **DOI: 10.1061/(ASCE)CP.1943-5487.0000592.** *This work is made available under the terms of the Creative Commons Attribution 4.0 International license, http://creativecommons.org/licenses/by/4.0/.*

Introduction

Building information modeling (BIM) refers to the continuous use of semantically rich three-dimensional (3D) building models throughout the entire lifecycle of a facility (Eastman et al. 2011). BIM data models capture the geometry as well as the semantics of buildings and its constituent parts in an object-oriented manner. They have been developed to achieve high-quality data exchange between software applications, reduce data loss, and increase interoperability (Eastman et al. 2005). Among the most widespread standards for the exchange of product manufacturing information in use today is standard for the exchange of product model data (STEP) (Xu and Nee 2009; International Organization for Standardization 1995). For the architecture, engineering, and construction (AEC) domain, STEP has been adapted to create the BIM data model industry foundation classes (IFC). IFC is developed and maintained by the buildingSMART organization (Eastman 1999). It has been adopted as an ISO standard (ISO 16739), forms part of AEC data-exchange regulations in many countries (Waterhouse et al. 2014; Weise et al. 2009) and has been implemented by a large number of software vendors. The IFC standard is also recommended by the U.S. National Institute of Building Science in the US BIM standard NBIMS-US VERSION 3 (buildingSMART Alliance 2015). In its current version, the IFC model focuses on the representation of buildings, however several extension projects are underway to include infrastructure facilities in future versions of the data model. The Infrastructure Alignment and Spatial Reference System project (also denoted as P6) focuses on the development of an alignment and reference system to provide data structures for representing alignments of roads and other linear infrastructure facilities. The data model developed is called IFC Alignment. In this paper, the first version of the IFC Alignment standard is considered as a point of departure and extended by a flexible alignment representation.

The IFC data model is a very large and complex data model. Some researchers have called the data model overly complex (Amor et al. 2007). But the data model is by design very fine-grained, i.e., it has an extensive class inheritance hierarchy and the defined classes have comprehensive attribute lists. As a consequence, both the standardization process as well as the subsequent

[1]Research Assistant and Ph.D. Candidate, Chair of Computational Modeling and Simulation, Leonhard Obermeyer Center, Technische Univ. München, Arcisstraße 21, 80290 Munich, Germany (corresponding author). E-mail: julian.amann@tum.de

[2]Professor, Chair of Computational Modeling and Simulation, Leonhard Obermeyer Center, Technische Univ. München, Arcisstraße 21, 80290 Munich, Germany. E-mail: andre.borrmann@tum.de

Note. This manuscript was submitted on December 17, 2014; approved on February 10, 2016; published online on June 3, 2016. Discussion period open until November 3, 2016; separate discussions must be submitted for individual papers. This paper is part of the ***Journal of Computing in Civil Engineering***, © ASCE, ISSN 0887-3801.

implementation (adoption) by software developers has been slow and laborious. To make it easy to extend the data model, the IFC includes a flexible extension mechanism that allows the ad hoc definition of additional attributes (IfcProperties) without modifying the underlying schema. While these extensions do not form part of the data model itself, they are subject to the standardization process, since some Property Sets are predefined by the IFC4 Standard (Pset_ActorCommon, Pset_WallCommon, etc.), and some Model View Definitions [e.g., COBie (East 2007)] define certain property sets as mandatory. In addition, it is possible for software implementers to agree on a custom defined property set in order to target a particular exchange scenario.

However, the most significant challenge for IFC implementers (and at the same time the greatest source of implementation errors) lies in the correct interpretation of the information encoded in the data model. Throughout this paper, the authors discuss this issue by referring to the representation and interpretation of transition curves as part of the upcoming IFC Alignment standard. There are numerous transition curves in practical use in the entire world, and integrating all of them explicitly in the data model is not a feasible option. The long history of IFC development has shown that international standardization should restrict itself to items that are relevant on an international level. Accordingly, an international standardization effort (which is typically limited with respect to time and resources) will only define the most common transition curves which are in international use. In IFC-Alignment 1.0, for example, only clothoids have been included. However, for IFC-Alignment to be of any practical use, there must be ways to represent other transition curves that are in use on a regional or national level. The IFC procedural language (IFCPL) approach presented here proposes a mechanism that allows regional standardization groups to define additional transition curves, and—most importantly—includes the interpretation semantics with the files being exchanged. Consequently, the proposed approach enables an international software vendor to interpret the transmitted regional transition curve correctly without the need to develop any particular code.

To this end, the authors propose extending the IFC data model with capabilities that allow the inclusion of interpretation algorithms in the data exchange. For this purpose, the IFC Procedural Language (IFCPL) has been developed: a simple imperative programming language that is introduced in this paper. Programs written in this language are included in IFC instance files and are thus exchanged between different software applications. On the one hand, this approach provides a high degree of flexibility, as parameters and interpretations can be defined and exchanged in an ad hoc manner, and on the other, it significantly decreases the effort required to implement this particular part of the data model, as the processing algorithm is delivered with the data. Additionally, it helps to reduce potential errors that arise through the misinterpretations of data models and provides an improved flexible extension mechanism that works for all participants of the data exchange process. The only prerequisite is the availability of an IFCPL interpreter on the receiving side.

Taking the higher level perspective of knowledge representation (Rasdorf 1985), it can be stated that the IFC building information model is so far restricted to capturing and exchanging static knowledge, with the exception of IfcContraints. IfcConstraints make it possible to formulate logical assumptions about the underlying data but lack typical procedural language features such as control flow statements (e.g., *if* and *for*) or subroutines/functions. However, it is not possible to capture procedural knowledge using the IFC data model, i.e., knowledge that describes the stepwise (algorithmic) performance of a task. This significant gap is filled by the proposed IFC Procedural Language.

Although this paper focuses on the application of IFCPL to describe arbitrary alignment curves, a wide spectrum of applications is imaginable. Whenever an algorithm is required for the correct interpretation of exchanged data, IFCPL can be employed to capture and transmit this algorithm in a vendor-independent manner.

The paper begins with a discussion of the state of the art of building information modeling followed by an introduction to the proposed programming language IFCPL and its main features. An example of IFCPL in application then follows with a detailed breakdown of the flexible definition of transition curves as part of the upcoming infrastructure extension of the IFC model, as a demonstration of the benefits that IFCPL brings. The paper's main findings are summarized and discussed in the conclusion.

State of the Art

Representing Algorithms in EXPRESS-Based Building Information Models

The basis of the product modeling standards STEP and IFC is formed by the data modeling language EXPRESS. Among its many features, EXPRESS makes it possible to associate the definition of a data model entity with integrity rules. To this end, the *WHERE* clause is applied, which makes it possible to specify constraints for the attributes of the entity. A simple example for this is given in Fig. 1.

```
TYPE age = INTEGER;
WHERE NotNegative : SELF >= 0;
END_TYPE;
```

```
ENTITY line;
   start : point;
   end : point;
DERIVE
   length : distance:=SQRT((end.x - start.x))**2
                        + (end.y - start.y)**2);
END_ENTITY;
```

```
RULE rule_name FOR (entity_type_1, ..., entity_type_N);
   (* executable statements *)
WHERE
   (* some expression that returns TRUE or FALSE *)
END_RULE;
```

```
RULE max_number_of_passengers FOR (aircraft);
WHERE
   max_is_853 : SIZEOF(bookshelf) <= 853;
END_RULE;
```

```
FUNCITON functionName(
   parameterName1 : entityType1, ..) : resultType;
LOCAL
   localVariableName : variableType;
   ...
END_LOCAL;
   REPEAT i := 1 to SIZEOF(localVariableName);
      IF(...) THEN
         result := result + 1;
      END_IF;
   END_REPEAT;
RETURN(result);
END_FUNCTION;
```

Fig. 1. Different concepts of the EXPRESS language to store high-quality knowledge

In the example shown, a type is defined (*TYPE weight*) with the constraint that the value of an instance of this type always needs to be positive. In the context of the transition curve application scenario, a *WHERE* clause can be used, for instance, to specify that the start radius of a clothoid always needs to be positive.

Alongside *WHERE* clauses, it is also possible to make use of the *DERIVE* mechanism. An example of a *DERIVE* attribute is also shown in Fig. 1. *DERIVE* attributes are computed from the explicitly defined attributes. Because of this, these attributes are only readable. EXPRESS also offers local and global *RULES*. Rules consist of a number of statements and a *WHERE* clause. Rules can operate on the data of different entities and return *TRUE* or *FALSE* depending on the evaluation of the *WHERE* clause. Again, an example (max_number_of_books) is shown in Fig. 1.

The most powerful capability of EXPRESS for representing procedural knowledge are *FUNCTION*s. A Function can have an arbitrary number of arguments (parameters) and perform arbitrary computations. Functions are used to implement *RULES* and to compute derived attributes. EXPRESS supports basic arithmetic operations and expressions, logical operators and expressions, numerical functions, operators on aggregates (e.g., sizeof), simple queries (e.g., all walls with a width smaller than 10 units), and entity equality test operators.

Further details on the EXPRESS standard can be found in Schenck and Wilson (1994) and in Xu and Nee (2009). The access and interpretation of *DERIVE*able attributes and the evaluation of *RULEs* and *FUNCTIONs* requires an EXPRESS interpreter that can interpret and execute (evaluate) the corresponding constructs. *RULEs* and *FUNCTIONs* were, however, not conceived to support the exchange of algorithmic knowledge but rather to improve data integrity across the model. Many implementations of STEP parsers do not support derived attributes and *WHERE* clauses that contain rules and/or functions. It seems that the great benefit of data integrity tests offered by *WHERE* clauses is ignored by most software developers. Nevertheless, to preserve data integrity, the ability to check the assertions made in the *WHERE* clauses can be of great help. The EXPRESS schema of IFC 4, for example, contains more than 430 *WHERE* clauses to check for data inconsistencies.

These functionalities of EXPRESS operate on the schema level, i.e., the rules and algorithms defined apply to all instances of the respective entity type in the same manner. Schema-level approaches for representing algorithmic knowledge are rather static and meant to persist for a long time. Modifications to these algorithms result in a new schema which (1) is subject to the standardization procedure, and (2) must be adopted by software vendors to provide read/write functionalities for the target applications. Both aspects involve significant effort.

In the IFCPL concept proposed in this paper, the algorithms (programs) are not part of the schema, but are defined on the instance level. This provides a much higher degree of flexibility, as programs can be defined and modified without having to alter the IFC schema. In combination with the available generic data structure IFC property set, this approach results in a powerful subschema extension mechanism for a very flexible definition of data structures and algorithms.

Extensible Markup Language

An alternative to STEP-P21 files for exchanging IFC instances is the use of a corresponding extensible markup language (XML) mapping of the schema, denoted as ifcXML (Nisbet and Liebich 2009; Liebich and Weise 2013). As XML technology is significantly more widespread and much better supported by programming infrastructure, it is expected that it will supersede EXPRESS in the near future. In the currently available EXPRESS-Schema-to-XML-Schema mappings, EXPRESS rules and functions are not included. By contrast, IFCPL programs are represented on an instance level, and integrating them into the XML mapping is therefore a straightforward procedure covered by the available mapping mechanisms.

In addition, XML technology also offers a number of promising features such as the XQuery language. XQuery is a strongly typed and Turing complete language. The main propose of the language is to query data from XML documents. But since the language is targeted at XML document queries, it is not suitable for embedding procedural knowledge into IFC-based data models.

Structured Query Language Stored Procedures

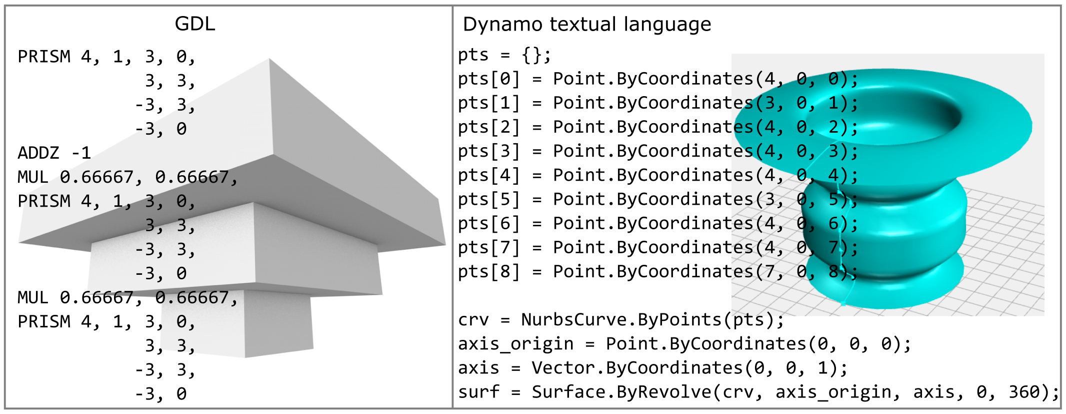
The structured query language (SQL) (ISO/IEC 2011) is a declarative language for defining and manipulating data structures within relational databases. Furthermore, it supports complex query mechanisms for filtering, searching, and restructuring data contained in relational databases. Several BIM software applications such as *ArchiCAD* or *Revit* provide an export feature that can map their internal database, which contains building elements (walls, slabs, roofs, shells, etc.), to an SQL database. Persistent stored modules (SQL/PSM) are a part of the SQL standard. SQL/PSM defines a procedural language for SQL that can be used in so-called stored procedures. This idea dates back to 1996 and was first introduced by Eisenberg (1996). The feature set of stored procedures includes control flow statements, variables, assignments, expressions, and subroutines. Stored procedures are used to describe complex tasks such as parts of the business logic of a company. Many dialects have been derived and implemented on top of SQL/PSM (e.g., Oracle PL/SQL or Microsoft Transact-SQL). SQL/PSM offers the possibility to define, for instance, business rules in a platform-independent way that can be accessed and used by different applications. This makes it possible to reuse code and provides client programs with a higher abstraction level. Besides this, stored procedures are often used for performance (e.g., reducing network traffic) or security reasons. Like the IFCPL approach, PL/SQL manages its stored procedures at the instance level. One of the main problems of SQL is the lack of awareness of the object-oriented principle of inheritance. Because of this, it is difficult to map IFC to SQL (Mazairac and Beetz 2013).

Object Constraint Language

The object constraint language (OCL) is part of unified modeling language (UML) (Fowler 2004). It was introduced into UML to define specific constraints. For example, a software architect wants to ensure that the class attribute *age* in a class *person* should never have a negative value. The OCL makes it possible to formulate this rule as an OCL constraint ($context\ Person\ inv{:}\ self.age >= 0$). Furthermore, OCL provides the ability to define preconditions and postconditions and offers other ways of checking and assuring the accuracy and consistency of data. While it is not possible to write a program with control flow statements in OCL, it is nevertheless an approach that makes use of an embedded language to formulate limited logical checks.

Parametric Design

Parametric design allows the creation of flexible geometric models using parameters for dimensions and makes it possible to define numeric relationships between these parameters by means of mathematical formulas as well as geometric-topological constraints between geometric entities. The result is a flexible geometric model



```
PRISM 4, 1, 3, 0,
            3, 3,
           -3, 3,
           -3, 0
ADDZ -1
MUL 0.66667, 0.66667,
PRISM 4, 1, 3, 0,
            3, 3,
           -3, 3,
           -3, 0
MUL 0.66667, 0.66667,
PRISM 4, 1, 3, 0,
            3, 3,
           -3, 3,
           -3, 0
```

```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(3, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(3, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);
pts[7] = Point.ByCoordinates(4, 0, 7);
pts[8] = Point.ByCoordinates(7, 0, 8);

crv = NurbsCurve.ByPoints(pts);
axis_origin = Point.ByCoordinates(0, 0, 0);
axis = Vector.ByCoordinates(0, 0, 1);
surf = Surface.ByRevolve(crv, axis_origin, axis, 0, 360);
```

Fig. 2. Example of GDL

that can be adjusted by manipulating its primary parameters (Woodbury 2010). In contrast to explicit geometric models with fixed dimensions, a parametric model can capture the design intent and represent domain knowledge. This makes it easier to rework the model when changes are made, and simultaneously provides a high degree of reusability in other similar projects. As a result, efficiency is significantly increased (Ji et al. 2013). In the early architectural-design and form-finding phases, parametric behavior is often realized by defining programs using a visual programming language (Ritter et al. 2013). In contrast to the predefined parametrics of BIM authoring tools, this is a typical example of procedural knowledge defined on an instance level. Today, it is not possible to transfer this kind of knowledge using the available features of the IFC data model.

Vendor-Specific Programming Languages for Capturing Procedural Knowledge

Several software applications in the BIM domain have come up with proprietary programming languages for specific tasks. For example, *ArchiCAD 18* offers a proprietary scripting language called geometric description language (GDL). It can be used to describe 3D parametric objects like doors, windows, stairs, or structural elements (Fig. 2, left). More details on GDL can be found in the program *ArchiCAD 18* itself.

Dynamo, a visual programming add-in for *Revit* and *Vasari* supports the so-called Dynamo textual programming language (formerly known as DesignScript) (Fig. 2, right). This language can also be used to describe geometry in a generative, parameterized manner.

Other software packages reuse existing programming languages to express procedural knowledge. For instance, *Generative Components*, which is based on *MicroStation*, as well as *Digital Project*, which is based on *CATIA*, use the programming language visual basic for applications (VBA) to describe the construction of geometric objects (Hubers 2010). The Grasshopper add-in for *Rhinoceros 3D* uses C# for scripting. These systems can be categorized as parametric design systems (Hubers 2010), generative design systems, or as design computing systems.

Besides such systems, formal languages have been developed for the acquisition of product-related information (Lee et al. 2006a). Although these languages are well-suited for particular use cases, each of them has specific limitations. They are either too focused on one specific use case, not platform-independent, or not powerful enough for representing complex procedural knowledge.

IFC Procedural Language

General Approach

The IFC procedural language (IFCPL) has been developed for embedding procedural knowledge in exchangeable IFC models, and is the main contribution of this paper. It makes it possible to define and exchange algorithms (programs) in a vendor-independent manner. The defined algorithms typically encode procedures for performing computation, processing, and analysis tasks. Possible applications of IFCPL range from the definition of complex functions for quantity take-off to the description of the behavior of parametric objects (Lee et al. 2006b). As the processing algorithms form part of the exchange process, they can be used directly by the receiving application, thus saving significant programming effort and avoiding misinterpretations and errors.

As discussed in the "Related Work" section, an IFCPL program is not included in the IFC schema, but defined on the instance level. Compared with a schema-level approach, this provides much higher flexibility with regard to defining new programs, as schema modifications require a lengthy standardization process. Nevertheless, IFCPL programs must be agreed upon by domain experts and the software vendors involved in the data exchange scenario to ensure the correctness of the defined algorithms. In combination with the generic data type *IfcProperty*, the proposed language provides a very powerful extension mechanism for the IFC schema.

IFCPL is a special-purpose programming language (Mitchell 2002) designed to meet the demands of typical AEC data exchange scenarios. It is a simple, imperative language providing the control structures loop and conditional branch. An IFCPL program consists of a sequence of statements. In addition to control flow statements (loops or conditional branches), subroutines can also be defined. The execution of an IFCPL program starts with the main entry function. The name of the entry function can be specified and is optional. If no name is defined, it is assumed that the entry function is called *main*.

Like many other programming languages, IFCPL provides the possibility to define and use variables. However, it does not demand the explicit declaration of the type of variable. Instead, the type of variable is implicitly defined in the course of the initialization process. IFCPL uses a dynamic, implicit and strong type system: Type checks are performed during runtime (dynamic typing), types of variables are derived by type inference (implicit typing) and there are restrictions on how types can be intermingled (strong typing). The language is deeply integrated with the IFC model by allowing access to its type system.

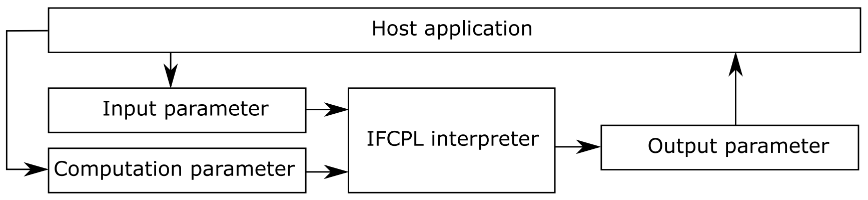


Fig. 3. Overview of the different groups of parameters

An IFCPL program makes use of three different groups of parameters (Fig. 3):

- Input parameters define the input for the algorithm;
- Computation parameters define how the computation is performed; and
- Output parameters define the result of the computation.

Parameter passing from and to the host program is realized by means of IFC Property Sets.

To execute an IFCPL program, a dedicated interpreter is required. As this may represent a significant hurdle to the adoption of the IFCPL language, the authors propose that a reference implementation is provided by the IFC standardization committee, which can be used free-of-charge for any commercial or noncommercial software development project. Since there are well-established cross platform and cross language techniques, it would be sufficient to provide the reference implementation in C/C++. A prototypical interpreter has been developed over the course of the presented research and can be downloaded from Sengupta and Amann (2015).

Extension of the IFC Schema

Although IFCPL programs are defined on an instance level, a minimal extension of the IFC schema is necessary to provide the data structures capable to hold the IFCPL code. To this end, the entity *IfcProgram* is introduced, which contains the relevant information of an IFCPL program. Fig. 4 shows how the *IfcProgram* entity can be integrated into the IFC EXPRESS schema.

An *IfcProgram* instance holds the corresponding source code of the IFCPL program in plain-text form (*SourceCode*), the name of the main entry point (*MainEntryPointName*) and the corresponding feature level (*FeatureLevel*). The feature level determines a well-defined set of functionality that is supported by this *IfcProgram* instance. At the time of writing, there is only one defined feature level denoted *IFCPL_1_0* (IFCPL Version 1), which means that this program supports all features of IFCPL Version 1. By use of these

```
TYPE IfcProgramFeatureLevel = ENUMERATION OF
        (IFCPL_1_0      (* First version of the IFCPL *)
        ,NOTDEFINED);   (* Unknown version *)
END_TYPE;

ENTITY IfcProgram;
 SUBTYPE OF (IfcRoot);
        FeatureLevel            : IfcProgramFeatureLevel;
        MainEntryPointName      : OPTIONAL IfcText;
        SourceCode              : IfcText;
END_ENTITY;

ENTITY IfcProgramInstance;
        ComputationParameter    : IfcPropertySet;
        Program                 : IfcProgram;
END_ENTITY;
```

Fig. 4. EXPRESS definition of *IfcProgram*, *IfcProgramInstance*, and *IfcProgramFeatureLevel*

feature levels, the IFCPL language can be extended in the future without breaking downward compatibility.

It is also conceivable to define a simplified subset of the IFCPL Version 1 and a corresponding feature level (e.g., SIMPLE_IFCPL_1_0), which does not support advanced language features such as subroutines or IFC types and is restricted to simple expressions.

An alternative to representing IFCPL in plain text would be to store the program's abstract syntax tree. To realize this, explicit entities for tokens and each grammar rule (e.g., *IfcIFCPLIfStatement* or *IfcIFCPLWhileStatement*) would have to be defined. This would allow only syntactically correct programs to be represented by IFC instance files. However, this approach would introduce many new items in the express schema and increase the size of instance files. Since syntax errors are also detected by the IFCPL compiler, plain text is used.

The *IfcProgram* entity inherits from the *IfcRoot* object (Fig. 4). Doing this, the *IfcProgram* inherits *IfcOwnerHistory* in which, among other things, the application which created the *IfcProgram* instance, can be defined. The *IfcProgramInstance* object can be used to connect a computation parameter property set with computation parameters. Different program instances can share the program or/and the same computation parameters.

IFCPL Grammar

The grammar of the proposed IFC Procedural Language is defined in Fig. 5 in Extended Backus Naur Form (ISO/IEC 1996). It defines valid token combinations (programs). The start symbol is $< Program >$.

The rationale behind the design of the grammar is straightforward. It supports basic arithmetic operations, control flow mechanisms, and function definitions. A program consists of statements that are concluded by a semicolon. Statements can be expressions that can be numeric values, strings or Boolean values. As depicted in Fig. 5, IFCPL supports conditional expressions ($if(\ldots)\{\}else\{\}$), a condition-controlled loop (*while*), and a count-controlled loop (*for*). Statements are executed one after another. Additionally, the language supports a C-style line comment (*// my comment*). Comments are not part of the grammar because they are ignored by the lexer. Computed values can be returned via the return function [$return(\ldots)$]. The return function supports multiple arguments, for instance the statement $return(1,3,2)$ will push the values 1, 3, 2 onto the return stack.

IFCPL provides several basic built-in functions like abs, sin, cos, tan, print, input, return, factorial, or sqrt. The sqrt function, for instance, computes the square root of a number. The print function can be used to write output for debugging purposes. Variables in IFCPL need not be defined. A variable always takes the type of the expression it is assigned to.

The IFCPL is similar to a C programming language. Since many software developers are accustomed to C-like or C++-like

```
Program     = Statement, ';';
Statements  = Statements, Statement, ";"
            | Statements, Block, ';';
Block       = '{', Statements, '}'
Statement   = While | If | Block | Func | Call | Return | For | PostIncrmnt;
While       = 'while', '(', Expr, ')', Statement;
For         = 'for', '(', Expr, ';', Expr, ';', Expr, ')', Statement;
If          = 'if', '(', Expr, ')', Statement
            | 'if', '(', Expr, ')', Statement, 'else', Statement
            | 'if', '(', Expr, ')', Statement, 'else', 'if', '(',
              Expression, ')', Statement, 'else', Statement;
Func        = 'func', Identifier, '(', ParamList, ')', Block;
ParamList   = Identifier | Numeric | ParamList, ',', { Identifier | Numeric };
Return      = 'func', Identifier, '(', ParamList, ')', Block;
Call        = Identifier, '(', ParamList, ')';
Expr        = AssignExpr, Expr | AssignExpr;
AssignExpr  = Identifier, '=', AssignExpr | OrExpr;
OrExpr      = AndExpr, '||', OrExpr | AndExpr;
AndExpr     = EqualExpr, '&&', AndExpr | EqualExpr;
EqualExpr   = RelExpr, '==', RelExpr
            | RelExpr, '!=', RelExpr
            | RelExpr;
RelExpr     = AddExpr, '<', AddExpr
            | AddExpr, '>', AddExpr
            | AddExpr, '<=', AddExpr
            | AddExpr, '>=', AddExpr
            | AddExpr;
AddExpr     = MulExpr, '+', AddExpr | MulExpr, '-', AddExpr | MulExpr
MulExpr     = UnaryExpr, '*', MulExpr | UnaryExpr, '/', MulExpr | UnaryExpr;
UnaryExpr   = Factor | '-', Factor | '!', Factor;
Factor      = Identifier
            | Call
            | Numeric
            | String
            | 'true'
            | 'false'
            | '(', Expr ')';
PostIncrmnt = Identifier, '++';
Identifier  = Character, { Character | Digit } ;
Character   = "A" | ... |"Z" ;
Digit       = "0" | ... | "9" ;
Numeric     = [-], {Digit} [ '.', {Digit}]
```

Fig. 5. Grammar of the IFCPL in Extended Backus Naur Form

programming languages such as C# or Java, it should not be hard for them to adapt to the concepts of IFCPL.

For interpreting IFCPL programs, a parser is required to convert the tokens into an abstract syntax tree according to the described grammar. In addition, an interpreter is needed to process the abstract syntax tree to execute an IFCPL program.

Use of IFC Types

In order to facilitate seamless data exchange between an IFCPL program and the IFC instance model, all IFC entity types are embedded in the language. This allows the use of all types defined by the IFC data model. In this manner, instances/variables of specific IFC entities like *IfcWall* can be used in IFCPL program code. To create an instance of a certain IFC type, there is a corresponding $create < Type >$ function where $< Type >$ can be replaced with the corresponding IFC type such as *IfcWall.*

For creating lists, bags, or sets, a $createList|Bag|Set < Type >$ function is provided. The different attributes of an IFC entity can be accessed by the dot-operator (.). Elements of collections can be counted via a *count* function. Moreover, elements can be added to collections using an *add* function, and a specific element in a collection can be accessed using a specialized *get* function.

Simple Example

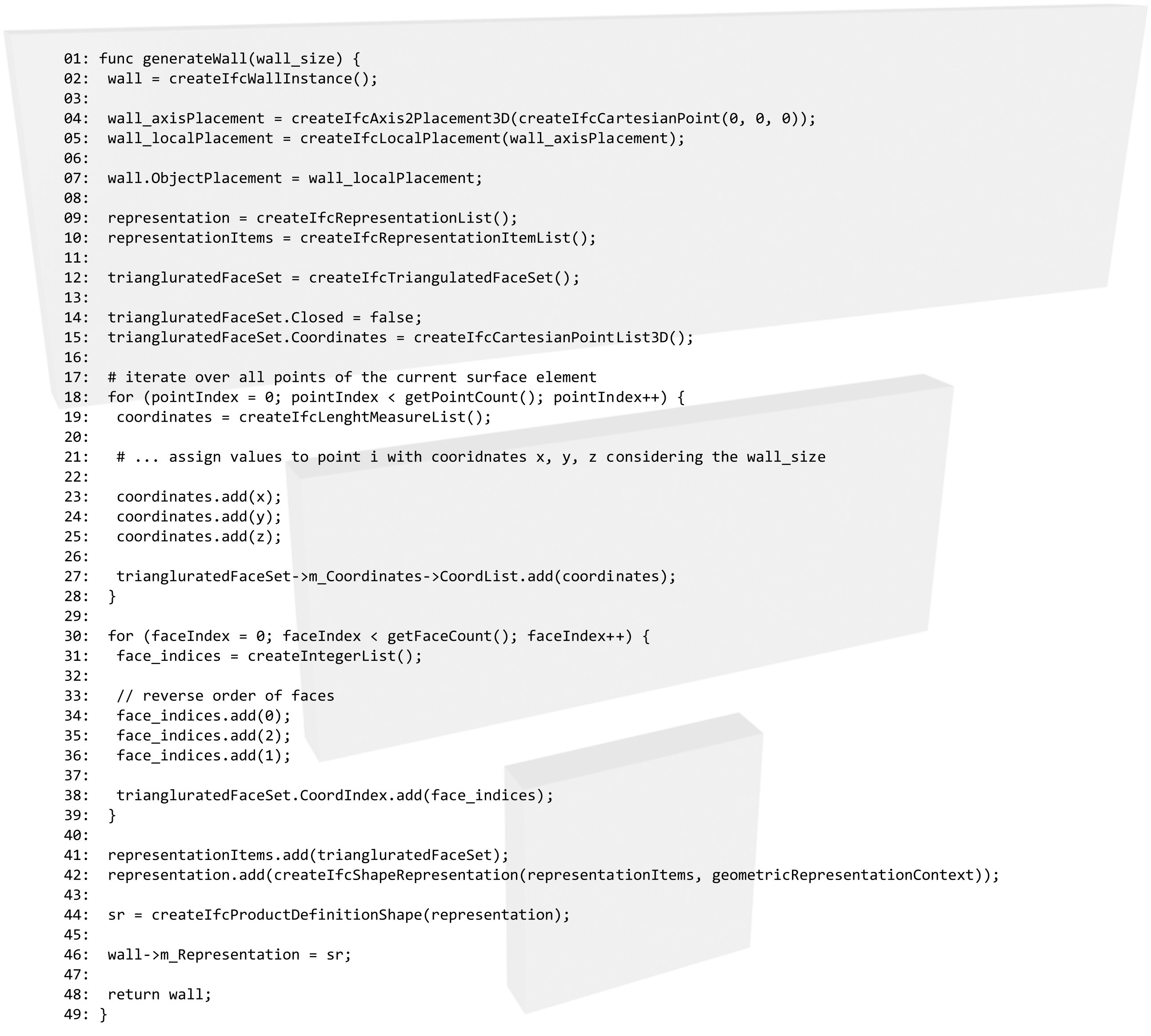
An example of an IFCPL program is provided in Fig. 6. The code shows how a simple wall generator can be implemented using the capabilities of the IFCPL. In Line 2, a variable of type *IfcWall* is created. Afterwards, the wall instance is filled with the necessary properties, including its placement and geometry. For the geometric shape representation, a triangulated face set is created in Line 12 and filled with vertex and index data in Lines 18 to 39. The generated wall is then returned as an *IfcWall* instance and hence added to the output parameter property set.

Handling Input, Computation, and Output Parameters

In most of the application scenarios, an IFCPL program requires the definition of computation parameters as well as input parameters (function arguments).

Computation parameters are provided through a mechanism called injection, which extends a program instance with the variables and initialization values defined in the computation parameter property set. Fig. 7 shows an overview of this process.

The example shows an IFC instance model that contains an *IfcThermalProgram* element. This element references a computation parameter property set as well as an IFCPL program for computing the thermal transmittance of a three-layered material. Furthermore,



```
func generateWall(wall_size) {
 wall = createIfcWallInstance();

 wall_axisPlacement = createIfcAxis2Placement3D(createIfcCartesianPoint(0, 0, 0));
 wall_localPlacement = createIfcLocalPlacement(wall_axisPlacement);

 wall.ObjectPlacement = wall_localPlacement;

 representation = createIfcRepresentationList();
 representationItems = createIfcRepresentationItemList();

 triangluratedFaceSet = createIfcTriangulatedFaceSet();

 triangluratedFaceSet.Closed = false;
 triangluratedFaceSet.Coordinates = createIfcCartesianPointList3D();

 # iterate over all points of the current surface element
 for (pointIndex = 0; pointIndex < getPointCount(); pointIndex++) {
  coordinates = createIfcLenghtMeasureList();

  # ... assign values to point i with cooridnates x, y, z considering the wall_size

  coordinates.add(x);
  coordinates.add(y);
  coordinates.add(z);

  triangluratedFaceSet->m_Coordinates->CoordList.add(coordinates);
 }

 for (faceIndex = 0; faceIndex < getFaceCount(); faceIndex++) {
  face_indices = createIntegerList();

  // reverse order of faces
  face_indices.add(0);
  face_indices.add(2);
  face_indices.add(1);

  triangluratedFaceSet.CoordIndex.add(face_indices);
 }

 representationItems.add(triangluratedFaceSet);
 representation.add(createIfcShapeRepresentation(representationItems, geometricRepresentationContext));

 sr = createIfcProductDefinitionShape(representation);

 wall->m_Representation = sr;

 return wall;
}
```

Fig. 6. An IFCPL example program that generates walls

the program expects three input parameters (*ThicknessLayer1*, etc.). The input parameters are provided to the program instance in a similar manner. Input parameters are defined by the host application and are provided to the interpreter as an *IfcPropertySet*. The interpreter updates the values of the corresponding program variables (*ThicknessLayer1*, etc.) and is then ready to execute the program.

Although the language itself is untyped, each variable is dynamically assigned the same type as defined by the IFC property set. For instance, the variable *ConductivityLayer1* is of type *IfcReal* since it is defined in the parameter property set (#33) as *IfcReal*.

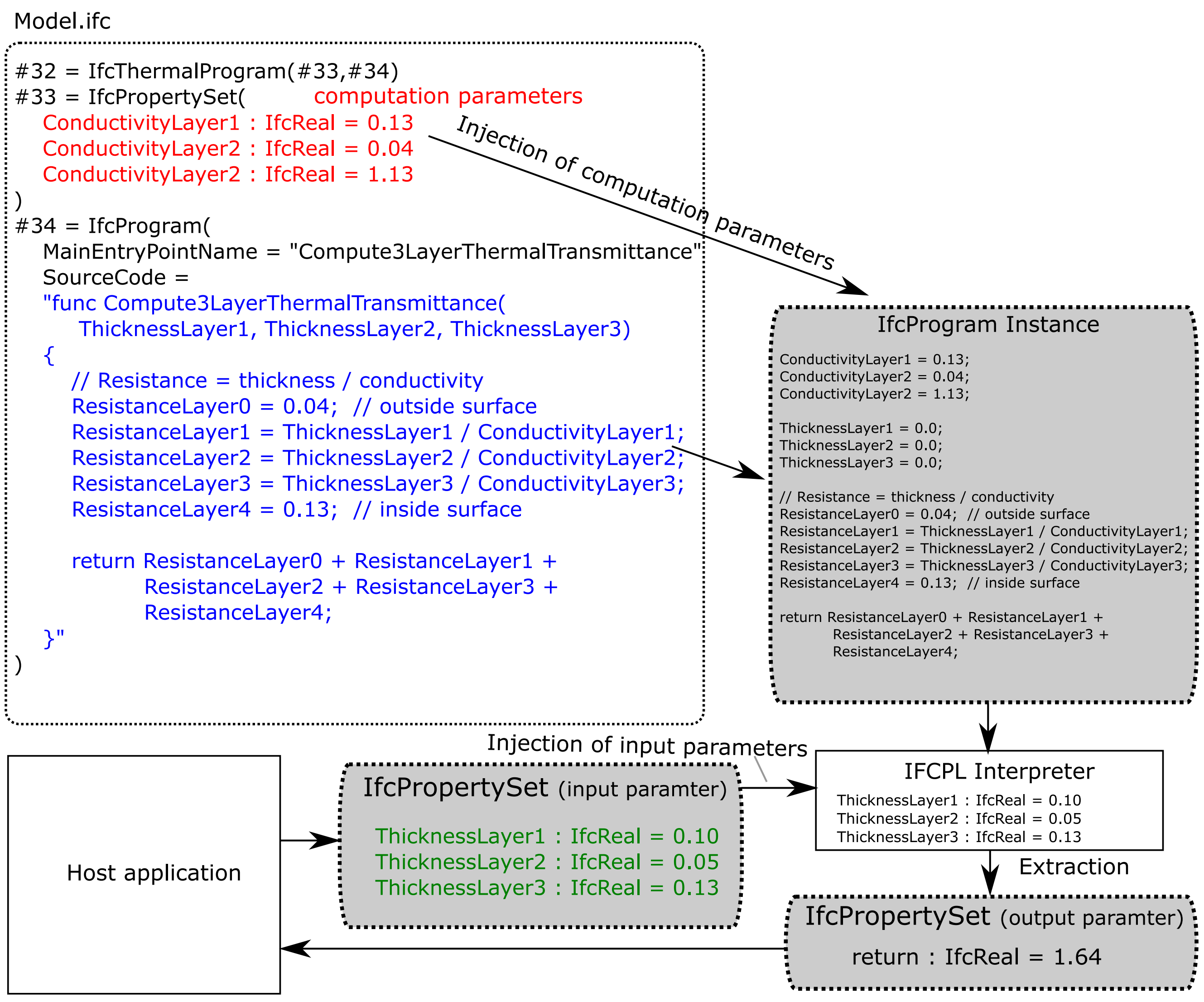
Before a program instance is executed, the first step is to initialize each of the corresponding variables with the values defined in the property set. In the example, the variable *ConductivityLayer1* is set to 0.13.

After the initialization process, the IFCPL program is executed by the IFCPL interpreter. As soon as the return statement of the main function is reached, the program finishes and its result is passed back to the host application through an *IfcPropertySet*. This *IfcPropertySet* is called the output parameter property set and can contain multiple properties since the return statement supports multiple arguments. Return values have the name *returnValue* and the type of the return expression—in the example of the program given earlier, it is an *IfcReal*. If a variable of the input property set is returned, the name of the variable is not *returnValue*; instead, it is added to the output property set with the originally defined name.

IFCPL Interpreter

To execute an IFCPL program, a corresponding interpreter is required. The interpreter must form part of the receiving application. The implementation of an IFCPL interpreter can be supported by standard tools such as lexer and parser generators.

The authors have prototypically implemented an IFCPL environment that consists of a lexer, a parser, and a corresponding interpreter in the C++ programming language. The lexer splits the IFCPL program into a set of tokens (identifiers, keywords, numbers, brackets, braces, etc.). These tokens are then processed by the parser, which generates an abstract syntax tree (AST).



Model.ifc
computation parameters
Injection of computation parameters
IfcProgram Instance
Injection of input parameters
IFCPL Interpreter
Extraction
Host application
IfcPropertySet (input paramter)
IfcPropertySet (output paramter)

```
#32 = IfcThermalProgram(#33,#34)
#33 = IfcPropertySet(
   ConductivityLayer1 : IfcReal = 0.13
   ConductivityLayer2 : IfcReal = 0.04
   ConductivityLayer2 : IfcReal = 1.13
)
#34 = IfcProgram(
   MainEntryPointName = "Compute3LayerThermalTransmittance"
   SourceCode =
   "func Compute3LayerThermalTransmittance(
       ThicknessLayer1, ThicknessLayer2, ThicknessLayer3)
   {
      // Resistance = thickness / conductivity
      ResistanceLayer0 = 0.04;  // outside surface
      ResistanceLayer1 = ThicknessLayer1 / ConductivityLayer1;
      ResistanceLayer2 = ThicknessLayer2 / ConductivityLayer2;
      ResistanceLayer3 = ThicknessLayer3 / ConductivityLayer3;
      ResistanceLayer4 = 0.13;  // inside surface

      return ResistanceLayer0 + ResistanceLayer1 +
             ResistanceLayer2 + ResistanceLayer3 +
             ResistanceLayer4;
   }"
)
```

```
ConductivityLayer1 = 0.13;
ConductivityLayer2 = 0.04;
ConductivityLayer2 = 1.13;

ThicknessLayer1 = 0.0;
ThicknessLayer2 = 0.0;
ThicknessLayer3 = 0.0;

// Resistance = thickness / conductivity
ResistanceLayer0 = 0.04;  // outside surface
ResistanceLayer1 = ThicknessLayer1 / ConductivityLayer1;
ResistanceLayer2 = ThicknessLayer2 / ConductivityLayer2;
ResistanceLayer3 = ThicknessLayer3 / ConductivityLayer3;
ResistanceLayer4 = 0.13;  // inside surface

return ResistanceLayer0 + ResistanceLayer1 +
         ResistanceLayer2 + ResistanceLayer3 +
         ResistanceLayer4;
```

```
ThicknessLayer1 : IfcReal = 0.10
ThicknessLayer2 : IfcReal = 0.05
ThicknessLayer3 : IfcReal = 0.13
```

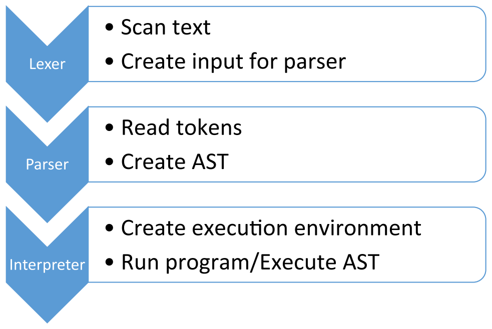
```
ThicknessLayer1 : IfcReal = 0.10
ThicknessLayer2 : IfcReal = 0.05
ThicknessLayer3 : IfcReal = 0.13
```

```
return : IfcReal = 1.64
```

Fig. 7. Overview of the injection and extraction process of variables

Finally, the interpreter walks through the AST executing the nodes, each of which corresponds to the linewise execution of the program code.

The IFCPL environment is provided as a shared library so it can be directly used by other software applications. A C# language binding has also been created for the IFCPL environment. Fig. 8 shows the different phases of the IFCPL execution environment.



Lexer
• Scan text
• Create input for parser
Parser
• Read tokens
• Create AST
Interpreter
• Create execution environment
• Run program/Execute AST

Fig. 8. Different phases of the IFCPL execution environment

The lexer has been generated using *Flex*. *Flex* is a well-known lexer generator (Appel and Ginsburg 2004). Additionally, an IFCPL parser has been generated using *Bison* (Donnelly and Stallman 2003). *Flex* and *Bison* can be easily combined (Aaby 2003). Fig. 9 demonstrates the definition of a number of grammar rules using the *Bison* grammar syntax.

Application Scenario: Alignment Transition Curves

Background

The availability of an imperative program language as part of IFC models enables a large range of possible applications. In this example, the proposed IFCPL is used for the flexible definition of alignment transition curves.

The alignment forms a fundamental basis of any linear infrastructure facility, including roadways, railways, bridges, and tunnels. A first step towards the major effort of including infrastructure facilities in the next version of the IFC data model was the creation of a dedicated alignment project that aims to define data structures to describe alignment in a vendor-independent manner (buildingSMART 2014).

```
whileStmt: TOKEN_WHILE '(' expression ')' statement{$$=makeWhile($3, $6);};

ifStmt: TOKEN_IF  '(' expression ')' statement{$$=makeIf($3, $6);};
      | TOKEN_IF  '(' expression ')' statement TOKEN_ELSE
      statement{$$=makeIf($3, $6, $9);};
      | TOKEN_IF  '(' expression ')' statement TOKEN_ELSE  TOKEN_IF '('
      expression ')' statement TOKEN_ELSE statement{$$=makeElseIf($3,
      $6, $10, $13, $16);};

expression: TOKEN_ID {$$=makeExpByName($1);}
       | TOKEN_NUMBER {$$=makeExpByNum($1);}
       | TOKEN_OPERATOR expression {$$=makeExp(NULL,$2,$1);}
       | expression TOKEN_OPERATOR expression {$$=makeExp($1, $3, $2);}
       | TOKEN_ID BOX_OPEN TOKEN_NUMBER BOX_CLOSE {$$=makeVec1delement($1, $3);};
       | TOKEN_ID BOX_OPEN TOKEN_NUMBER BOX_CLOSE BOX_OPEN TOKEN_NUMBER BOX_CLOSE
       {$$=makeVec2delement($1, $3, $6);}; ...
```

Fig. 9. Definition of three Bison grammar rules

The alignment is typically described by means of two interrelated two-dimensional (2D) curves: the horizontal and the vertical alignment. Each of these 2D curves is composed of several segments. In the horizontal alignment (the focus of this example), the most-typical segment types are linear segments, circular arcs, and transition curves.

The purpose of the transition curve is to provide a continuously changing curvature of alignment to provide driving comfort and reduce accidents (AASHTO 2011). Transition curves are either placed between a sequence of two arcs with different radius, or between an arc and a linear segment.

The most widespread type of transition curve is the clothoid (AASHTO 2011) but numerous other transition curve types are also in practical use in different parts of the world, for example the Wiener Bogen, Bloss curve, cubic spiral, cubic parabola, sinusoidal curve, cosinusoidal curves, sine half-wavelength diminishing tangent curve, Lemniscates curve, or the quadratic spiral. All these curve types require a different set of parameters and different algorithms for their evaluation.

State of the Art: Neutral Data Models for Roadway Alignments

A large number of data formats for exchanging roadway/railway alignment data exist, the most common of which is the LandXML data exchange standard (Rebolj et al. 2008; Ziering et al. 2007). LandXML 1.2 supports 16 different transition curve types.

A transition curve is denoted *Spiral* in LandXML. In addition, LandXML offers a generic template with different parameters, for example the start and end point, length, radius at start point, radius at end point, rotation orientation (clockwise versus counterclockwise), and start and end direction, to name a few. LandXML also permits one to overdetermine the spiral type, for instance, by specifying a clothoid with more parameters than actually needed to reconstruct it in a unique way. Unfortunately, this also allows one to specify impossible transition curves, which violates the principle of data integrity.

Besides LandXML, other standards such as RoadXML (Chaplier et al. 2010; Ducloux and Millet 2009), JHDM (Japan Highways Data Model), or TransXML (Scarponcini 2006) likewise lack construction rules. The official IFC Alignment extension also excludes procedural specifications.

Many other alignment models that were developed in the context of research projects such as IFC-Bridge (Yabuki et al. 2006; Lebegue et al. 2012), IFC-Tunnel (Hegemann et al. 2012; Koch 2008; Yabuki 2009), OpenBrIM (U.S. Department of Transportation and Federal Highway Administration 2013), or IfcAlignment (Amann et al. 2013) also do not contain algorithmic descriptions.

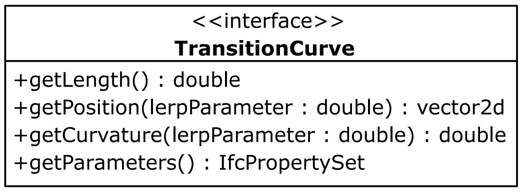
Objektkatalog für das Straßen-und Verkehrswesen (OKSTRA) (Schultze and Buhmann 2008) is a German standard for the data exchange of road design information. In a current research project related to the OKSTRA standard, an approach was developed that was used to describe cross sections by means of OKSTRA RQCode (Singer and Amann 2014). OKSTRA RQCode is not a new programming language, and uses the syntax of Microsoft Visual Basic (Kornbichler 2000). A number of predefined objects such as *RQLine*, *RQBoundaryLine*, or *RQPoint* exist within the OKSTRA *RQCode* environment (Feser et al. 2004). This environment also offers access to the underlying alignment model, making it possible to define the construction of specific cross sections. To simplify matters, the real Visual Basic Runtime is used as an interpreter for *RQCode*, obviating the need for a dedicated *RQcode* interpreter.

Representation of Transition Curves in IFC

With respect to including transition curves in the IFC data schema, there are two general options. The first is to take the conventional data modeling approach and include a preferably large but fixed set of different transition curves as explicit entities of the data model. This approach has a number of shortcomings, including:

- In a specific region or for a specific application scenario, a transition curve might be necessary that has not been included in the schema. For example, Schramm curves are extensively used in German Railway engineering but rarely elsewhere, and Wiener Bogen is mainly used in Austria. Such transition curves are accordingly not represented by the IFC model and the necessary schema extension has to undergo the lengthy standardization process.
- A specific interpretation algorithm has to be implemented for each of the curve types. As this requires significant effort, software vendors will invariably decide to implement only a subset of the defined curve types, resulting in unpredictable incompatibilities between software applications.
- Stakeholders must agree on a common parameter set that is used to describe a specific transition curve.

The second option is to define a generic transition curve capable of describing any of the concrete types listed earlier as well as any disregarded transition curve. This would overcome the first of the aforementioned issues. To resolve the second of the mentioned issues, however, the interpretation algorithms must be included as



```
<<interface>>
TransitionCurve
+getLength() : double
+getPosition(lerpParameter : double) : vector2d
+getCurvature(lerpParameter : double) : double
+getParameters() : IfcPropertySet
```

Fig. 10. Interface for accessing all necessary information

part of the exchanged instance files. This is where the use of IFCPL is proposed.

The provided algorithms must fit the desired information processing needs on the receiving side. By taking a closer look at the purpose of the intended data exchange, the following set of information to be computed/derived from the curve data can be identified:

- Compute the length of a transition curve segment;
- Compute the x, y-coordinates for a given abscissa; and
- Compute the curvature for a given abscissa.

This generic set of information would make it possible to display any transition curve in an arbitrary resolution and forms a suitable basis for most of the engineering tasks involved in roadway or railway design.

Programming Interface

By including the computation procedures (compute length, compute x, y-coordinates, and compute curvature) as part of the exchanged IFC model (instance file), the receiving application is able to access all the necessary information via the interface depicted in Fig. 10. The interface is generic and independent of the concrete type of transition curve represented by the underlying IFC data. A key advantage for most application scenarios is that no additional programming is needed at the receiving application when a new type of transition curve is to be transmitted via the IFC exchange mechanisms.

The getLength() method is used to query the length of a transition curve segment. The method getPosition() is used to retrieve a 2D position on the transition curve (since the transition curve is part of the horizontal alignment, it has a 2D position). The *lerpParameter* passed to the method defines the position along the curve (linear referencing) in a scaled manner. The parameter is assigned a value of between 0 and 1, where 0 refers to the start and 1 to the end position of the curve. For instance, a lerp parameter of 0.3 returns the point reached after travelling 30% of the length of the transition curve. The getParameter() method is used to query the parameters of the transition curve (computation parameters).

Definition of a Generic Transition Curve

This section describes an IFC data model for including a generic version of a transition curve, including the necessary IFCPL programs. The point of departure is formed by the alignment model proposed by the P6 project of buildingSMART.

First, the model is extended with the new entity *IfcArbitraryTransitionCurveSegment2D*. The *IfcArbitraryTransitionCurveSegment2D* is a subclass of *IfcCurveSegment2D*. A parameter of the transition curve is represented as an *IfcProperty*. The different *IfcProperty* values are collected in the *ComputationParameters* property set. Furthermore, *IfcArbitraryTransitionCurve* is associated with three instances of *IfcProgram*. Fig. 11 shows an EXPRESS-G diagram of the different entities described.

The following example illustrates the use of IFCPL for describing how a transition curve is defined. For the sake of simplicity, a straight line has been chosen instead of an actual transition curve as it offers a good and easy way of demonstrating how IFCPL works, even though using a straight line as a transition curve makes little sense in the context of real-world project.

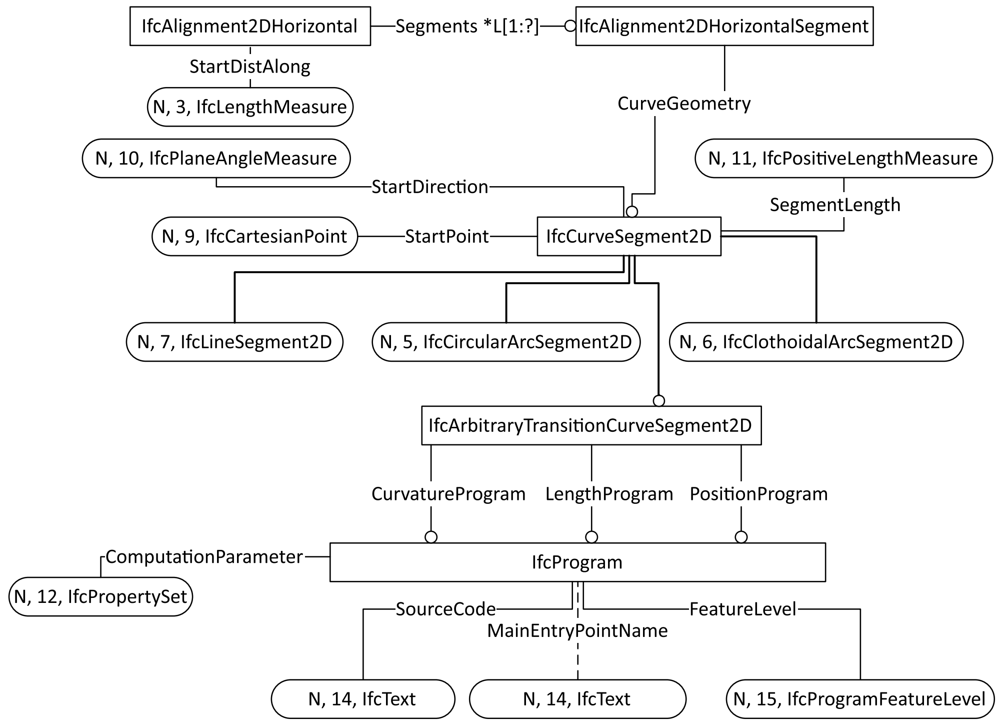


Fig. 11. EXPRESS-G diagram depicting the integration of arbitrary transitions curves into the P6 IFC Alignment schema

```
dx = xEnd - xStart;
dy = yEnd - yStart;
dx2 = dx * dx;
dy2 = dy * dy;
return(sqrt(dx2+dy2));
```

Fig. 12. Computation of the length in IFCPL

It is assumed that the linear segment is defined by its start and end point. For the start point, the xStart and yStart value of type *IfcPositiveLengthMeasure* is stored in the parameter set (*ComputationParameter*) of the *IfcArbitraryTransitionCurveSegment2D*. The same is done for the end point (xEnd and yEnd).

The next step is to create a program that defines the actual computation of the clothoid. To this end, three programs, a LengthProgram, PositionProgram, and CurvatureProgram, have been defined. The length program defines how the length of the transition curve is computed. In the example, the length between two points can be computed using the Euclidian distance. The length program is depicted in Fig. 12.

The tokens and abstract syntax tree of the length program is shown in Fig. 13.

The length program uses the variables xStart, yStart, xEnd, and yEnd. These variables and values are known to the program because all variables contained in the computation parameter set are injected into the length and position program of the transition curve on startup.

```
vx = (1-lerpParamter) * xStart + lerpParamter * xEnd;
vy = (1-lerpParamter) * yStart + lerpParamter * yEnd;
return(vx);
return(vy);
```

Fig. 14. IFCPL position program

Additionally, the lerpParameter variable is injected into the position program. Fig. 14 shows the position program. A return statement adds the computed value to the result property set for further processing by the host application.

The programs shown can be stored in a STEP P21 file and be exchanged between different applications. The receiving application can execute the program by means of an IFCPL interpreter and reconstruct (display) the curve without further knowledge of the concrete type of curve and its internal representation. Fig. 15 shows a pseudo instance file that contains a position program and demonstrates how this program can be reused by different arbitrary transition curves.

Implementation Prototype

To demonstrate the feasibility of the proposed concept, a prototype of this generic approach to handle transition curve types has been implemented by defining a length, position, and curvature program for a clothoid. The exchange of the generically defined transition curves has been tested using several software products, including *Revit*, *NX*, and *Dynamo* (Fig. 16).

Tokens (word, type)	
dx	[IDENTIFIER]
=	[ASSIGN]
xEnd	[IDENTIFIER]
-	[SUB]
xStart	[IDENTIFIER]
;	[SEMICOLON]
dy	[IDENTIFIER]
=	[ASSIGN]
yEnd	[IDENTIFIER]
-	[SUB]
yStart	[IDENTIFIER]
;	[SEMICOLON]
dx2	[IDENTIFIER]
=	[ASSIGN]
dx	[IDENTIFIER]
*	[MUL]
dx	[IDENTIFIER]
;	[SEMICOLON]
dy2	[IDENTIFIER]
=	[ASSIGN]
dy	[IDENTIFIER]
*	[MUL]
dy	[IDENTIFIER]
;	[SEMICOLON]
return	[IDENTIFIER]
(	[BRACKET_OPEN]
sqrt	[IDENTIFIER]
(	[BRACKET_OPEN]
dx2	[IDENTIFIER]
+	[ADD]
dy2	[IDENTIFIER]
)	[BRACKET_CLOSE]
)	[BRACKET_CLOSE]
;	[SEMICOLON]

Abstract Syntax Tree

```
[-]PROGRAM
    [Right]STMNT
        [Left]STMNT
            [Left]STMNT
                [Left]STMNT
                    [Left]STMNT
                        [Right]FCALL
                            [Left]IDENTIFIER
                            [Right]FCALL
                                [Left]IDENTIFIER
                                [Right]ADD
                                    [Left]IDENTIFIER
                                    [Right]IDENTIFIER
                    [Right]ASSIGN
                        [Left]IDENTIFIER
                        [Right]MUL
                            [Left]IDENTIFIER
                            [Right]IDENTIFIER
                [Right]ASSIGN
                    [Left]IDENTIFIER
                    [Right]MUL
                        [Left]IDENTIFIER
                        [Right]IDENTIFIER
            [Right]ASSIGN
                [Left]IDENTIFIER
                [Right]SUB
                    [Left]IDENTIFIER
                    [Right]IDENTIFIER
        [Right]ASSIGN
            [Left]IDENTIFIER
            [Right]SUB
                [Left]IDENTIFIER
                [Right]IDENTIFIER
```

Fig. 13. Token and abstract syntax tree of the length program

```
#7 = IfcProgram = func getPosition(lerpParameter) {
        L = startL_ + (endL_ - startL_) * lerpParameter;

        localPosition = computeLocalPosition(L);

        angle = computeT(startL_, clothoidConstant_);
        localOffset = localPosition - computeLocalPosition(startL_);

        if (!isEntry()) {
               angle *= -1;
               localOffset.x() *= -1;
        }

        if (!counterClockwise_) {
               angle *= -1;
               localOffset.y() *= -1;
        }

        position = startPosition_ +
                   createRotationZ22d(startDirection_ - angle) * localOffset;

        return (position);
     } ... );

#8 = IfcArbitraryTransitionCurveSegment2D(
            #4 /* computation parameters */,
            #5 /* curvature program      */,
            #6 /* length program         */,
            #7 /* position program       */);

#9 = IfcArbitraryTransitionCurveSegment2D(
            #11 /* computation parameters */, // Different computation parameters,
            #5  /* curvature program      */, // but the same programs are reused
            #6  /* length program         */,
            #7  /* position program       */);
```

Fig. 15. A pseudo instance file, which contains a position program and demonstrates how this program can be reused by different arbitrary transition curves

A partial standalone prototype of an IFCPL environment can be found on GitHub (http://github.com/tumcms/BlueCompiler).

Conclusion and Future Work

The paper introduced the IFC Procedural Language (IFCPL), a simple imperative language that makes it possible to capture algorithmic knowledge and exchange it between software applications in a vendor-independent manner. IFCPL programs are not defined on the schema level but at an instance level, which means that programs can be flexibly defined without the need to modify the IFC schema.

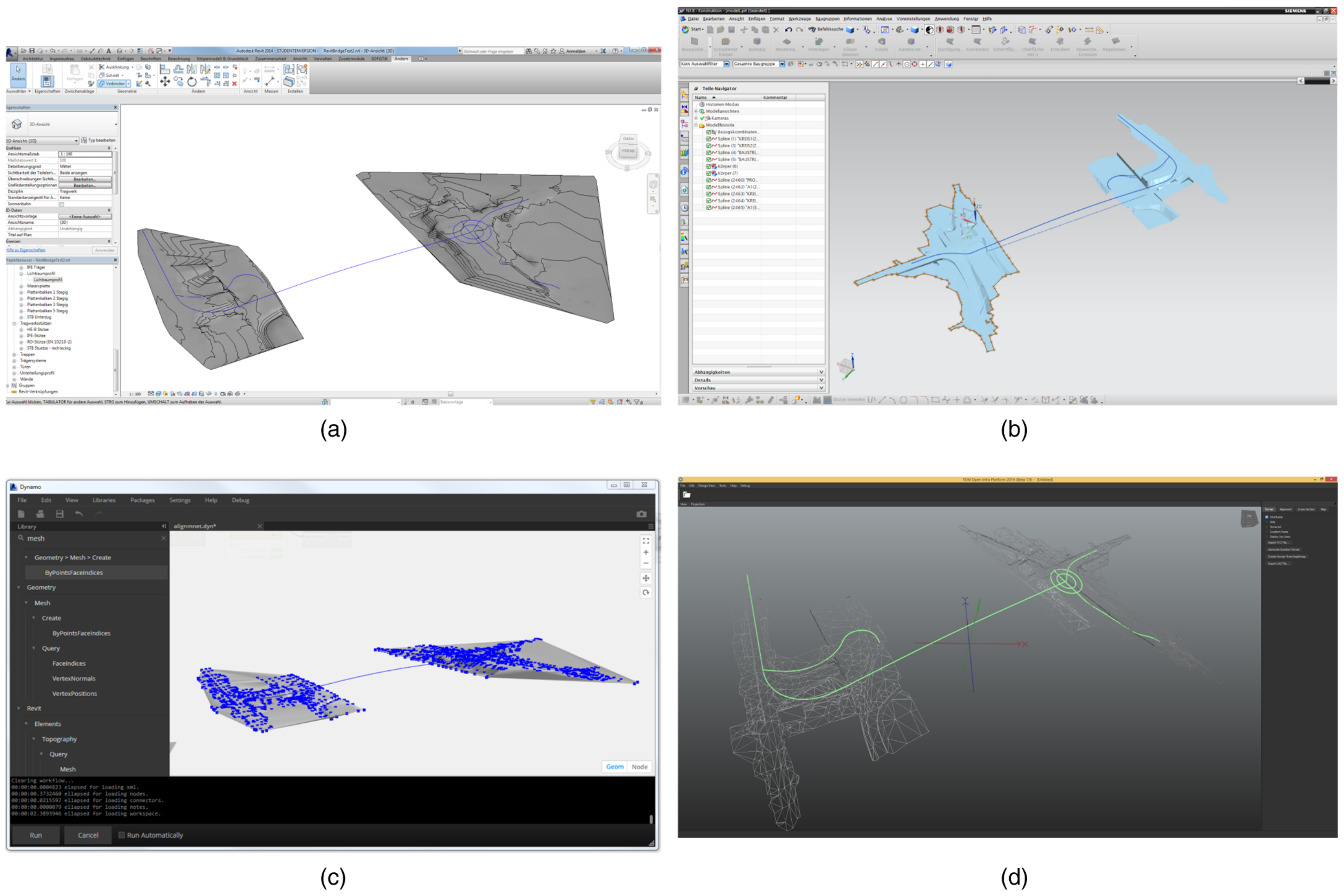
The main purpose of IFCPL is to define algorithms for processing available data and compute new information. This functionality can be used to embed the procedural knowledge required to correctly interpret the transmitted data in the IFC model. This provides a very high degree of flexibility in data exchange scenarios, as data structures and their correct interpretation do not have to be defined a priori in the schema definition, but can be dynamically specified by the sending application at the time of data transmission. The embedding of the interpretation algorithm also implies that potential data interpretation errors by different software developers are avoided.

To illustrate the advantages of IFCPL, the paper discussed its application for the flexible definition of horizontal alignment transition curves. As a very large set of transition curves are used in practice worldwide, including them all in the IFC standard is not a feasible option. Using IFCPL, arbitrary types of transition curves can be described and exchanged. The IFCPL programs embedded in the corresponding IFC instance models provide the algorithms required to determine the length of the curve, to compute the x, y coordinates for a given abscissa and to calculate the curvature at any given point. These three functions cover the majority of typical application scenarios, ranging from the pure display of the curve to determining earthwork haulage and computing perpendicular forces.

The described concept is generic and can also be applied to elements of the vertical alignment such as parabolas. It can likewise support the description of parameterized curves for the structural engineering domain. For example, this approach could be used to describe B-Spline or nonuniform rational basis spline (NURBS) curves or surfaces.

The introduction of a programming language for IFC reveals possibilities that go far beyond the description of curves. More advanced applications range from embedding quantity take-off rules to the software-independent description of the behavior of parameterized BIM objects. These and other fascinating application areas will be addressed in future publications.

The only limitation of the proposed approach is that a dedicated interpreter is necessary on the receiving side. As this could represent a hurdle to the adoption of the IFCPL language, a logical proposal would be that the IFC standardization committee should provide a reference implementation that can be used free of charge for any commercial or noncommercial software development project. The prototype interpreter developed in the framework of the research project presented here could provide a suitable basis for this reference implementation.



```
Clearing workflow...
00:00:00.0004823 elapsed for loading xml.
00:00:00.3732460 ellapsed for loading nodes.
00:00:00.0215597 ellapsed for loading connectors.
00:00:00.0000079 ellapsed for loading notes.
00:00:02.3893946 ellapsed for loading workspace.
```

Fig. 16. Arbitrary transition curves exchanged between different applications: (a) Autodesk *Revit* (with permission from Autodesk); (b) Siemens *NX* (with permission from Siemens); (c) *Dynamo* (with permission from Autodesk); (d) *TUM* open infraplatform (with permission from TUM)

Besides addressing new use cases for IFCPL in the future, the extension of the syntax of IFCPL to include object-oriented features (information hiding, inheritance and polymorphism) is planned along with additional useful features such as an embedded spatial query language (Daum and Borrmann 2013). Furthermore, the speed of the IFCPL runtime library will be optimized to support more computationally intensive use cases using the IFCPL (e.g., radiosity computation for lighting design studies). Currently, the simple interpreter used to interpret IFCPL programs is, compared with binary translated interpreters, rather slow. This situation could be improved by using the LLVM Compiler Infrastructure (Lattner and Adve 2004). The LLVM Compiler Infrastructure provides a virtual machine that can execute a virtual assembler language. The virtual machine of LLVM also needs to interpret assembler language, but also provides sophisticated optimization techniques such as binary translation (Smith and Nair 2005) to improve execution speed.

Different software implementers will, of course, implement similar IFCPL programs. In the proposed concept, libraries of basic and advanced IFCPL functions will evolve through contributions by the community. To this end, a form of modularization is proposed. Each module would be a text file containing IFCPL source code. A module is imported using the import-Statement (e.g., *import Math.ifcpl;*), making it possible to collect common code in an IFCPL standard library. This IFCPL standard library can eventually become part of the IFC standard and can be distributed along with the IFC schema files. This additionally reduces the amount of duplicated and redundant work, and at the same time the IFC community can benefit from the advantages of IFCPL discussed in this paper.

References

Aaby, A. (2003). "Compiler Construction using Flex and Bison." Walla Walla College, College Place, WA.

AASHTO. (2011). "A policy on geometric design of highways and streets." Washington, DC.

Amann, J., et al. (2013). "A refined product model for shield tunnels based on a generalized approach for alignment representation." *Proc., 1st Int. Conf. on Civil and Building Engineering Informatics (ICCBEI 2013)*, ICCBEI Organizing, Tokyo, 8.

Amor, R., Jiang, Y., and Chen, X. (2007). "BIM in 2007—Are we there yet?" *Proc., CIB W78 Conf. on Bringing ITC Knowledge to Work*, Maribor, Slovenia.

Appel, A. W., and Ginsburg, M. (2004). *Modern compiler implementation in C*, Cambridge University Press, New York.

ArchiCAD 18 [Computer software]. GRAPHISOFT, Waltham, MA.

buildingSMART. (2014). "Summary of IFC releases." ⟨http://www.buildingsmart-tech.org/specifications/ifc-releases/summary⟩ (Apr. 18, 2016).

buildingSMART alliance. (2015). "National BIM standard—United StatesTM version 3." ⟨http://www.nationalbimstandard.org/⟩ (Apr. 18, 2016).

CATIA [Computer software]. Dassault Systèmes, Vélizy-Villacoublay, France.

Chaplier, J., et al. (2010). *Toward a standard: RoadXML, the road network database format*, 211–220.

Daum, S., and Borrmann, A. (2013). "Processing of topological BIM queries using boundary representation based methods." *Adv. Eng. Inform.*, 28(4), 272–286.

Donnelly, C., and Stallman, R. (2003). "Bison manual: Using the YACC-compatible Parser generator." Free Software Foundation, Boston.

Ducloux, P., and Millet, G. (2009). *Road network description XML format specification*, 1–88.

Dynamo [Computer software]. Autodesk, San Rafael.

East, E. W. (2007). "Construction operations building information exchange (COBIE): Requirements definition and pilot implementation standard." *ERDC/CERL TR-07-30*, Construction Engineering Research Laboratory, Champaign, IL.

Eastman, C., et al. (2005). "Deployment of an AEC industry sector product model." *Comput.-Aided Des.*, 37(12), 1214–1228.

Eastman, C., Teicholz, P., Sacks, R., and Liston, K. (2011). *BIM handbook: A guide to building information modeling for owners, managers, designers, engineers and contractors*, Wiley, Hoboken, NJ.

Eastman, C. M. (1999). *Building product models: Computer environments supporting design and construction*, CRC Press, Boca Raton, FL.

Eisenberg, A. (1996). "New standard for stored procedures in SQL." *ACM SIGMOD Rec.*, 25(4), 81–88.

Feser, B., Kornbichler, D., and Rosenthal, R. (2004). *Entwicklung des Objektes "Dynamisches Querprofil*, Wirtschaftsverl. NW, Verl. für Neue Wiss, Bremerhaven, Germany (in German).

flex 2.5.37 [Computer software]. Win Flex-Bison, ⟨https://sourceforge.net/projects/winflexbison/⟩ (Apr. 18, 2016).

Fowler, M. (2004). *UML distilled: A brief guide to the standard object modeling language*, 3rd Ed., Pearson Paravia Bruno Mondad, Boston.

Hegemann, F., Lehner, K., and König, M. (2012). "IFC-based product modeling for tunnel boring machines." *eWork and eBusiness in architecture, engineering and construction*, Balkema, Netherlands, 289.

Hubers, J. (2010). "IFC based BIM or parametric design?" ⟨http://www.engineering.nottingham.ac.uk/icccbe/proceedings/pdf/pf73.pdf⟩ (Dec. 3, 2014).

International Organization for Standardization. (1995). "Standard for the exchange of product model data." *ISO 10303*, Geneva.

ISO/IEC. (1996). "Information technology—Syntactic metalanguage—Extended BNF: First edition." *ISO/IEC 14977:1996(E)*, ⟨http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip⟩ (Apr. 18, 2016).

ISO/IEC. (2011). "Information technology—Database languages—SQL." *ISO/IEC 9075:2011*.

Ji, Y., et al. (2013). "Exchange of parametric bridge models using a neutral data format." *J. Comput. Civ. Eng.*, 10.1061/(ASCE)CP.1943-5487.0000286, 593–606.

Koch, C. (2008). "Bauwerksmodellierung im kooperativen Planungsprozess: Mit der Objektorientierung zur Verarbeitungsorientierung." Dissertation, Bauhaus-Universitäts, Weimar, Germany (in German).

Kornbichler, D. (2000). "Zwischenbericht zur Geometrischen Modellierung." Forschungs- und Entwicklungsauftrag FE 09.122/2000/DGB OKSTRA Dynamisches Querprofil, Bundesanstalt für Straßenwesen, Bergisch Gladbach, Germany.

Lattner, C., and Adve, V. (2004). "LLVM: A compilation framework for lifelong program analysis and transformation." *Int. Symp. on Code Generation and Optimization, CGO*, IEEE, San Jose, CA, 75–86.

Lebegue, E., Fiês, B., and Gual, J. (2012). "IFC-bridge V3 data model–IFC 4." buildingSMART, IAI French, France.

Lee, G., Eastman, C. M., Sacks, R., and Navathe, S. B. (2006a). "Grammatical rules for specifying information for automated product data modeling." *Adv. Eng. Inform.*, 20(2), 155–170.

Lee, G., Sacks, R., and Eastman, C. (2006b). "Specifying parametric building object behavior (BOB) for a building information modeling system." *Autom. Constr.*, 15(6), 758–776.

Liebich, T., and Weise, M. (2013). "ifcXML4 specification methodology." *Model support group (MSG) of buildingSMART international*, ⟨http://www.buildingsmart-tech.org/downloads/ifcxml/ifcxml4/ifcxml4_specification_methodology_v1-1.pdf⟩ (Apr. 13, 2016).

Mazairac, W., and Beetz, J. (2013). "BimQL: An open query language for building information models." *Adv. Eng. Inf.*, 27, 444–456.

MicroStation version 10.0 [Computer software]. Bentley Systems, Exton.

Mitchell, J. C. (2002). *Concepts in programming languages*, Cambridge University Press, New York.

Nisbet, N., and Liebich, T. (2009). "ifcXML implementation guide." *buildingSMART*, ⟨http://dl.acm.org/citation.cfm?id=1593063⟩ (Dec. 3, 2014).

NX 8.5 [Computer software]. Siemens PLM, Plano, TX.

Rasdorf, W. J. (1985). "Perspectives on knowledge in engineering design." *Proc., 1985 ASME Int. Computers in Engineering Conf. and Exhibition*, NASA.

Rebolj, D., et al. (2008). "Development and application of a road product model." *Autom. Constr.*, 17(6), 719–728.

Revit [Computer software]. Autodesk, San Rafael.

Rhino 5.1 [Computer software]. McNeel Europe, Barcelona, Spain.

Ritter, F., Trautmann, B., Podgorski, C., and Borrmann, A. (2013). "Automated design space exploration for improved early-stage decision-making." *Proc., Conf. sb13 Munich Implementing Sustainability—Barriers and Chances*, Fraunhofer IRB, Stuttgart, Germany.

Scarponcini, P. (2006). "TransXML: Establishing standards for transportation data exchange." *Proc., Joint Int. Conf. Computing and Decision Making in Civil and Building Engineering (CIB-W78 2006)*.

Schenck, D. A., and Wilson, P. R. (1994). *Information modeling: The EXPRESS way*, Oxford University Press, New York.

Schultze, C., and Buhmann, E. (2008). "Developing the OKSTRA® standard for the needs of landscape planning in context of implementation for mitigation and landscape envelope planning of road projects." *Int. Conf. Information Technologies in Landscape Architecture*, Wichmann, Heidelberg, Germany.

Sengupta, R., and Amann, J. (2015). "BlueCompiler." ⟨https://github.com/tumcms/BlueCompiler⟩ (Apr. 18, 2016).

Singer, D., and Amann, J. (2014). "Erweiterung von IFC Alignment um Straßenquerschnitte." *Proc., 26th Forum Bauinformatik*, Shaker, Aachen, Germany (in German).

Smith, J., and Nair, R. (2005). *Virtual machines. Versatile platforms for systems and processes (The Morgan Kaufmann series in computer architecture and design)*, Morgan Kaufmann.

U.S. Department of Transportation and Federal Highway Administration. (2013). "Open BrIM standards." Washington, DC.

Waterhouse, R., et al. (2014). "NBS national BIM report." RIBA, U.K.

Weise, M., Liebich, T., and Wix, J. (2009). "Integrating use case definitions for IFC developments." ⟨http://www.inpro-project.org/media/Integrating_UseCase_def.pdf⟩ (Dec. 2, 2014).

Woodbury, R. (2010). *Elements of parametric design*, Taylor and Francis.

Xu, X., and Nee, A. Y. C. (2009). *Advanced design and manufacturing based on STEP*, 1st Ed., Springer, Berlin.

Yabuki, N., et al. (2006). "International collaboration for developing the bridge product model IFC-bridge." *Proc., Joint Int. Conf. on Computing and Decision Making in Civil and Building Engineering (ICCCBE)*, Montreal.

Yabuki, N. (2009). "Representation of caves in a shield tunnel product model." *Proc., 7th European Conf. on Product and Process Modelling (ECPPM2009)*, Balkema, Netherlands.

Ziering, E., Harrison, F., and Scarponcini, P. (2007). "TransXML: XML schemas for exchange of transportation data." *NCHRP Rep. 576*, Transportation Research Board, Washington, DC.