ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume IV-4/W8, 2019
14th 3D GeoInfo Conference 2019, 24–27 September 2019, Singapore

CREATING AND MAINTAINING IFC–CITYGML CONVERSION RULES

Helga Tauscher[1] *

[1] Dresden University of Applied Sciences, Dresden, Germany – helga.tauscher@htw-dresden.de

KEY WORDS: BIM, GIS, IFC, CityGML, triple graph grammar, graph transformation, domain-specific language

ABSTRACT:

We employ a triple graph grammar to enable configurable conversion from IFC to CityGML. In this paper, we present the mathematical framework behind the graph transformation approach as well as an application to create, store and maintain transformation rules implementing this framework. Particular emphasis is put on how the approach enables graphical representation and static analysis of rules and rulesets, both in the theoretical framework and prototypical implementation. Even if various publications and tools for general graph transformation do already exist, we hope that the BIM–GIS community will benefit from a domain-specific introduction to the theory and dedicated software tools.

1. INTRODUCTION

1.1 IFC2CityGML context

This work evolved as part of the IFC2CityGML project, where we employ graph transformation methods to achieve cross-domain conversion of digital building models into digital city models. We focus on two well-known data standards in the respective domains: the ISO (2013) standard Industry Foundation classes (IFC) on the source side and the OGC (2012) standard CityGML on the target side. The project aims for a complete and correct conversion, where completeness and correctness are defined through the requirements of the target specification derived from use cases in the geospatial domain.

These objectives lead to the following implications: First, a use case driven approach requires an adaptable framework with separation of general conversion logic and use-case specific logic. Second, cooperation with stakeholders to develop use-case specific details of the conversion can be facilitated with representations of the conversion logic that are more tangible than code in a general-purpose programming language. Third, completeness and correctness considerations require a method that is backed by a rigorous formal framework.

1.2 Triple graph transformation

We employ graph transformations for the following reasons:

Visual representation. Graphs have an inherent visual representation that makes complex situations easier to understand.

Runtime configuration. With a rule-based approach, the use-case specific details of the conversion procedure can be specified at runtime as opposed to approaches where the whole conversion is compiled and its scope is fixed at runtime. This also allows incremental development where the conversion process can be adapted to evolving use-case requirements.

Declarative specification. A declarative specification allows domain experts to work on the details of the conversion, independent from developers implementing the general conversion logic with an imperative approach.

Formal representation. The sound mathematical theory behind algebraic graph transformation allows for formal assessment of completeness and correctness.

Different operationalizations. Triple graph grammars allow for different operationalizations, e.g. the same grammar can be used to derive forward, backward and synchronization rules.

1.3 Related work

Integration of building information modelling with the geospatial domain has attracted considerable attention recently, because it carries the promise to scale up the advantages of each domain beyond their respective scopes and tackle cross-domain use cases. One way to approach the issue is to convert data between the most popular standards of the BIM and GIS domain — IFC ISO (2013) and CityGML OGC (2012) respectively. However, none of the existing attempts considers use-case specific configurations or employs a rigid formal approach to mapping.

Graph transformation (or graph grammar) is a well-known approach applicable to a range of problems in the software engineering domain where the problems can be represented as graph structures, see e.g. König et al. (2018) for an introduction. Ehrig et al. (2006) have worked out a seminal theory of algebraic graph transformation and also applied it to model transformation and integration problems Ehrig et al. (2015). The use of three graphs to represent source, target and connection data as well as the left- and right hand sides of transformation rules was already proposed by Schürr in 1995.

In preliminary work we have demonstrated the application of the approach for IFC-to-CityGML conversion Stouffs et al. (2018) and discussed details of graph representations on the IFC side Tauscher and Crawford (2018). Here, we present the theoretical background and its implementation in a transformation rule repository application in more detail. The reminder of this work is organized as follows: We first give an introduction to TGG as applied in our project (Section 2). We then introduce a software application to maintain these rules and describe how the theoretical concepts reflect in the implementation (Section 3). In Section 4 we show how the application has been employed for IFC-to-CityGML conversion. Finally we discuss limitations and future work.

* Corresponding author

2. FORMAL MODEL

We will first give a slightly simplified introduction to triple graph grammars using the triple of connected IFC and CityGML graphs as example case. For the conversion we restrict ourselves to forward transformations. We will show the formal specifics of forward transformation rules and elaborate on how static analysis can be carried out over these rules.

2.1 TGG primer using the IFC2CityGML example

In triple graph grammars the two sides of an integration or transformation as well as their connections are represented as three distinct graphs. For unidirectional cases such as ours we can identify a source and a target side and call the three graphs source, target and connection graph. A triple graph grammar then consists of production rules to generate all possible consistent graph triples starting from an empty triple. A grammar can also be seen as a transformation system, where productions are called transformations and the start structure is not restricted to be empty. In this section we will look at the mathematical elements of a grammar or transformation system for IFC–CityGML integration or conversion.

A directed graph G consists of a set of nodes N, a set of edges E, and two functions $s, t : E \to N$ that assign source or target nodes to edges: $G = (N, E, s, t)$. A graph morphism is a mapping between two graphs such that the graph structure is maintained. Given two graphs $G_1 = (N_1, E_1, s_1, t_1)$ and $G_2 = (N_2, E_2, s_2, t_2)$, a morphism M is defined by two functions $m_N : N_1 \to N_2$ and $m_E : E_1 \to E_2$, mapping the nodes and edges of the two graphs: $M = (m_N, m_E)$. To guarantee the structure-preserving quality of a morphism, it must hold that $s_2 \circ m_E = m_N \circ s_1$ and $t_2 \circ m_E = m_N \circ t_1$. We write short $M : G_1 \to G_2$ for such a morphism.

In our application of triple graph grammars, graph morphisms appear in different places. First of all, the source and target graphs of a triple are typed, which means that there are type graphs IFC and GML representing the schemas. For an IFC-typed graph G_{IFC} there is a morphism $type_{IFC} : G_{IFC} \to IFC$, and similarly for a GML-typed graph G_{GML} there is a morphism $type_{GML} : G_{GML} \to GML$. For every morphism $m : G_{T1} \to G_{T2}$ between two typed graphs G_{T1} and G_{T2} with the same type graph T and typing morphisms $type1 : G_{T1} \to T$ and $type2 : G_{T2} \to T$, we require $type2 \circ m = type1$. Intuitively this means, that in addition to preserving the graph structure, a morphism of typed graphs must also preserve the type of the mapped nodes and edges.

A central concept used in graph transformation approaches based on category theory is the so-called pushout. For two morphisms $b : A \to B$ and $c : A \to C$, which share a domain A, a pushout is a graph P with another two morphisms $d : B \to P$ and $e : C \to P$ which share a co-domain. The resulting diagram is said to commute, because $d \circ b = e \circ c$. Further, for a push-out the graph P must be the most general or universal graph that satisfies the commutative condition. It has been shown that this graph is unique up to isomorphism. For example, in Figure 1, the four graphs and morphisms $l : K \to L$, $d : K \to D, m : L \to G, g : D \to G$ form such a commutative diagram with shared domain K and co-domain G. If G is the most general graph fulfilling the commutation, then it is said to be the pushout of L and D over K.

We are adopting double-pushout (DPO) graph-rewriting, where each production rule is represented as a span $(L \leftarrow K \to R)$ with three graphs L (left hand side of the rule), R (right-hand side), K (invariant or glue graph) and two morphisms $l : K \to L$ and $r : K \to R$. These rules are then successively applied to a start graph to retrieve productions. The application of a rule to a host graph G is represented by a diagram of two push-outs, a so-called DPO diagram as shown in Figure 1. The upper row of the diagram shows the rule span, whereas the lower row represents the application of the rule. After finding a match $m : L \to G$, that is an appearance of the left-hand side L in the host graph G, a context graph D is constructed to complete the left push-out diagram. Successively, the production H can be constructed to complete the right push-out diagram, with the co-match $m^* : R \to H$. Figuratively speaking, the part of the host graph G that matches the left-hand side of the rule L is replaced with the right-hand side R to derive H.

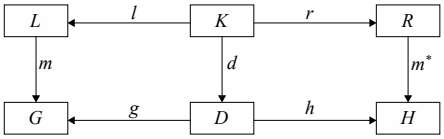


Figure 1. Double pushout

Now, a double push-out for triple graphs encompasses not only a single graph, but a triple of two typed graphs and one untyped graph. Thus each box in the original pushout diagram will contain a triple of three graphs and each arrow will represent three morphisms, where two of the graphs and two of the morphism are typed. For instance the triple graph L consists of the three graphs L_{IFC}, L_{CON}, and L_{GML} with typing morphisms $ifc_L : L_{IFC} \to IFC$ and $gml_L : L_{GML} \to GML$. As an example for a triple morphism $m : L \to G$ consists of the three morphisms

$$m_{IFC} : L_{IFC} \to G_{IFC}, \tag{1}$$
$$m_{CON} : L_{CON} \to G_{CON}, \tag{2}$$
$$m_{GML} : L_{GML} \to G_{GML}, \tag{3}$$

where m_{IFC} and m_{GML} are typed and those we require $ifc_G \circ m_{IFC} = gml_L$ and $ifc_G \circ m_{GML} = gml_L$. The resulting DPO diagram for IFC–CityGML conversion is shown in Figure 2. It contains the following graphs and correlating morphisms:

- Triple L (left-hand side): L_{IFC}, L_{CON}, L_{GML}
- Triple K (invariant/glue): K_{IFC}, K_{CON}, K_{GML}
- Triple R (right-hand side): R_{IFC}, R_{CON}, R_{GML}
- Triple G (input): G_{IFC}, G_{CON}, G_{GML}
- Triple D (glue): D_{IFC}, D_{CON}, D_{GML}
- Triple H (output): H_{IFC}, H_{CON}, H_{GML}
- Type graphs IFC and GML
- Typing morphisms IFC: $ifc_L : L_{IFC} \to IFC$, similarly ifc_K, ifc_R, ifc_G, ifc_D, ifc_H
- Typing morphisms GML: $gml_L : L_{GML} \to GML$, similarly gml_K, gml_R, gml_G, gml_D, gml_H
- Rule span morphisms: $l_{IFC} : K_{IFC} \to L_{IFC}$ and $r_{IFC} : K_{IFC} \to R_{IFC}$ similarly $l_{GML}, r_{GML}, l_{CON}, r_{CON}$

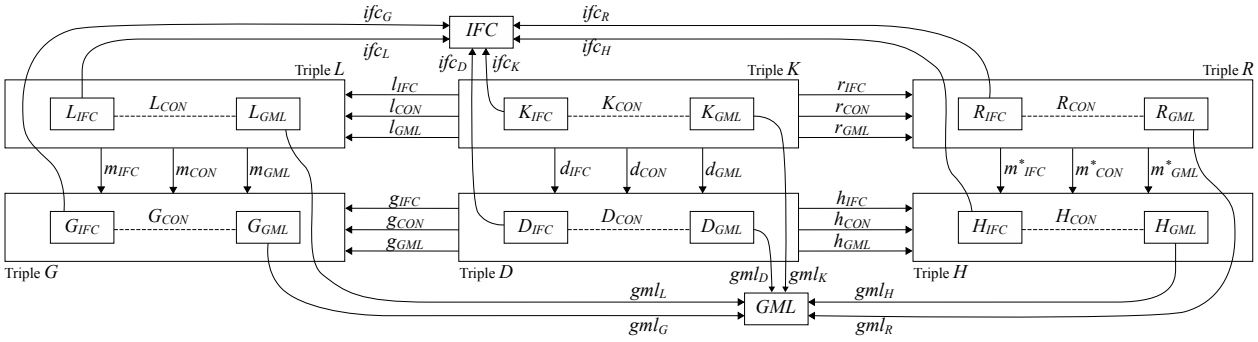


Figure 2. Double pushout for a triple graph rule with all participating graphs and morphisms

- Host span morphisms:
  $g_{IFC} : K_{IFC} \to G_{IFC}$ and $h_{IFC} : K_{IFC} \to H_{IFC}$
  similarly $g_{GML}, h_{GML}, g_{CON}, h_{CON}$
- Match and co-match morphisms:
  $m_{IFC} : L_{IFC} \to G_{IFC}$ and $m^*_{IFC} : R_{IFC} \to H_{IFC}$
  similarly $m_{GML}, m^*_{GML}, m_{CON}, m^*_{CON}$
- Glue morphism: $d_{IFC} : K_{IFC} \to D_{IFC}$,
  similarly d_{GML} and d_{CON}

It can be seen that the 6 triples result in 18 graphs of which 6 are IFC-typed, 6 are GML-typed and 6 are untyped.

2.2 Integrated representation for visualisation

To cater for the visualization (Section 3.1) and for later operationalization in forward transformation (Section 4) we are storing the rule spans ($L \leftarrow K \rightarrow R$) in an integrated way, where nodes and edges appearing in all three graphs of the span are stored only once. Instead of repeating the invariant graph we have thus nodes and edges which appear only in the left hand side marked explicitly as deleted elements and those only in the right hand side marked as added elements.

An example is shown in Figures 3 and 4. The explicit representation in Figure 3 corresponds to the upper row in Figure 2. The morphisms are not explicitly depicted, but can be understood from corresponding node identifiers, e.g. "ifcBuilding", "gmlBuilding" etc. Figure 4 shows the integrated representation of the three triples in Figure 3. More detail on visual rule representation is given in Section 3.1.

This is possible as long as the rule span morphisms are isomorphic, that is the node and edge mapping functions are bijective. If this is the case, we can drop the morphisms l and r and remove the isomorphic images of the invariant/glue graph K from the left- and right-hand graphs L and R. Given the three graphs $L = (N_L, E_L, s_L, t_L)$, $K = (N_K, E_K, s_K, t_K)$, $R = (N_R, E_R, s_R, t_R)$ and isomorphisms $l : K \to L$, $r : K \to R$, we will end up with one graph $I = (N_I, E_I, s_I, t_I)$ and subsets $N_D \subset N_I$, $E_D \subset E_I$ of deleted nodes and edges, $N_A \subset N_I$, $E_A \subset E_I$ of added nodes and edges, and $N_P \subset N_I$, $E_P \subset E_I$ of preserved nodes and edges.

We omit the details of converting the original rule-span into the integrated representation and vice versa and just characterize the original and the integrated representation of the rule span:

$$N_D, N_P, N_A \text{ are pairwise disjunct} \quad (4)$$
$$E_D, E_P, E_A \text{ are pairwise disjunct} \quad (5)$$
$$N_K = N_P \text{ and } E_K = E_P \quad (6)$$
$$N_L = N_D \cup N_P \text{ and } E_L = E_D \cup E_P \quad (7)$$
$$N_L = N_A \cup N_P \text{ and } E_L = E_A \cup E_P \quad (8)$$

2.3 Forward transformation

One advantage of triple graph grammars is that they allow for the derivation of different operational transformation systems. However, in this project we focus on forward transformation and thus chose to directly specify the transformation rules instead of deriving them. This choice eliminates potential complexity at the cost of reduced generalization and flexible utilization Stouffs et al. (2018).

This decision implies some restrictions on the transformation rules. In particular, no changes are allowed on the source graph. The start (triple) graph of a forward transformation consists of the input IFC data as graph G_{IFC} together with empty CityGML and connection graphs G_{GML} and G_{CON}. The input data will not be modified, while the CityGML and connection graph are populated during the conversion process, hence the final production $H_{IFC,n}$ after n rule applications, should be equal to the initial start graph G_{IFC}. This can be confirmed for two cases: 1. G_{IFC} and H_{IFC} are isomorphic for every rule application. This is the case if and only if $L_{IFC}, D_{IFC}, R_{IFC}$ are isomorphic too for every rule. 2. Every added intermediate node or edge is removed in later rule application. We allow the latter, but only for the elements of enhancement types (e.g. context), not for nodes and edges of IFC types.

As an example, the rule in Figure 4 demonstrates the limitations: newly created nodes and edges appear only on the GML side, not on the IFC side and deletions are restricted to the context node. Technically these restrictions simplify the model, since we can remove or omit a few graphs and morphisms from the model. Another such simplification on the CityGML side is the prohibition of removals and limitation to additions. Apart from the restriction to forward transformations we require a few further restrictions specific to our conversion procedure which are described later in the implementation Sections 3.4 and 3.5.

2.4 Static analysis

With this formal model in place, we can carry out some static analysis on the ruleset, without knowing about particular input data and without actually executing the conversion.

Check forward transformation restrictions. We can check whether the rules follow the restrictions posed for forward

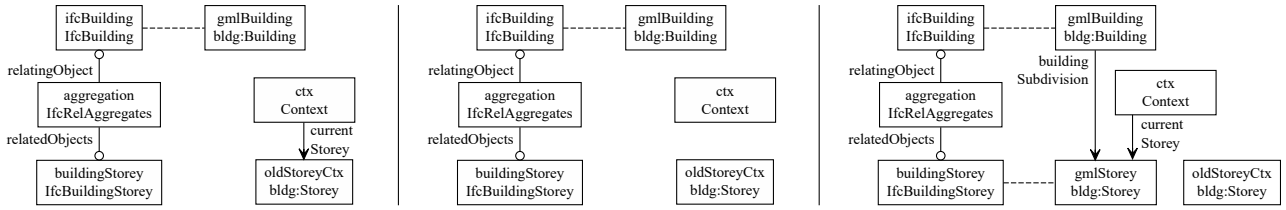


Figure 3. Forward transformation rule, explicit representation with left hand side (left), glue graphs (centre) and right hand side (right) of a triple graph transformation rule

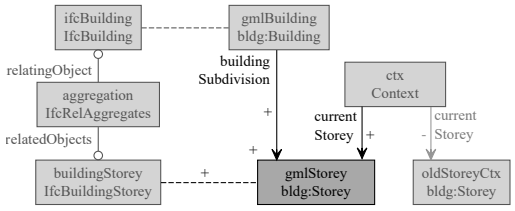


Figure 4. Forward transformation rule, integrated representation

transformation in the previous Section 2.3. Further restrictions can ensure rule structures specific to conversion algorithms.

Schema conformance. We can check whether the typing morphisms do indeed exist. This ensures that our rules conform to the source and target metamodel. This aims at verifying conversion correctness, where all conversions process and create only valid IFC and CityGML models.

Schema coverage. We can check how many of the nodes and edges in the source and target type graphs are covered by the co-domain of the rules' typing morphisms. This will eventually verify conversion completeness, that is to which degree we can process and create all possible valid IFC and CityGML models.

Identify rule dependencies and conflicts. Even without actual input data, we can make some statements about potential rule application. For instance, critical pair analysis allows to identify dependent and conflicting rules. This is useful, for instance, to select rules for parallel application. Further, we can determine rule application order for our conversion algorithm, identify rules that will potentially never apply and asses termination of the transformation.

It must be noted that with schema conformance and coverage analysis, we can only assess partial aspects correctness and completeness. For a thorough analysis, we would need to deduce grammars from the source and target metamodel and from the rule's isolated the source and target template parts. We could then try to compare the metamodel and ruleset deduced grammars. Unfortunately it is not easy to decide whether one language or grammar is equal or a subset of another.

3. RULE REPOSITORY

We have created a web application for editing and management of triple graph transformation rulesets. The application is implemented with Grails, a web application framework for the JVM written in Groovy Smith and Ledbrook (2014). Part of the functionality can also be used as a library. Grails contains GORM, which provides object relational mapping (ORM) and allows for transparent integration of different relational databases. Throughout our project we used the default H2 database in embedded mode, which can be swapped easily.

We take advantage of the intuitive visualization of graph transformation rules (Section 3.1) and provide a domain-specific language to specify the rules (Section 3.2). The rules are fully flexible to be specified for any operationalization or as general rules, but we provide validations with regard to the limitations of forward transformation as required in our project (3.3). Further the tool includes some simplified rule notations (Section 3.5) and functionality to group and manage the rules (Section 3.6).

3.1 Visualization

Section 2.2 introduced the underlying mathematical model for visualizations and a first example diagram (Figure 4). Here we add more detail about the design and creation of these diagrams.

Each node is represented as a box labelled with its identifier and type (the name of its image node in the type graph). Each edge is represented as a continuous line connecting two nodes, annotated with its type and a line end marker indicating its direction: either a dot for the IFC graph inspired by Express-G ISO (2004) or an arrow for the CityGML graph inspired by UML. The type annotations of nodes and edges capture the typing morphisms, that is the mapping into the type graph which is not explicitly represented. Connection graph rules are undirected, untyped and represented as dashed lines.

Nodes are filled in grey, with a darker tone for created nodes and edges and a lighter tone for deleted nodes and edges. In addition, plus and minus symbols are used to annotate created and deleted elements. We forego colours in favour of an accessible design and to support greyscale printing.

The visualizations are generated in GraphViz DOT format Gansner and North (2000) and then rendered to Scalable Vector Graphics (SVG) using the Viz.js libary[1]. The SVG can be displayed natively in modern browsers. The application also offers a download link for further processing of the visual representations. GraphViz offers various graph layout algorithms. We have chosen an orthogonal layout for directed graphs, to tie in with familiar EXPRESS-G and UML layouts.

3.2 Domain specific language

We developed a domain specific language (DSL) to specify the graph transformation rules. As an introductory example, consider the listing below. It describes the rule from Figure 4, without the context nodes and edges.

The rule description consists of three main blocks, one for the glue graph (preserved), one for the graph elements only contained in the left-hand graph (deleted), and one for the graph elements only contained in the right-hand graph (added). Empty

[1] https://github.com/mdaines/viz.js

blocks can be omitted. Each of these blocks contains three blocks pertaining to the three graphs of a triple and each graph contains a node as well as an edge block. Nodes are grouped by type and listed by identifier. Edges are grouped by type and listed as pairs of source and target node identifiers.

```
ifc2citygml.rule {
  preserved {
    ifc {
      nodes {
        IfcBuilding { ifcBuilding() }
        IfcRelAggregates { aggregation() }
        IfcBuildingStorey { ifcStorey() }
      }
      edges {
        relatedObjects = [[aggregation, ifcStorey]]
        relatingObject_ = [[aggregation, ifcBuilding]]
      }
    }
    gml {
      nodes {
        'bldg:Building' { gmlBuilding() }
      }
    }
    con {
      edges {
        _ = [[ifcBuilding, gmlBuilding]]
      }
    }
  }
  created {
    gml {
      nodes {
        'bldg:Storey' { gmlStorey() }
      }
      edges {
        buildingSubdivision = [[gmlBuilding, gmlStorey]]
      }
    }
    con {
      edges {
        _ = [[ifcStorey, gmlStorey]]
      }
    }
  }
}
```

Groovy supports the implementation of DSLs with various language constructions. Our implementation makes use of Groovy's builder support Dearle (2010). The syntactic elements were chosen such that the implementation is easy to accomplish, the language is readable, writing is efficient and the resulting code is compact. For instance, we deliberately decided against XML, with verbose start and end tags, and against JSON, with mandatory quotation marks for keywords.

By using the node identifiers and Groovy expressions, we can define predicates that act as constraints for the rule application (validators) and also functions to be executed during rule application to populate attribute values (converters).

3.3 Validation of rules and rulesets

The graph visualization of a rule provides a way to assess entered DSL code for plausibility and conformance to the intentions of the author. It helps to spot errors such as disconnected nodes, wrong edge directions, wrong start or end nodes of edges. But in order to produce the visualization the DSL code needs already to be valid on a basic level, e.g. the general syntax must be valid, there should be no dangling edges. After visual plausibility checking passes, there are some severe issues that are harder to identify by visual inspection, e.g. typos in IFC or CityGML type names, node-edge connections that do not conform to the schemas, edges to be created that connect nodes to be deleted, violations of the forward transformation restriction. To catch these two categories of errors (before and beyond visual plausibility checking) and to provide appropriate feedback to rule authors, the application carries out some automated validation upon editing of a rule. In particular, we check for syntactic correctness, semantic correctness (schema conformance), and further semantics, e.g. the specific rule structure required for the conversion algorithms.

Syntactic correctness. Since the DSL is an embedded DSL which makes use of host language constructions, a large part of the syntactic errors are covered by the Groovy parser, e.g. non-matching parenthesis, expressions with wrong structure, assignments in inadequate places etc. These errors are reported from the parser with DSL code line number and can be passed to the UI as they are. Some errors, which are syntactically correct in Groovy, but not in the DSL, can only be covered later during construction of the model from the DSL script, e.g. missing angular brackets where a list is expected. For these errors, separate error messages are generated in the DSL builder.

Schema conformance. This validation ensures a first level of semantic correctness. We want to guarantee that the type morphisms (Section 2.1) do exist. For IFC, the type graph is represented as object graph of the buildingSMART library[2] and we programmatically check the morphism's existence and preservation of graph structure. Mismatches such as non-existent node types or inappropriate combinations of edge and node types are reported. A similar check could be carried out on the CityGML side using the XML Metadata Interchange (XMI) representation of the schema, though in the prototype we have only implemented the IFC schema validation.

Further semantics of single rules. Additional validation ensures specific structures of transformation rules. For instance, we can limit rules to forward transformations (Section 2.3) or enforce limitations presented by the rule application (conversion) procedures. In our case, the conversion engine requires the rules to conform to one of a given set of rule types. We describe these rule types in Section 3.4.

Static analysis of rulesets. Apart from single rules, we can evaluate the consistency of the whole ruleset. Instead of searching an instance graph for matches of the rule's left-hand side (graph parsing), we use our confined rule types with their well-defined pre- and post-conditions to trace their potential application order. This way it is possible to identify isolated rules that will never apply or ambiguities and infer missing rules or refining conditions. Other static analysis that can be carried out is the evaluation of schema coverage on source and target side or termination qualities of the rule set by identifying potential circular rule application sequences.

In our prototype, the following are implemented: validation of syntactic correctness, schema conformance on the IFC side, forward transformation and rule type evaluation. It should be noted that these validations are unrelated to the validators mentioned in Section 3.2. While the former validate rules statically independent of their application to a specific data set (or start graph), the latter are invoked during graph transformation to guide rule application for a given data set.

[2] https://github.com/opensourceBIM/BuildingSMARTLibrary

3.4 Rule types

The application can recognize and mark rules as being of a specific type. We identify four different rule types that are relevant to our conversion methods. These are shown in Figure 5. Anjorin et al. (2015) present guidelines for the development of graph transformation rules and identify very few generic patterns that appear in the rule design process: islands, extensions, and bridges. These correspond roughly to our root, standard, and reference types.

The rule types differ and are distinguishable by the number of connected pairs of an IFC and CityGML node, and the graph elements created on CityGML side (either nodes, edges or none). For instance, the standard rule (Figure 5b), has two connected pairs of IFC and CityGML nodes, with a path of edges and nodes between the two CityGML nodes. We call these two IFC–CityGML pairs entry and exit pair. The CityGML exit node the CityGML path elements appear only in the right hand side of the rule (created nodes and edges).

It is important to note that the rules as depicted in Figure 5 are only prototypical rules. They can contain arbitrarily complex left-hand side constructions on the IFC as well as the CityGML side to define their application scope. The connecting paths between the IFC nodes and the CityGML nodes of the pairs can be arbitrarily complex as well, where the prototypical rules only contain a simple edge, for instance the *gmlRelation* edge between *gmlParent* and *gmlChild*.

The particulars of each rule type can be defined mathematically in the formal framework laid out in Section 2, but we are omitting the details for the limited scope of this paper.

3.5 Rule definition shortcuts and expansion

To facilitate rule development, we introduced shortcut notations for some common rule constructions.

As mentioned in Section 3.2, the most flexible way to notate validators is to describe them as a predicate expression over identifiable elements of the transformation rule. However, rules are easier to read and understand if the validators appear near the elements that they relate to. Thus we allow validators that constrain an attribute value to be defined directly at the node, using notation similar to UML (Figure 6a). This is implemented for literal String attribute values and lists of possible values.

We use a similar notation to specify getters and setters for attributes as shortcuts (Figures 6b and 6c). These shortcuts are specific to a rule construction that we use in our conversion procedure: a global context to hold values across rule applications. The context can be seen as a dedicated node to which we attach and from where we access global values. This is only possible because we generate a tree-like structure on the CityGML side. Because we can follow an application order of rules that walks the tree top-down, we can assume that rules further up in the tree structure are already processed and context is populated with the respective values. Figures 6b and 6c show how this is expressed as a graph rule and how the shortcut simplifies the rule graph.

Getters can exist on CityGML and IFC side, while setters should only appear on the CityGML side or on the IFC context. There is also a shortcut notation to put whole nodes into context and notations to specify attribute value converters.

Nodes created with a standard rule (Figure 5b) are implicitly ”put” into context with their class name as index. That means any edge from context to an existing node of the same class is deleted and a new edge is added from context to the node just created. In Figure 4, the nodes ”ctx” and ”oldStoreyCtx” and the two ”current” relations can be omitted in the shortcut form.

We provide another rule type, which combines a standard rule (Figure 5b) with a reference rule (Figure 5c). The resulting standard-reference rule type is shown in Figure 7. Rules of this type are required by the IFC–CityGML conversion algorithm. They allow for the generation of links in the tree structure across multiple branches of the tree. In our implementation they are to be specified directly in the combined form, but they could also be generated automatically through static analysis from combining a standard and a reference rule with corresponding types of the CityGML node. Note that the standard-reference rule again permits to omit the explicit context node declaration. The dangling CityGML node which is not connected to an IFC counterpart is to be taken from the context.

3.6 Management, modularity, grouping, packaging

The web application also offers some supporting functionality for the management of the rules. Rules have identifying names and numeric IDs, they can be viewed in lists with paging and sorting, and they can also be deleted.

Rules can be grouped into rulesets, with each rule being assigned to one or more identifiable and named sets. All rules in a ruleset can be downloaded at once as a zip archive and conversely such an archive can be imported to create a new ruleset in the database. The database can also be queried for orphaned rules, which are not assigned to any ruleset.

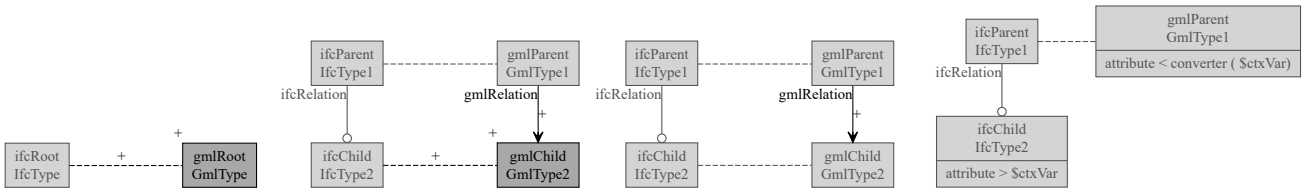
This basic functionality does already allow for the creation of modular rule repositories to explore different conversion options, for instance regarding the spatio-semantic paradigms.

4. EMPLOYING THE RULES

The rules and rulesets in the rule repository can be employed in a variety of ways, from static analysis, to population of graph transformation systems for actual conversion, to deduction of IFC or CityGML side model requirements (e.g. in MVD form for IFC), or checking of IFC input data against a given ruleset. For the IFC2CityGML project we have implemented both input checking for rule applicability and actual conversion through a custom application. Both are implemented as separate applications that receive an IFC file as input and in addition configuration information in JSON format deduced from a given ruleset. This section describes how the JSON payload for these applications is generated from the rulesets.

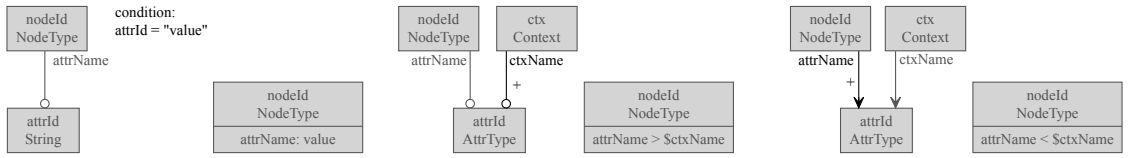
4.1 Pre-processing for checking

The checking application produces a report in table form, with records for each rule stating how often this rule would apply for a given IFC file. In order to achieve this, it processes a tree spanning the IFC part of the rule (which is part of the left-hand side by definition of the forward transformation) to match the rule against the IFC object graph starting at potential IFC entry nodes and computing the match count as the rule tree is traversed in parallel with the IFC input graph. The rule repository application extracts this IFC spanning tree for every rule in a



(a) Root rule (b) Standard rule (c) Reference rule (d) Property rule

Figure 5. Rule types as deduced from the requirements of the conversion procedure



(a) Validators, long and short form (b) Getters, long and short form (c) Setters, long and short form

Figure 6. Shortcuts to define attribute-specific validators, setters and getters, long forms (left) and short forms (right)

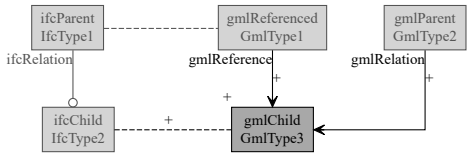


Figure 7. Combined rule type for conversion

ruleset and exposes it in JSON form for the checking application to be used.

We used the results of this checking during the rule-development process to shortcut the feedback loop and replace validation and analysis of actual conversion output. The checking results allow to assess where rules are missing or have preconditions which are too broad or too narrow. As opposed to a full conversion run, each rule is evaluated in isolation and thus context can not be considered. Because of this it must be noted that this can not be reasonably applied for standard-reference rules. Checking ignores the non-standard part of the rule, because context is not populated. For these rules to get more complete checks, we would have to split them into standard and reference rules and thus incorporate the context information into the reference rule.

4.2 Pre-processing for conversion

The conversion application actually carries out the actual conversion from IFC to CityGML. The details of the conversion process are beyond the scope of this paper, but the requirements of the conversion process were developed in parallel to the rule management application described here and led to the rule types described in Section 3.4 as well as some of the shortcuts in Section 3.5. The conversion also takes advantage of the simplifications described in Section 2.3. This is why the JSON payload for the conversion can be generated pretty straight-forward.

To generate these operational conversion rules we extract the connected IFC and CityGML nodes (entry and exit pairs), the paths between connected IFC nodes and connected CityGML nodes (if the rule has two pairs) and potentially to the referenced CityGML node. Further we collect the setters, getters, and validators and resolve the path from a defined node (usually the IFC or CityGML exit node).

5. CONCLUSIONS

5.1 Summary

We have shown a formal mathematical framework to apply graph transformation for the IFC-to-CityGML conversion and its implementation in a rule management application including a custom DSL and visual representation of the transformation rules. We have also demonstrated how the general graph transformation approach was tailored to our conversion algorithm.

It must be admitted, that rules that rely on predictable rule application order, do not conform to the original ideas of graph transformation. Instead we mix in concepts of other programming paradigms. We could express that same conversion logic in rules that only use the local rule context, but these would be more complex and rule application procedures would be complicated in terms of finding appropriate matches. We strived to find a balance between general graph transformation and manageable operationalization.

A similar objective has led to and comparable approach has been taken with layered grammars, where rules are organized in layers to be applied in a particular order, while the order among rules of the same layer stays undetermined. This way, for instance, deletion of nodes can be postponed to a stage where they are not needed as precondition for other rules.

5.2 Limitations

In favour of achieving a working end-to-end conversion process, we accepted some limitations on the theoretical side and in the implementation.

First of all, in our theoretic framework there is no proper handling for primitive data types and hence attribute mapping. This

is not relevant for our implementation, since in Java primitive types can be treated similar to non-primitive types. However, for more rigid analysis and checking in particular of the validator and converter parts of the rules, it would be necessary to add a data type algebra to the model.

The formal model is also still weak with regard to the representation of generalization and specialization in the type graph. For now, we resort to a flattened type graph where an association between a given source and target type is represented as a set of edges for every pair of nodes representing the source type or any of its supertypes and the target type or any of its supertypes. Transformations based on rules with flattened have been shown to be equivalent to such with explicitly modelled generalization **?**, but while they remove complexity from the formal model and some analysis such as type checking, other analysis may become more complicated ore performs worse, such as static application order analysis.

The integrated representation makes it easier to understand the rules, but it can not handle splitting and joining of nodes (non-isomorphic rule span morphisms). The current implementation of the shortcut validators is limited to String equality checks. We do not support negative application conditions, which would be necessary for more complex rule applications. All of these result from deliberate prioritization choices during prototype implementation.

5.3 Future work

The graph-transformation based conversion approach opens up the possibility for static analysis, of which we have implemented only a small portion and which would be a promising field for future research and development. This includes, but is not limited to assessment of conversion completeness and correctness, ruleset consistency, and schema coverage on source and target side.

It would also be interesting to investigate whether the current forward transformation rules can be transformed into general TGG rules which allow for the derivation of different operational transformation systems, e.g. model synchronization. It could be also worth trying to populate a general graph transformation system such as AGG with the rules. Connected to this, the standard-reference rule type would need reconsideration.

Some concepts omitted from the implementation, such as negative application conditions would need to be implemented to leverage the full potential of graph transformation. Other thing nice to have would be better rule editing support with syntax highlighting and autocompletion, maybe even interactive visualization and graphical editing of the rules. CityGML and ADE schema conformance of rules with XMI would facilitate rule development.

ACKNOWLEDGMENTS

This material is based on research/work supported by the National Research Foundation under Virtual Singapore Award No. NRF2015VSG-AA3DCM001-008. I would like to acknowledge the work of the whole IFC2CityGML research group and in particular thank Joie Lim and Amol Konde for discussions during the implementation of rules and conversion algorithms.

REFERENCES

Anjorin, A., Leblebici, E., Kluge, R., Schürr, A., Stevens, P., 2015. A systematic approach and guidelines to developing a triple graph grammar. A. Cunha, E. Kindler (eds.), *Proceedings of the 4th International Workshop on Bidirectional Transformations (BX2015)*. L'Aquila, Italy, 81–95.

Dearle, F., 2010. *Groovy for Domain-Specific Languages*. Packt Publishing, Birmingham.

Ehrig, H., Ehrig, K., Prange, U., Taentzer, G., 2006. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Berlin. doi:10.1007/3-540-31188-2.

Ehrig, H., Ermel, C., Golas, U., Hermann, F., 2015. *Graph and Model Transformation: General Framework and Applications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Berlin. doi:10.1007/978-3-662-47980-3.

Gansner, E.R., North, S.C., 2000. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11), 1203–1233.

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.

König, B., Nolte, D., Padberg, J., Rensink, A., 2018. A tutorial on graph transformation. R. Heckel, G. Taentzer (eds.), *Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig*, Springer International Publishing, Cham, 83–104. doi:10.1007/978-3-319-75396-6_5.

OGC, 2012. OGC City Geography Markup Language (CityGML) encoding standard. Technical Report Version 2.0.0, Open Geospatial Consortium.

Schürr, A., 1995. Specification of graph translators with triple graph grammars. E.W. Mayr, G. Schmidt, G. Tinhofer (eds.), *Proc. of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '94)*, Springer, Berlin, Heidelberg, 151–163.

Smith, G., Ledbrook, P., 2014. *Grails in Action*. New York.

Stouffs, R., Tauscher, H., Biljecki, F., 2018. Achieving complete and near-lossless conversion from IFC to CityGML. *International Journal of Geo-Information (IJGI)*, 7(9), 355. doi: 10.3390/ijgi7090355.

Tauscher, H., Crawford, J., 2018. Graph representations for querying, examination, and analysis of IFC data. *Proc. 12th European Conference on Product and Process Modelling (ECPPM)*. Copenhagen, Denmark, 421–428. doi: 10.1201/9780429506215-53.

Revised August 2019