Graph representations and methods for querying, examination, and analysis of IFC data

H. Tauscher
National University of Singapore

J. Crawford
Ordnance Survey, UK

ABSTRACT: This paper presents an approach to apply graph-based methods to IFC data. We discuss (1) the representation of object-oriented data and schemas as graphs including IFC-specific issues, (2) the specification of IFC queries as subgraph templates including their conversion to Model View Definition (MVD) fragments, BIMserver and Cypher queries, (3) graph grammars for IFC model transformation, and (4) visualization schemas for the different graph structures. These fundamentals provide the basis for future work in graph-based model transformation.

1 INTRODUCTION

We are considering graph-based transformation methods for the conversion of BIM data in IFC format into GIS data in CityGML format. Previous attempts at the conversion from IFC to CityGML have shown that no reliable solutions are possible without the employment of formal methods (Isikdag & Zlatanova 2009, e.g.).

We specifically selected graph-based transformation methods since the sound mathematical model behind this approach allows to tackle questions of correctness, completeness, and consistency of transformations. Moreover, the rule-based nature of graph-based transformation methods works well with declarative specifications rather than imperative and allows thus an incremental development of the complex conversion process.

For the conversion of given IFC data into CityGML we have to consider three different graphs: the graph of the given IFC entities, the graph of CityGML entities to be created, and the graph of correspondence relations. In this paper we are focussing on the IFC-graph only.

1.1 *Related work*

Building information models lend themselves to graph methods due to their object-oriented nature.

Several other researchers have recently studied the application of graph-based methods to BIM data in general, e.g. Isaac, Sadeghpour, & Navon (2013) propose applications of graph theory for construction management. Vilgertshofer & Borrmann (2017) apply graph rewriting to generate parametric infrastructure models. Others targets IFC data in particular: Khalili & Chua (2015) trys to use graph methods for topological IFC queries. Tauscher, Bargstädt, & Smarsly (2016) suggest to employ Dijstra's shortest path algorithm to simplify queries. Ismail, Nahar, & Scherer (2017) dump IFC data into neo4j for further processing.

1.2 *Method and structure of the work*

We investigate graph representations of IFC data and subsequently three different application scenarios for graph methods: queries with graph templates, decomposition with graph grammars, analysis with graph visualization. For each of these cases, we describe our approach and technology choices and compare alternatives.

First, in Section 2, we investigate how to represent object-oriented data, particularly the IFC schema and IFC object graphs, as typed directed graphs. For a reference implementation in a graph database we use Neo4J.

Second, in Section 3, we suggest subgraph templates as a generic way to express queries on IFC object graphs. We derive BIMserver queries, Neo4J cypher queries and MVD fragments from these templates, compare the three derivations regarding their expressiveness and evaluate the performance of queries on a BIMserver instance and a

Neo4J database.

Subgraph templates are only capable of describing local fragments. With a graph grammar however, it is possible to capture the complete structure common to a set of graphs. To tap the grammar approach for IFC data, we derive a graph grammar from the IFC schema in Section 4. This grammar can then be used to generate valid instance graphs, but also to decompose a given graph, that is to parse the graph according to the grammar.

Finally, in Section 5, we aim to employ graph representations to foster visual inspection of IFC data sets, but also of operations carried out on them and analysis or validation results. To this end, we introduce a graphical representation schema for the different graph structures presented before: for schema and instance graphs, sub graph templates, as well as for transformation rules.

1.3 *Technology choices and sample data*

As part of Section 3 we present a sample implementation of the graph model and the subgraph template-based query process. We are using neo4J (Community edition, version 3.2.5) as a graph database and BIMserver 1.5.95 to store IFC data. As sample data set we exported the Revit advanced tutorial model with the default IFC4 (ISO 2013) Reference view settings. The resulting IFC file has 21 MB and contains 144521 IFC entities.

2 GRAPH REPRESENTATIONS OF OBJECT-ORIENTED DATA

For the graph representation of IFC schema and data, we identified key design issues: the representation of edges and attributes in the mathematical model, the treatment of objectified relations, the representation of lists, and the integration of the instance and metamodel graph. We are first focussing on the graph model of an IFC instance graph and then proceed integrating the graph model of the IFC schema.

2.1 *Mathematical model of nodes and edges*

To represent an IFC instance graph, we model entities as nodes and their entity data typed attributes as edges. For now we are excluding simple, enumeration, defined, and select data types [1]. A common mathematical definition of a graph G is $G = (N, E)$ with a set of nodes N and set of edges E. There are multiple ways to express the relation between edges and nodes.

The native model defines a directed edge $e \in E$ as the pair of its source node and target node: $(n_S, n_T) \in N \times N$. This node edge model is used by Isaac, Sadeghpour, & Navon (2013) and Tauscher, Bargstädt, & Smarsly (2016). This model does not allow to express parallel edges between nodes, since those would be represented by the same tuple of nodes and could thus not be distinguished. However, investigating some IFC files using the Perl style regular expression `(#\d+)(?=\D.*\1\D)` revealed some cases where duplicate edges may exist: For example an `IfcPolyline` entity's `Points` attribute may contain the same `IfcCartesionPoint` entity as first and last element of the list for closed polylines. Another example is `IfcSurfaceStyleRendering`, where the attributes `SurfaceColour` and `DiffuseColour` may both refer to the same `IfcColourRgb` entity. We thus have to use a slightly more general model of directed edges. With source and target functions $s, t : E \to N$ assigning source or target nodes to edges we can describe parallel edges.

This can be mitigated by including edge labels in the tuples [2], but a more flexible way to model labelling of graph edges is through labelling functions which would not heal this shortcoming.

In Express, there are three types of attributes: explicit, derived, and inverse. We are currently excluding derived attributes. Inverse attributes are inferred from other explicit attributes relating to entities and are thus redundant. They are specified in order to name and constrain them. To facilitate navigation, we are adding inverse edges to our graph model when they are specified in the schema. An edge e_1 and its inverse edge e_2 fulfill $s(e_1) = t(e_2)$ and $s(e_2) = t(e_1)$.

2.2 *Treatment of objectified relations*

In the IFC schema, relations between entities of the core layer and above (shared and domain specific schemas), that is subtypes of `IfcRoot`, are modelled as entities of type `IfcRelationship`. Such a relationship entity references the relating and related entities through its attributes. In contrast, entities of the resource layer reference each other and are referenced by higher level entities directly via their attributes, without a mediating relationship entity.

We are representing the objectified relationships as nodes. That is, we have a dedicated set $N_R \subset N$ of relationship nodes and two sets $E_S, E_T \subseteq E$ of edges that connect to the relating and related nodes respectively. For each objectified relationship between two non-relationship nodes $n_S, n_T \in N \setminus N_R$, we then have a node $n_R \in N_R$ and two

[1] Select data types will pose an issue to node attribution, because they may resolve to both entity data types or defined data types which will be modelled differently.

[2] The resulting edge representation adheres more to the structure of a Resource Description Framework (RDF) triple.

edges $e_S \in E_S$ and $e_T \in E_T$ such that $s(e_S) = s(e_T) = n_R$, $t(e_S) = n_S$, and $t(e_T) = n_T$.

Some graph approaches to IFC (e.g. Isaac, Sadeghpour, & Navon (2013), Ismail, Nahar, & Scherer (2017)) collapse objectified relations into edges instead of modelling them as nodes. For the relationship between nodes n_S and n_T they would then just have an edge e_R with $s(e_R) = n_S$ and $t(e_R) = n_T$.

In collapsing the edges, IFC information will be omitted from the graph representation for those relations that contain additional information. Examining IFC4 Add2 TC1 revealed a list of 15 subtypes of `IfcRelationship` that have at least one attribute in addition to the relating and related entities, which would be lost after collapsing. In total we found 27 such attributes, one of which is mandatory (`IfcInterferesElements.ImpliedOrder`). A prominent example is `IfcRelSpaceBoundary`.

2.3 *Representation of collections: sets and lists*

Although Express knows four kinds of aggregation data types — SET, BAG, LIST, ARRAY — (ISO 2004), in IFC only two of them are used. Thus we will limit our elaboration to ordinary sets and lists and ignore multisets (BAG)[3] and lists with unset elements (ARRAY).

Attributes of SET type (that is unordered aggregations) are represented as multiple edges of the same edge type with a common source node. Given an entity with an attribute having a set of i entities as its value, we will then have one node n_0 for the initial entity, a set $N_A = \{n_1, \ldots, n_i\}$ of nodes for the attribute value set, and for each $n \in N_A$ we have an edge $e \in E$ with $s(e) = n_0$, $t(e) = n$.

For lists, we need to represent order of nodes, which is not a native concept in graph theory. Pauwels, Terkaj, Krijnen, & Beetz (2015) discuss a similar problem for OWL. There are two fundamentally different ways to solve this. The first possibility is to construct a graph structure that connects successive elements, thus constructing something similar to the basic data type of a linked list. The second possibility is to enhance the node–edge construction for sets with an edge label denoting the position of the element, thus constructing something similar to the basic data type of an array with an index. The index is modelled as a labelling function $i : E \to \mathbb{N}$ over the range of the natural numbers.

Technically, the indexed representation would be more appropriate for an ARRAY type aggregate whereas the linked representation would better match the LIST type aggregate. However we have tentatively decided to use an indexed representation in our model for the sake of convenient direct element access [4].

[3]Note that the BAG type would result in duplicate edges between two nodes with the same label, which would definitely require to model edges as described in 2.1.

2.4 *Integration of instance and metamodel graph*

In the graph theoretic approach, the schema is also represented as a graph similar to the instance graph. Each entity type is represented as a node, while each entity data typed attribute is represented as a directed edge. These sets of entity type nodes N_{IFC} and attribute edges E_{IFC} with source and target functions $s_{IFC} : E_{IFC} \to N_{IFC}$ and $t_{IFC} : E_{IFC} \to N_{IFC}$ form the type graph $IFC = (N_{IFC}, E_{IFC}, s_{IFC}, t_{IFC})$.

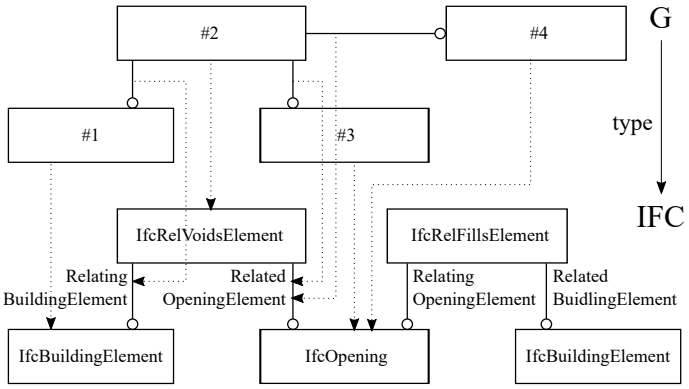

Figure 1: Type and instance graph with typing morphism

The relation between the instance and the type graph is modelled as a graph morphism, that is two functions mapping the nodes and edges of the graphs such that the graph structure is maintained. Given an instance graph $G = (N, E, s, t)$ and a type graph $IFC = (N_{IFC}, E_{IFC}, s_{IFC}, t_{IFC})$ this morphism is defined as $type = (G, IFC, m_N, m_E)$ with the two functions $m_N : N \to N_{IFC}$ and $m_E : E \to E_{IFC}$. To preserve the graph structure, it must hold that $s_{IFC} \circ m_E = m_N \circ s$ and $t_{IFC} \circ m_E = m_N \circ t$. Together, an instance graph G, a type graph IFC and a morphism $type$ form the typed graph $G_{IFC} = (G, IFC, type)$. We are speaking of a graph G typed over a type graph IFC and write short $type : G \to IFC$ for the morphism. Figure 1 shows the morphism between type and instance graph.

Inheritance can be represented as dedicated graph $I = (N_{IFC}, E_I, s_I, t_I)$ which adds another set of directed edges E_I to the nodes N_{IFC} of the type graph with their target nodes being supertypes of their source nodes. This graph must be acyclic and in the case of IFC even forms a rooted in-tree, since multiple inheritance is not used in IFC (although theoretically possible in STEP). A subset $A \subseteq N_{IFC}$ is considered as abstract nodes which may not be instantiated, that is for any

[4]Those familiar with Lisp data structures will know the beauty and pain of linked lists.

typed instance graph $G_{IFC} = (G, IFC, type)$ they shall not appear in the codomain of any type morphism's function m_N, that is $\forall n \in N\ \forall a \in A : m_N(n) \neq a$.

When inheritance is modelled like this, the inheritance tree has to be flattened in order to obtain a typing morphism as described above. Intuitively speaking this is done by duplicating all edges for all subtype nodes in the type graph. Alternatively, instead of the morphism to the flattened type graph, an equivalent mapping to higher levels of the inheritance tree called *clan-morphism* can be used. This model for typed graphs with inheritance was first described by Bardohl, Ehrig, de Lara, & Taentzer (2004).

2.5 *Implementation in neo4J*

The implementation in neo4J is straight-forward according to the mathematical model described above. We created a BIMserver serializer to generate a Cypher query that creates the respective graph in a Neo4J database.

Currently, we are not creating the type graph in neo4J, but use labels to denote the concrete types of nodes and edges. This labelling is essentially a function from nodes and edges to the sets of entity type and attribute name labels respectively. Ehrig, Ehrig, Prange, & Taentzer (2006) has shown that labelling is equivalent to typing over a basic type graph (without inheritance) using a morphism. In order to address abstract supertypes in queries or other graph processing, we will either have to resort to the type graph with flattened inheritance or use multi labels in neo4J. We hypothesize that multi-labelling of supertypes can be shown to be equivalent to clan-morphism.

As described above, we are explicitly generating nodes for objectified relationships. The creation of edges for inverse relations is implemented as a serializer option. List order is persisted as an edge attribute *i*, that is essentially a named labelling function denoting an Integer index.

3 SUBGRAPH TEMPLATES FOR MODEL REDUCTION AND CHECKING

For model reduction we want to be able to identify specific parts of the graph that we want to retain or remove. For model checking we want to verify the existence or non-existence of specific parts of the graph. This section presents a method to specify such parts as *graph templates*, shows how these relate to query languages and MVDs, and finishes with a comparison of BIMserver queries with Neo4J cypher queries.

3.1 *Mathematical model*

To specify a part of an IFC typed graph, we use another IFC typed graph as a *template*. In order to identify a corresponding occurrence we are then looking for a morphism from the template graph to the instance graph. Such a morphism is called a *match*.

Given an IFC-typed graph $G_{IFC} = (G, IFC, type_G)$ and an IFC-typed template graph $T_{IFC} = (T, IFC, type_T)$ a match is a morphism $m : T \to G$ such that $type_T \circ m = type_G$. Intuitively this means that the matched nodes and edges in the entity graph G are of the same type as their counterparts in the template graph T. Figure 2 shows and example of a match morphism.

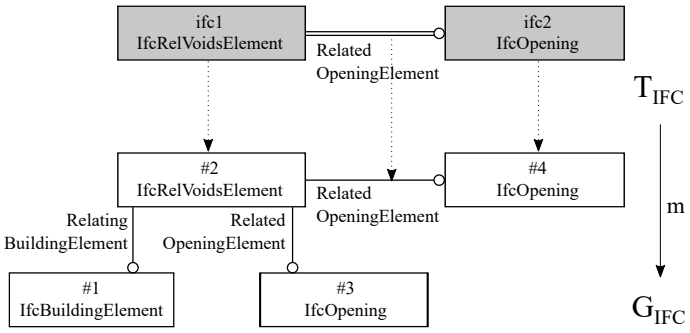

Figure 2: Template, instance graph and match morphism

3.2 *Subgraph definition with a DSL*

Despite previous attempts to standardize an exchange format for graphs (Taentzer 2001, Winter 2002), we are defining our own DSL in favour of a more compact notation than XML and more flexibility for the implementation of applications based on graph theory. Below the notation is described in Extended Backus-Naur form (EBNF).

```
lb ::= NEWLINE | ';'
template ::= 'graph {' nodes edges '}'
nodes ::= 'nodes {' nodeType { lb nodeType } '}' lb
edges ::= 'edges {' edgeType { lb edgeType } '}' lb
nodeType ::= ifcType '{' node { lb node } '}' lb
edgeType ::= ifcType '= [' edge { ',' edge } ']' lb
node ::= id, '{' [attr, { lb attr }]  '}'
edge ::= '[' id ',' id, ']'
```

An example subgraph involving walls with their openings and filling doors reads as follows:

```
graph {
  nodes {
    IfcWall { wall {} }
    IfcRelVoidsElement { void {} }
    IfcOpeningElement { opening {} }
    IfcRelFillsElement { filling {} }
    IfcDoor { door {} }
  }
  edges {
    RelatingBuildingElement = [[void, wall]]
    RelatedOpeningElement = [[void, opening]]
    RelatingOpeningElement = [[filling, opening]]
    RelatedBuildingElement = [[filling, door]]
  }
}
```

Note that we are only using direct attribute names, not the inverse ones, but we could as well replace those with inverted edges.

3.3 *Conversion*

In this section we evaluate conversion of subgraph templates to MVD-XML fragments, BIMserver queries, and neo4J Cypher queries.

3.3.1 *MVD-XML*

This is an equivalent MVD-XML fragment:

```
<ConceptTemplate applicableEntity="IfcWall"><Rules>
  <AttributeRule AttributeName="HasOpenings"><EntityRules>
    <EntityRule EntityName="IfcRelVoidsElement"><AttributeRules>
      <AttributeRule AttributeName="RelatedOpeningElement"><EntityRules>
        <EntityRule EntityName="IfcOpeningElement"><AttributeRules>
          <AttributeRule AttributeName="HasFillings"><EntityRules>
            <EntityRule EntityName="IfcRelFillsElement"><AttributeRules>
              <AttributeRule AttributeName="RelatedBuildingElement"><EntityRules>
                <EntityRule EntityName="Door" />
              </EntityRules></AttributeRule>
            </AttributeRules></EntityRule>
          </EntityRules></AttributeRule>
        </AttributeRules></EntityRule>
      </EntityRules></AttributeRule>
    </AttributeRules></EntityRule>
  </EntityRules></AttributeRule>
</Rules></ConceptTemplate>
```

The reader is invited to compare this to the "Element Voiding" concept in the "General usage" MVD. Note that due to the tree-like structure of the specification, MVDs are confined to use inverse relation attributes. This applies to BIMserver queries as well.

3.3.2 *BIMserver query*

A similar BIMserver query would look like this:

```
{
  "type": {
    "name": "IfcWall"
  },
  "include": {
    "type": "IfcWall",
    "field": "HasOpenings",
    "include": {
      "type": "IfcRelVoidsElement",
      "field": "RelatedOpeningElement",
      "include": {
        "type": "IfcOpeningElement",
        "field": "HasFillings",
        "include": {
          "type": "IfcRelFillsElement",
          "field": "RelatedBuildingElement"
        }
      }
    }
  }
}
```

3.3.3 *Neo4J Cypher query*

With Neo4J we can use both queries with or without inverses. Here we show the version with inverses.

```
(:IfcWall)-[:HasOpenings]->(:IfcRelVoidsElement)
  -[:RelatedOpeningElement]->(:IfcOpeningElement)
  -[:HasFillings]->(:IfcRelFillsElement)
  -[:RelatedBuildingElement]->()
```

3.3.4 *Discussion*

While we can translate the graph template into a corresponding MVD fragment, BIMserver query and Neo4J Cypher query, there are subtle but essential differences in the meaning of these statements even though they appear similar. While our graph templates identify a set of matches, a BIMserver query does just restrict the whole object graph without identifying matches. The different purpose of defining what to include in an export is reflected in the term "filter language" instead of "query language". Neo4J can assemble result records similar to our graph template approach, but also create the restricted graph. For MVD definitions it depends on the use case: MVDs deliberately allow to be interpreted differently depending on whether they are used for filtering, validation, or documentation.

3.4 *Comparison of BIMserver and neo4J*

In the previous section we focussed on the expressiveness of different query DSLs, now we are going to compare the performance of IFC queries on BIMserver and neo4J.

3.4.1 *Test setup*

To execute the tests, queries are issued through the REST interfaces of BIMserver and neo4J running on local machine. To this end, cUrl is used with the option `-w "\n%{time_total}"` to measure response time. Two queries are issued this way:

1. all walls with their openings and building elements that fill those openings
2. all building elements that fill openings in a specific wall

To ensure comparability, we started tests with a fresh BIMserver install, a new home directory as well as new project and check-in. Similarly, a fresh neo4J database was setup with new project imported. We used the query without inverses.

3.4.2 *BIMserver results*

The import took 161.965 seconds as measured on the command line. Of those, BIMserver reported it needed 292ms for writing and 2min 36s for geometry processing (triangulation). The database size after import was 61.3 MB.

The results of querying are listed in Table 1. We repeatedly issued the queries four times, clearing the cache after the second run. For both queries, there are three values listed: the time for initiating the query, the time for downloading the result, and the total. Query initiation and result download are separate calls to BIMserver to allow for clients to issue non-blocking (asynchronous) requests for long-running queries.

3.4.3 *neo4J results*

The first attempt to import the IFC data into neo4J using the Cypher query exported from BIMserver used 6G of memory with a tendency to still

Table 1: Query profiling results for BIMserver

	all openings			specific opening		
run	init	stream	total	init	stream	total
1	0.047	0.042	0.087	0.011	0.029	0.040
2	0.009	0.024	0.033	0.008	0.009	0.017
3	0.022	0.041	0.063	0.023	0.026	0.049
4	0.020	0.047	0.067	0.023	0.022	0.045

Table 2: Query profiling results for neo4J

	all openings	specific opening
first run	0.158	0.035
repeated	0.067	0.010

claim more memory and did not finish in reasonable time. The import was issued as on big `create` statement, thus it was not possible to control progress. Following that the Cypher code was improved to use less memory due to separated statements. This might use more time, especially for the creation of edges, since start and end node have to be looked up from the database. Further improvements might be possible by joining the edge creation statements per entity. In total, the import took roughly 17 hours: 11am to 4am the next morning. The resulting DB size was 215.42 MiB.

The profiling results for the queries are listed in Table 2.

3.4.4 *Comparison*

BIMserver and Cypher queries perform equally well for small files, while for large data sets the import into Neo4J constitutes an upstream performance issue. The resulting database size is 4 times larger for neo4J compared to BIMserver.

4 TOWARDS GRAPH DECOMPOSITION WITH GRAMMARS

We are considering graph transformations following the double pushout (DPO) approach. In this approach, a graph production p consists of a left- and a right-hand side typed template graph L and R as well as a glueing or interface graph I and two graph morphisms $l : I \to L$ and $r : I \to R$. Figure 3 shows an example production. We restrict the morphisms to injective morphisms and thus standard replacement [5]. From a production p and a match $m : L \to G$ we can then construct a graph transformation $G \Rightarrow^{m,p} H$. We are saying that the production p is applied to the graph G via the match m. For details on the construction of these transformations, refer to König, Nolte, Padberg, & Rensink (2018) and Ehrig, Ehrig, Prange, & Taentzer (2006).

Graph transformations can be concatenated, such that we obtain a transformation $G_0 \Rightarrow^* G_n$ from a sequence of direct transformations $G_i \Rightarrow G_{i+1}$. A typed graph transformation system consists of a type graph *IFC* and a set of productions P. Such a system can describe potentially infinite amounts of transformations which are obtained by selecting sequences of $p \in P$ and successively applying these productions via appropriate matches to a typed start graph S. A graph transformation system together with a start graph is also called a grammar, because it implicitly describes the set of all graphs that can be obtained by application of sequences of productions. This language L is denoted as $L = \{G \mid \exists S \Rightarrow^* G\}$.

[5] A non-injective l would result in splitting and a non-injective r in merging of graph nodes or edges.

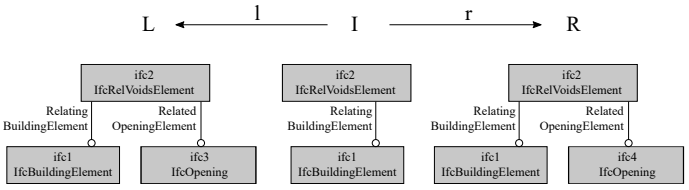

Figure 3: Transformation rule replacing an opening in a wall

We are now aiming to obtain a graph grammar from the IFC schema, that is an equivalent to the meta model in terms of graph transformations. Such a grammar could serve different use cases: First, it could foster the generation of instance graphs, e.g. for testing purposes. Second, it could aid "semantic analysis or transformation of model graphs" (Fürst, Mernik, & Mahnič 2015), that is it can provide a framework to deconstruct instance graphs in order to either better understand or transform them in a piecewise manner.

Ehrig, Ehrig, Prange, & Taentzer (2006) and Fürst, Mernik, & Mahnič (2015) describe different general approaches to convert metamodels into graph grammars. We intend to study their applicability in the context of IFC. We already identified key considerations for the IFC graph grammar: representation of mandatory and optional attributes, resources reuse, and inheritance.

For a graph grammar to be equivalent to the schema we need two conditions to be full-filled: completeness and consistency. For completeness we need all valid graphs to be contained in the language. For consistency, we need all graphs contained in the language to conform to the schema. Depending on the use cases for such a grammar, we can relax the requirement on either consistency or completeness. E.g. to be able to analyse or transform all potentially valid IFC graphs, we need completeness, but can relax consistency, assuming that we only operate on valid instance graphs. In contrast, for instance graph creation, we need consistency, but can relax completeness, accepting that we will only retain a subset of potentially valid graphs. For validation purposes we would need both completeness and consistency, because lack of consistency would result in false positives whereas lack of completeness would result in false negatives.

5 EXPLORATION WITH GRAPH VISUALIZATION

In the first part of the paper we have used graphs with various connotations: to model the IFC schema, an IFC data set, query templates, and rules. Diagrammatic representations are easy to generate from these graphs, but in order to ease the understanding of the concepts behind them, it must be clear what each graphical element is denoting. That is why we use this section to develop unique graphical representations for the concepts introduced previously. The visualization schema we are contributing is inspired by Express-G, but extended and modified to accommodate the specifics of the different types of graphs.

First, to represent the schema in isolation, we are using a type graph. Express-G is a dedicated diagrammatic notation for Express schemas and also used in the IFC documentation. It essentially provides a direct representation of the type graph: entity type nodes are represented as rectangular boxes with the name of the type name written inside.

To represent the instance graph, we use a similar representation, but each node and edge is labeled with an object (and optionally relation identifier) instead of the type, e.g. numbered consecutively. We can represent the object and type graph explicitly and indicate the typing morphism with dotted lines as shown in Figure 1. A more compact notation integrates the object and type graph representations by adding type names to the node and edge identifiers as shown in Figure 4.

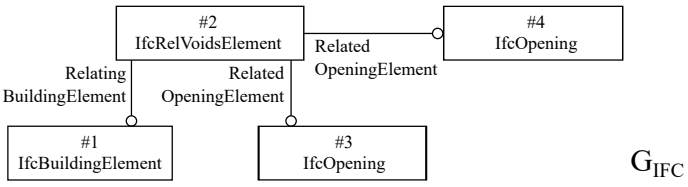

Figure 4: Integrated representation of type and instance graph (explicit version in Figure 1)

To represent a template graph we use the same diagrammatic representation as for integrated object and type graphs, but add a filling to nodes and use a duplicate line for edges to distinguish them from ordinary type graphs. We can then represent the match morphism (between a template and an object graph) similar to the typing morphism with dotted lines as in Figure 2 or again in an integrated form where the match is hight-lighted. While node IDs are following the IFC line numbers for object graphs, pure template graphs have freely chosen node IDs.

Transformation rules can be represented either as left-hand and right-hand side template graphs together with a gluing graph or a mapping (Taentzer 2001), see Figure 3. As Lambers (2005) points out, most graph transformation tools use an integrated, more compact notation, where the common parts of left- and right-hand graph templates and potentially the gluing graph are not repeated. Instead, deleted, retained and added graph parts receive a specific annotation or formatting. This representation is however restricted to specific forms of transformations where the left- and right-hand side morphisms are homomorphic. Since our current rules are not affected from this restriction, we will be using the integrated version as shown in Figure 5.

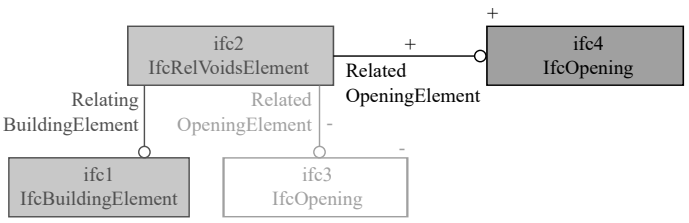

Figure 5: Integrated rule representation (explicit version in Figure 3)

We can use the very same presentation to show rule application, that is we are representing a match in the source graph and the comatch in the target graph at the same time.

6 CONCLUSION AND OUTLOOK

We have shown our approach to the application of graph-based methods to IFC data. In future work, we will apply these methods to specific use cases, such as model-checking and requirements specification using subgraph templates, in-place transformation of IFC graphs, IFC schema conversion, or transformation from IFC to other formats. For the latter we will extend the methods described here to the triad of source, target and connection graph.

The methods still need further refinement, since we excluded some detailed investigation in favour of a comprehensive overview of the fundamentals and the different types of graphs that are to be considered in the context of graph-based model transformations. For instance we excluded data nodes or labels. When we consider them in a next step, we will arrive at the category of *attributed type graphs with inheritance (ATGI)* according to Ehrig, Ehrig, Prange, & Taentzer (2006). Also for the subgraph templates we just used a fairly simple query. More complex queries are work in progress. We only touched the subject of graph grammars, but we anticipate that they play an essential role in graph-based model analysis and transformation. Therefor it would be interesting to evaluate generic graph transformation tools against a custom IFC-specific implementation.

We acknowledge the funding support from the National Research Foundation (Singapore)

under the Virtual Singapore programme grant NRF2015VSG-AA3DCM001-008. We thank Rudi Stouffs and Patrick Janssen for their comments on an early version of this manuscript.

REFERENCES

Bardohl, R., H. Ehrig, J. de Lara, & G. Taentzer (2004). Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In M. Wermelinger and T. Margaria-Steffen (Eds.), *Proc. of Fundamental Approaches to Software Engineering (Fase)*, Berlin, Heidelberg, pp. 214–228. Springer.

Ehrig, H., K. Ehrig, U. Prange, & G. Taentzer (2006). *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Berlin: Springer.

Fürst, L., M. Mernik, & V. Mahnič (2015, July). Converting metamodels to graph grammars: doing without advanced graph grammar features. *Software & Systems Modeling 14*(3), 1297–1317.

Isaac, S., F. Sadeghpour, & R. Navon (2013, August). Analyzing building information using graph theory. In *Proc. of the 30th International Symposium on Automation and Robotics in Construction (ISARC)*, Montreal, Canada, pp. 1013–1020.

Isikdag, U. & S. Zlatanova (2009). Towards defining a framework for automatic generation of buildings in CityGML using building information models. In J. Lee and S. Zlatanova (Eds.), *3D Geo-Information Sciences*, pp. 79–96. Berlin, Heidelberg: Springer.

Ismail, A., A. Nahar, & R. Scherer (2017, July). Application of graph databases and graph theory concepts for advanced analysing of BIM models based on IFC standard. In C. Koch, W. Tizani, and J. Ninic (Eds.), *EG-ICE 2017: 24th International Workshop on Intelligent Computing in Engineering*, Nottingham, UK, pp. 146–157.

ISO (2004). Industrial automation systems and integration. product data representation and exchange. part 11: Description methods: The EXPRESS language reference manual. Technical Report 10303-11, International Organization for Standardization, Geneva, Switzerland.

ISO (2013). Industry foundation classes (IFC) for data sharing in the construction and facility management industries. Technical Report 16739, International Organization for Standardization, Geneva, Switzerland.

Khalili, A. & D. K. H. Chua (2015). IFC-based graph data model for topological queries on building elements. *Journal of Computing in Civil Engineering 29*(3), 04014046.

König, B., D. Nolte, J. Padberg, & A. Rensink (2018). *A Tutorial on Graph Transformation*, pp. 83–104. Cham: Springer International Publishing.

Lambers, L. (2005). A new version of GTXL : An exchange format for graph transformation systems. *Electronic Notes in Theoretical Computer Science 127*(1), 51–63. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004).

Pauwels, P., W. Terkaj, T. Krijnen, & J. Beetz (2015, June). Coping with lists in the ifcOWL ontology. In *Proceedings of the 22nd International Workshop on Intelligent Computing in Engineering (EG-ICE)*, Eindhoven, The Netherlands, pp. 111–120.

Taentzer, G. (2001). Towards common exchange formats for graphs and graph transformation systems. *Electronic Notes in Theoretical Computer Science 44*(4), 28–40. Uniform Approaches to Graphical Process Specification Techniques (UNIGRA 2001, a Satellite Event of ETAPS 2001).

Tauscher, E., H.-J. Bargstädt, & K. Smarsly (2016, July). Generic BIM queries based on the IFC object model using graph theory. In *Proc. of the 16th International Conference on Computing in Civil and Building Engineering (ICCCBE)*, Osaka, Japan, pp. 905–912.

Vilgertshofer, S. & A. Borrmann (2017). Using graph rewriting methods for the semi-automatic generation of parametric infrastructure models. *Advanced Engineering Informatics 33*, 502 – 515.

Winter, A. (2002). Exchanging graphs with GXL. In P. Mutzel, M. Jünger, and S. Leipert (Eds.), *Graph Drawing*, Berlin, Heidelberg, pp. 485–500. Springer.