Data transformations in custom digital workflows:
Property graphs as a data model for user-defined mappings

PATRICK JANSSEN[1], RUDI STOUFFS[1,2], ANDRE CHASZAR[2],
STEFAN BOEYKENS[3] and BIANCA TOTH[4]

[1] National University of Singapore, Singapore
patrick@janssen.name, stouffs@nus.edu.sg
[2] Delft University of Technology, Netherlands
r.m.f.stouffs@tudelft.nl, a.t.chaszar@tudelft.nl
[3] KU Leuven, Belgium
stefan.boeykens@asro.kuleuven.be
[4] Queensland University of Technology, Australia
bianca.toth@qut.edu.au

Abstract. This paper describes the use of property graphs for mapping data between AEC software tools, which are not linked by common data formats and/or other interoperability measures. The intention of introducing this in practice, education and research is to facilitate the use of diverse, non-integrated design and analysis applications by a variety of users who need to create customised digital workflows, including those who are not expert programmers. Data model types are examined by way of supporting the choice of directed, attributed, multi-relational graphs for such data transformation tasks. A brief exemplar design scenario is also presented to illustrate the concepts and methods proposed, and conclusions are drawn regarding the feasibility of this approach and directions for further research.

1. Introduction

There is a clear and urgent need for new approaches to information exchange that address the persistent lack of interoperability and integration in building design, analysis and construction. The continuing and active discourse amongst AEC practitioners and researchers alike highlights current limitations in both process and technology that commonly challenge design collaboration. We propose that bottom-up, user-controlled and process-oriented approaches to linking design and analysis tools are more appropriate than current top-down, standards-based and model-oriented strategies, because they provide degrees of flexibility critical to the process(es) of design (Coons 1963). This approach emerged out of discussions at the “Open Systems and Methods for Collaborative BEM (Building Environment Modelling)” workshop held at the CAAD Futures Conference in July 2011, and was further developed into a proposal for a platform (Toth et al, 2012). This paper does not intend to give a comprehensive overview of the proposed platform, but instead focuses on critical aspects of overcoming interoperability hurdles.

The proposed platform is based on existing scientific workflow systems that enable the composition and execution of complex task sequences on distributed computing resources (Deelman et al, 2009). These systems exhibit a common reference architecture and consist of a graphical user interface (GUI) for authoring workflows, along with a workflow engine that handles invocation of the applications required to run the solution (Curcin and Ghanem, 2008). Workflows are represented as networks of nodes and wires that can be configured and reconfigured by users as required. Nodes perform some useful function; wires support the flow of data, linking an output of one node to an input of another node. We consider three types of nodes: process nodes, input/output (IO) nodes and control nodes. Process nodes perform data analysis and transformation procedures. They have a number of typed input and output ports for receiving and transmitting data files, as well as meta-parameters that can be set by the user to guide task execution. IO nodes and control nodes are simpler types of nodes not associated with specific design task related computational procedures, but rather with providing functionality related to workflow initiation, execution and completion. In this paper we focus primarily on the role of process nodes in workflows, and the development of a strategy to support custom data transformation procedures.

Process nodes can be further classified into tool nodes and mapper nodes. Tool nodes wrap existing applications to make their functionality and data accessible to the workflow; while mapper nodes apply transformation procedures to data sets in order to map the output from one tool node to the input for another. Figure 1 shows a conceptual diagram of an example network in which a parametric CAD system node is connected via a set of mapper nodes (denoted by 'M') to EnergyPlus [1] and Radiance [2] simulation nodes. The CAD system node encapsulates a procedure that starts the CAD system, loads a specified model, and then generates a model instance by setting certain parameter values. The resulting geometric output undergoes two separate transformations that map it into both EnergyPlus and Radiance compatible formats. The simulation nodes then read in this transformed data, run their respective simulations, and generate output data in the form of simulation results.

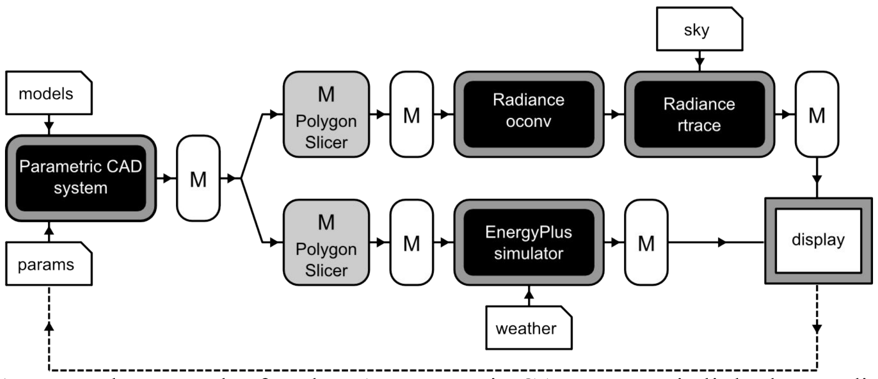


Figure 1: Example network of nodes. A parametric CAD system is linked to Radiance and EnergyPlus via a set of mappers (M). End users may contribute the white components, node developers will build the grey components, and the black components are existing tools.

Data mapping can be approached in a number of ways, ranging from an ontological approach, where a common ontological data structure is imposed on data exchanges, to an open-world approach, where the user is expected to resolve data format issues manually - a process known as 'shimming' (Altıntaş, 2011). In the AEC industry, the most prominent solution to this issue is the current evolution of Building Information Modelling (BIM), which tends toward the all-encompassing ontological end of the spectrum, with the Industry Foundation Classes (IFC) as its embodiment. However, IFC – like any standardised ontology – has significant epistemological, practical and technical limitations (Bowker and Starr 1999, Kiviniemi 2005, Pazlar and Turk 2008). Therefore, rather than reading and writing to a common representational structure, we propose that tool nodes be coupled more flexibly through mapper nodes that allow direct data exchange via any data or file format.

The applicability of a particular mapping approach may also depend on the specific design scenario and, more importantly to us, on the skills of the designer. In this research, we are focusing on communities of users with limited programming skills, with the goal to develop a mapping approach that allows such non-programmers to easily create and share custom mappings in a collaborative manner. To achieve this goal, the mapping approach must be both flexible and user-friendly. It must be flexible so that users can apply the same mapping approach to any type of data. This approach must therefore be semantically agnostic. It must also be user-friendly to the extent that it supports users with limited programming skills in the process of creating and debugging mappers.

There will however always be cases where complex mappings need to be defined that require more advanced programming skills. Consider the example in Figure 1. The output from the CAD system cannot be easily mapped to either the input for EnergyPlus or input for Radiance. Instead, an additional step is required that performs Boolean operations on the polygons. For EnergyPlus, surface polygons need to be sliced where there are changes in boundary conditions (as each surface can only have one boundary condition attribute), and then infer what these boundary conditions are, i.e. internal, external or ground contact. For Radiance, surface polygons need to have holes cut where there are windows. These additional steps may have to be performed by a *scripted* mapper, denoted the PolygonSlicer.

The need for this additional mapper illustrates the difference between mappers that perform relatively straightforward manipulations of the data structure and data content, and mappers that perform more complex data transformations requiring specific computational functions. While we aim to support users of different skill levels to create these mappers, this may not always be possible with scripted mappers, which would typically need to be created by developers with programming skills. Ideally, such scripted mappers should be developed to apply to a wide variety of situations and contexts, so as to be easily reusable.

2. Data models for data mappers

With regards to creating mappers, a distinction can usefully be made between three steps: data reading, data transformation, and data writing. Reading and writing data to specific file formats and databases are functions that can be generated mainly using existing technology, specifically, parser generators and serialiser generators. Therefore, we focus here on the data transformation step, which requires more in-depth interpretation. An explicit model for data transformation is needed to relate constructs in the source representation to constructs in the target representation (Kilian, 2006). This may involve deriving new data as well as discarding data that is superfluous in the target format. The aim of the transformation is to create a target data set that is as close as possible to the target file format, so that serialisation is reduced to a very simple formatting procedure, which can be fully automated.

When transforming data, the data model provides ways of organising information at an abstract level, including defining the data structure, the data constraints, and the data operations. Two approaches to data modelling are the *general-purpose* approach versus the *domain-specific* approach. A general-purpose model organises data using generic constructs that are often highly domain independent. Due to this generic nature, the range of data that can be described tends to be very broad. It offers a way of defining a data structure that is very flexible but relies on human interpretation of semantic meaning.

A domain-specific model represents domain-specific information using semantic constructs related to a particular domain. The model may be defined using either a Data Definition Language (DDL) or a Domain Specific Modelling Language (DSML). Due to the highly specific nature of the constructs, the type of information that can be described tends to be relatively narrow. However, this manner of representing information supports automated interpretation of semantic meaning.

In many cases, domain-specific models are defined on top of a general-purpose one, by specifying additional constraints on the general-purpose model. An example is the many different XML schemas defined on top of the XML data model using the XML Schema DDL.

Data mappings will typically need to transform both the data content and the data structure. In the case of domain-specific models, when the semantic schemas for both the source data model and the target data model are available, the definition by the user of a list of semantically equivalent entities between these schemas may be sufficient. Based on this user-defined information, a mapping procedure can then be automatically generated that will transform the source data set to the target data set. In some cases, it may be possible to define such mappings using visual tools such as Altova's XML MapForce [3] (Altova 2005) that generates XSLT/XQuery code based on a mapping between elements in two XML schemas. We denote this approach a *declarative equivalency mapping*. One key problem with this approach is that only fairly simple mappings can be created using direct semantic mappings. More complex mappings may require a number of source entities to be processed in some way in order to be able to generate another set of target entities.

Alternatively, the user may create data transformation rules using languages specialised for particular types of data models. We denote this approach a *procedural query mapping*. These specialised languages include data query languages and data manipulation languages. The former are used for retrieving data from a data set, and the latter for inserting and deleting data in a data set. In many cases, the same language can be used for both querying and manipulation. A popular example is the Structured Query Language (SQL), which is used for both retrieving and manipulating data in relational databases. Other languages for retrieving and manipulating data include XQuery / XPath for data stored in XML data models, and SPARQL for data stored in Resource Description Framework (RDF) data models. Although such languages are specialised for certain data model, the languages themselves are still highly generic semantically.

For creating user-defined mappings within workflows, the procedural query mapping approach is seen as being more appropriate since it is semantically agnostic and therefore highly flexible. In such a case, the input data and output data for each of the tools could be made to adhere to the same general-purpose model, and this commonality would allow the tools to be more easily sharable. The user could download diverse tools (and possibly mappers) developed by different groups from a shared online repository, and then string these together into customised workflows (e.g., see the workflow in Figure 1). Each tool developer would specify a domain specific model for the input data, referred to as the input schema (and optionally also a domain specific model for the output data, referred to as the output schema). The user's task would then be to write mappers, where necessary, that generate data sets that adhere to the input schemas of the selected tools.

So far, the tools that have been considered have been design tools such as parametric modelling software and simulation programs. However, any existing tools that can be wrapped could be included in the tool library. Two types of tools that would likely be desirable are spreadsheet tools (such as Microsoft Excel) and data analysis and visualisation tools (such as Tableau Desktop [4]). In addition, existing data mapping tools could also be leveraged for creating specialised types of mappings. One example already mentioned above is Altova's Mapforce tool. Another example would be FME Desktop by SafeSoftware [5] (SafeSoftware 2008), which allows users to visually construct data mappings for geo-spatial data types by selecting and configuring predefined sets of data transformers. Although such mapping tools may be useful in certain cases, a more general approach to creating mappings would nevertheless still be needed.

With respect to the procedural query mapping approach, we have considered various general-purpose data models and query languages from the point of view of applicability and ease of use. The three main data models that were considered are relational data models using SQL, XML data models using XQuery, and property graphs using Gremlin [6].

Relational data models organise data into table structures consisting of rows and columns, XML data models organise data into hierarchical trees consisting of elements and attributes, and property graphs organise data into network structures consisting of vertices and edges. Of the three, the property graph is the least well known, and is a directed graph data structure where edges are assigned a direction and a type and both vertices and edges can have attributes called properties. This allows property graphs to represent complex data structures with many types of relationships between vertices. In graph theoretic language, a property graph is known as a directed, attributed, multi-relational graph. Many other graph data models such as RDF graphs can be viewed as a special kind of property graph. Gremlin is a domain-specific graph traversal language for navigating such graphs.

Typically, modelling and simulation of design problems requires data structures with highly complex relationship networks. Relational data models and XML data models are not able to represent such complex networks in an elegant way. Furthermore, the query languages used for these models are cumbersome when working with complex relationship networks. This is especially problematic in supporting users with little or no programming skills to understand these complex relationship networks. In the case of SQL, “join” clauses are required to relate data in multiple tables, while in XQuery, “idref” functions have to be used to relate data from different parts of the hierarchical tree. In contrast, property graphs use an inherently networked data structure and therefore do not suffer from these drawbacks. Below, an example scenario is described in which the property graph data model was used as the underlying general-purpose data model, and Gremlin as the query and manipulation language.

3. Example scenario

In order to demonstrate the feasibility of our approach, we have implemented part of the example scenario shown in Figure 1. A parametric CAD system is used to generate a model of a small building and EnergyPlus and Radiance are used to evaluate building performance. In this scenario, we have used an existing workflow system called Kepler [7] (McPhillips et al, 2009) to connect the various tools together. The Kepler workflow is shown in Figure 2.

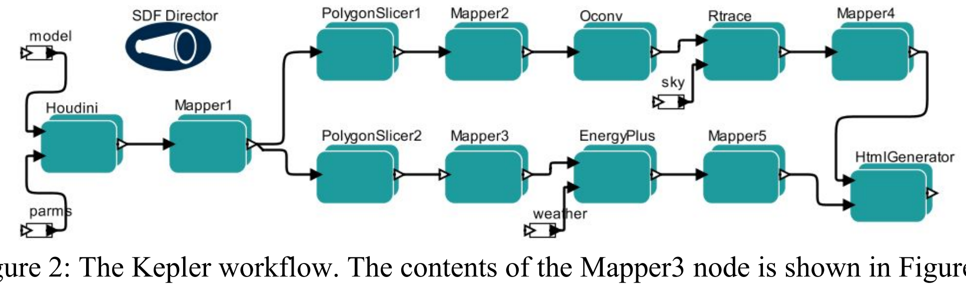
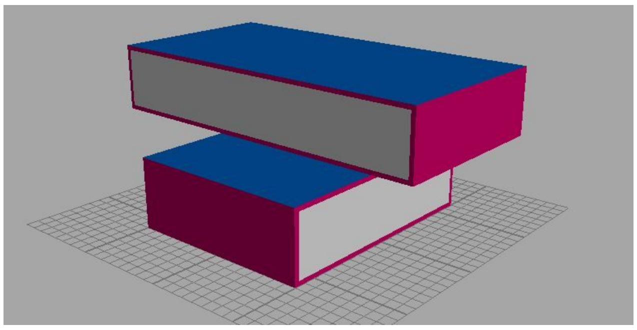
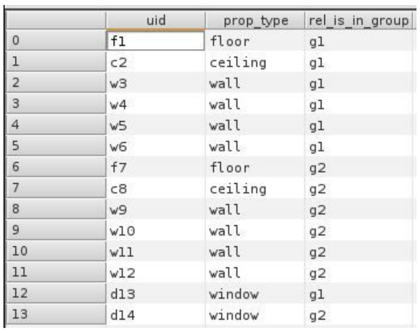


Figure 2: The Kepler workflow. The contents of the Mapper3 node is shown in Figure 5.

For this demonstration, the model is highly simplified, consisting of only two spaces stacked on top of each other, each with a window in one wall (Figure 3). In total there are 14 polygons, and each polygon is assigned a set of attributes that are used for generating the property graphs.

SideFX Houdini [8] is used as the parametric CAD system, but various other (parametric) CAD software could also be used. The main requirement for this system would be the ability to create a customised procedure to output the geometric data as a simple JSON file (using the GraphSON library [9]). Existing parsers can then be used to generate a property graph model from such a file. Similarly, EnergyPlus is used as the energy analysis simulation program and Radiance is used as the lighting analysis program, but other simulation tools can also be considered for this purpose. The Houdini application, the EnergyPlus program, and the two Radiance programs have been wrapped in Python wrappers and scripted mapper nodes have been created to transform the inputs and output files into GraphSON files.





	uid	prop_type	rel_is_in_group
0	f1	floor	g1
1	c2	ceiling	g1
2	w3	wall	g1
3	w4	wall	g1
4	w5	wall	g1
5	w6	wall	g1
6	f7	floor	g2
7	c8	ceiling	g2
8	w9	wall	g2
9	w10	wall	g2
10	w11	wall	g2
11	w12	wall	g2
12	d13	window	g1
13	d14	window	g2

Figure 3: The CAD model consisting of 14 polygons, each with three attributes (one unique ID called "uid", one property called "prop_type", and one relationship called "rel_is_in_group").

In order to define the graph mappings, a set of nodes were created in Kepler to provide a set of basic mapping functions for mapping graph vertices and graph edges. These nodes have various parameters that allow users to customise the mappings. When these nodes get executed, Gremlin mapping scripts are automatically generated based on these parameter settings.

Three different types of mapping nodes were created: graph manipulation nodes, vertex creation nodes, and iterator nodes. Graph manipulation nodes are used for merging, splitting, and filtering graphs. Vertex creation nodes are used for creating new vertices from scratch. Iterator nodes are used for iterating over the contents of an input graph and triggering certain actions. For each type of node, a parameter called "select" allows users to specify a Gremlin selection filter on the input graph. For each entity (i.e. vertex or edge) in the filtered input graph, a particular action is triggered, which could be the creation or modification of vertices or edges.

The mapping process from Houdini to EnergyPlus will be described in more detail. The first graph mapper maps the output from Houdini to the input of the PolygonSlicer. The second mapper maps the output of the PolygonSlicer to the input of EnergyPlus. In both mappers, the different GraphSON files are automatically parsed into and serialised from property graphs by existing tools and, therefore, the user only needs to focus on the transformation of the property graphs. Figure 4 shows the overall structure of the property graphs, and Figure 6 shows the properties associated with three of the vertices in each property graph.

The first step is for the user to define the parametric model of the design together with a set of parameter values. The Houdini wrapper will trigger Houdini to generate a model instance and will retrieve the geometric data for that instance. The geometric data will then be automatically restructured as a property graph and saved as a GraphSON file. Points and polygons in the model will be mapped to vertices in the property graph. In addition, the user can influence the restructuring process by defining a set of attributes for the polygons in the parametric model. An attribute with the name "uid" (i.e. unique ID) is used to define the node name, attributes with names that starts with "prop_" will result in properties being added to the vertices, and attributes with names that start with "rel_" will result in relationships being generated between vertices.

In this scenario, the user knows that in order to map to EnergyPlus, surfaces will need to be assigned different types and will also need to be grouped into zones. The user has therefore defined certain attributes in order to simplify this mapping process. Each polygon has been given a "uid" attribute, a "prop_type" attribute that will define a type for each surface, and a "rel_is_in_group" attribute that groups surfaces according to zones (see Figure 3).

The user then needs to create the graph mappers using the graph mapping nodes. The PolygonSlicer and the EnergyPlus simulator both have input graph schemas that specify the required structure of the graph and the required properties of the vertices. The task for the user is therefore to create mappings that generate graphs that adhere to these schema constraints. Figure 4 shows the overall structure of the property graphs at each stage.

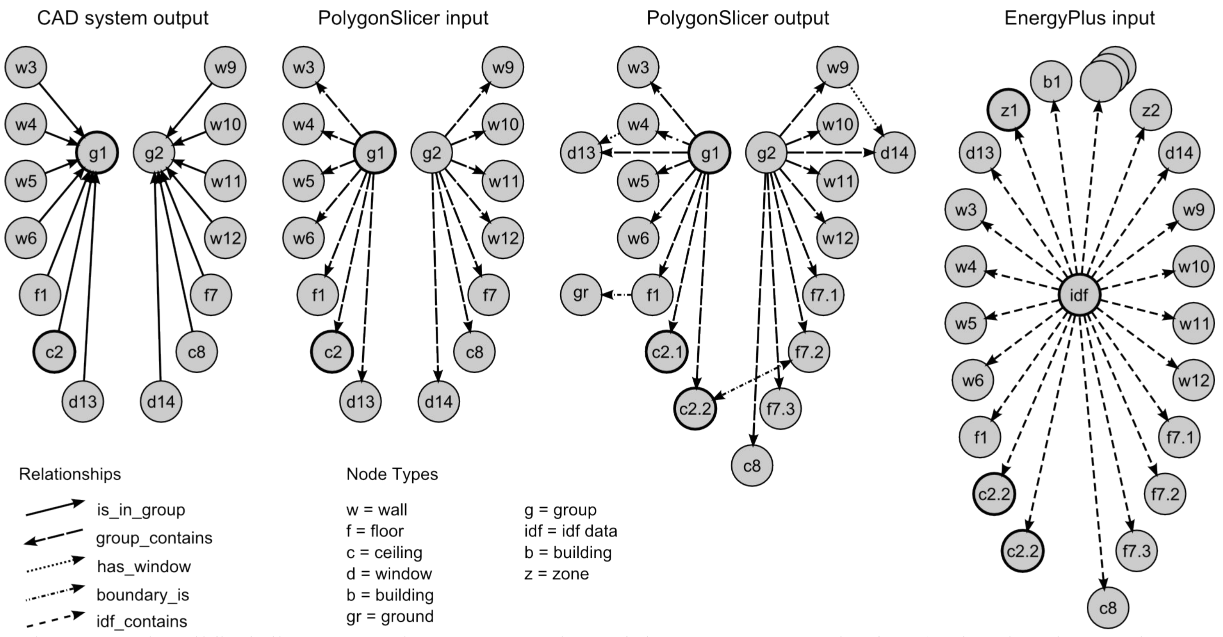


Figure 4: Simplified diagrammatic representation of the property graphs for. (Point data is not shown in order to reduce the complexity of the diagrams. In the actual graph, there are also 24 points, each with x,y, and z properties.)

In the first mapping, where the output of Houdini is mapped to the input of the PolygonSlicer, the number of vertices remains constant, but the edges between the vertices need to be reversed and the edge label changed from "is_in_group" to "has_a". Iterator nodes are used to reverse the edges and to add "type" properties to the nodes. The PolygonSlicer then transforms its input graph by dividing the surfaces for the ceiling of the lower zone ("c2") and the floor of the upper zone ("f7") so as to ensure that each surface has a homogeneous boundary condition. The PolygonSlicer also detects the relationships between the floors and ceilings, between the floors and the ground, and between windows and walls.

In the second mapping, where the output of the PolygonSlicer is mapped to the input of the EnergyPlus simulator, additional properties are added to the existing vertices in the input graph, and a number of additional vertices are also added to define a set of other objects required in the EnergyPlus input file. Vertex creator nodes are used to create the additional vertices, and iterator nodes are used to copy and modify existing vertices (Figure 5). The groups are mapped to EnergyPlus zones, and the polygons to EnergyPlus surfaces. In the process of mapping, the iterator node also transforms the edges that existed in the input graph into properties in the output graph. The output graph becomes a simple list of vertices under the "idf" root node. For example, in the input graph the window is connected to the wall with an edge, while in the output graph the window is no longer connected but instead has a property that specifies the wall name.

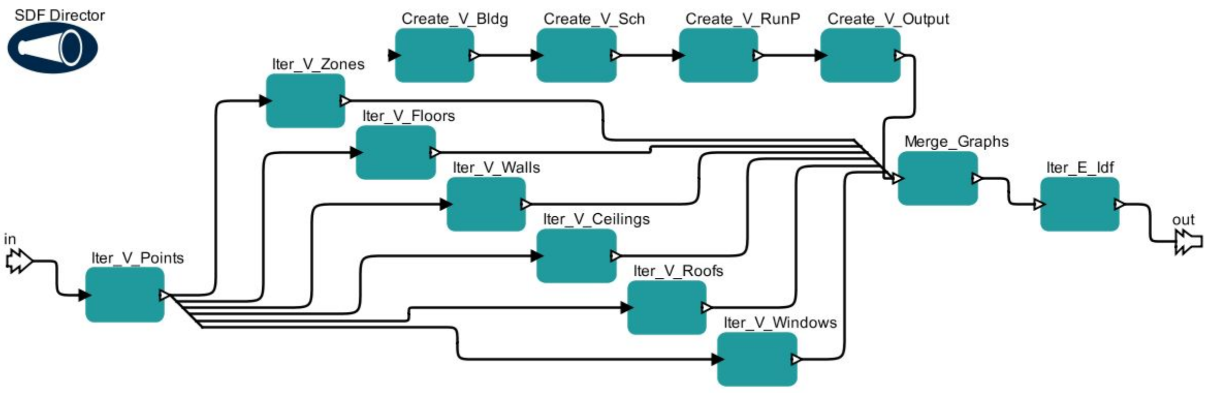


Figure 5: The Kepler mapper that maps the output of the PolygonSlicer node to the input of the EnergyPlus node. See the "Mapper3" node in Figure 2.

For each different surface type, a separate iterator node is created. For example, consider the "Iter_V_Ceilings" node in Figure 5. This node generates the ceilings of the two zones. Table 1 shows the two main parameters for the node. The "select" parameter filters the input graph so that the remaining vertices all have an "Entity" property with a value of "polygon" and a "Type" property with a value of "ceiling", and in addition have an outgoing "boundary_is" edge that points to another polygon (i.e., the floor above). The node then has a set of parameters that define name-value property pairs. For each polygon in the filtered input graph, the iterator node will generate a vertex in the output graph with the specified properties.

Table 1: The parameter names and values for the Iter_V_Ceilings node. Gremlin code is shown in italics, and makes use of two predefined local variables: 'g' refers to the input graph, and 'x' refers to the entity being iterated over (which in this case is a vertex).

Parameter	**Parameter value**		
Filter graph: Select	*g.V.has('Entity','polygon').has('Type','ceiling').as('result') .out('boundary_is').has('Entity','polygon').back('result')*		
Generate vertices: Vertex properties	Object	:	'BuildingSurface:Detailed'
	Name	:	*x.Name*
	Surface_Type	:	'CEILING'
	Construction_Name	:	'light ceiling'
	Zone	:	*x.in('group_contains').Name*
	Outside_Boundary_Cond	:	'SURFACE'
	Outside Boundary_Cond_Object	:	*x.out('boundary_is').Name*
	Sun_Exposure	:	'NOSUN'
	Wind_Exposure	:	'NOWIND'
	Points	:	*x.Points*

Note that when the user is specifying the property values, they can insert Gremlin commands that extract these values from the input graph, thereby ensuring that the values can be dynamically generated. Figure 6 shows the changes for a number of vertices in the property graph as the data is mapped and transformed. When the "Iter_V_Ceilings" node iterates over the "c2.2" polygon in the input graph, it will generate the "c2.2" EnergyPlus surface.

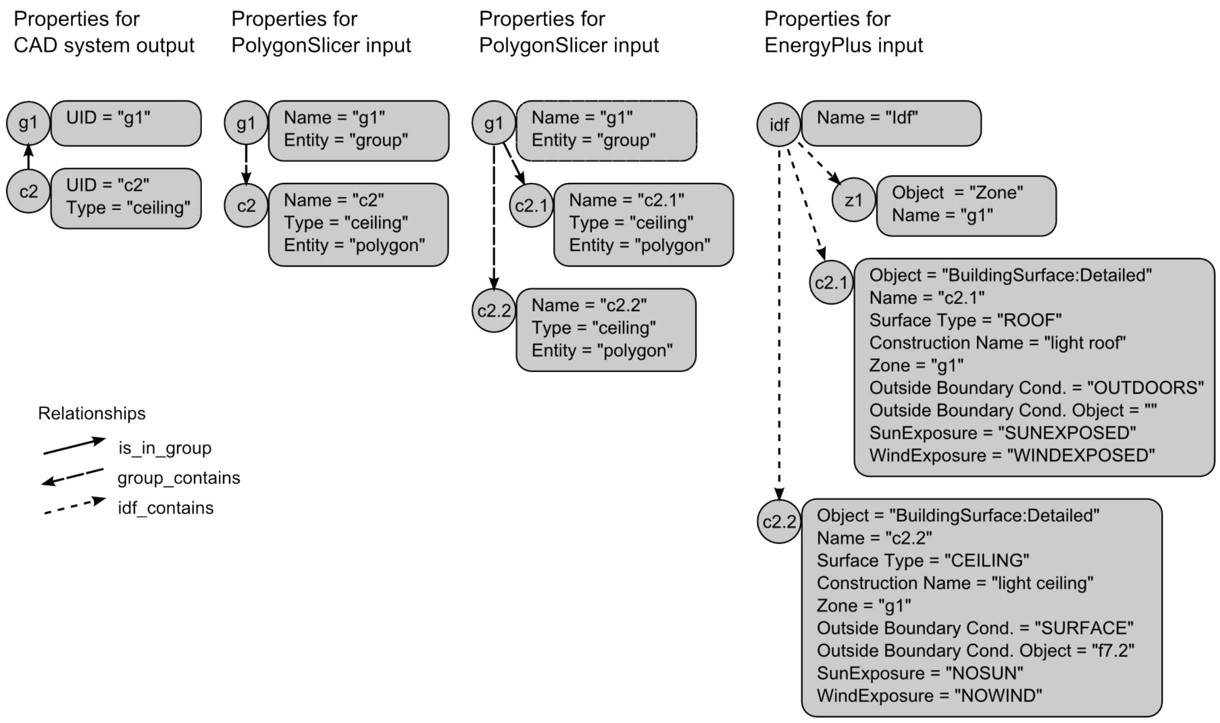


Figure 6: An example of the property data for a few of the vertices in the property graphs. Typically, the property graphs will undergo a process of information expansion, where data is gradually added to the model as needed.

4. Future work

In order to support a bottom-up, user-controlled and process-oriented approach to linking design and analysis tools, a data mapping approach is required that allows designers to create and share custom mappings. To achieve this goal, the data mapping approach should be both flexible in that it can be applied to a wide variety of tools, and user-friendly in that it supports non-programmers in the process of easily creating and debugging mappers. The use of a general-purpose data model ensures that the mapping approach is highly flexible, and in addition also allows for improved sharability. The example scenario demonstrated how designers with minimal scripting skills would be able to set up complex digital workflows that enable the fluid and interactive exploration of design possibilities in response to custom performance metrics.

The next stage of this research will explore the scalability of the user-defined graph mapping approach when working with larger data sets and more complex data schemas (such as the IFC schema). In the current demonstration, the data sets and data schemas are small, and as a result the graph mappers are relatively simple. However, if data sets increase in size and the number of entity and relationship types is very large, then the graph mappers could potentially become more difficult to construct. In order to deal with this increased complexity, we foresee that the user will require additional data management and schema management tools. The data management tools could enable users to visualise, interrogate and debug property graph data during the mapping process (Stouffs, 2001). An example is Gephi [10], a rich interactive tool for analysing and visualising complex graph data sets (Bastian et al, 2009). Schema management tools could enable node developers to define formal graph schemas for the input and output data for their nodes. This could in turn allow end-users to identify and isolate subsets of large schemas relevant to their particular design scenario.

Acknowledgements

The authors would like to acknowledge and thank the participants of the “Open Systems and Methods for Collaborative BEM (Building Environment Modelling)” workshop held at the CAAD Futures 2011 Conference in Liège, Belgium, 4 July 2011, and of the LinkedIn Group sharing the same name, for their contributions to the discussions leading to the ideas presented and described in this paper. We invite interested parties to contribute to the development of these ideas and to join the discussions in the LinkedIn Group.

References

Altıntaş, İ.: 2011, Collaborative Provenance for Workflow-driven Science and Engineering, PhD Thesis, University of Amsterdam, Amsterdam.

Altova Inc.: 2005, Data Integration: Opportunities, challenges, and Altova MapForce™ 2005, Whitepaper available at: http://www.altova.com/whitepapers/mapforce.pdf

Bastian M., Heymann S., Jacomy M.: 2009, Gephi: an open source software for exploring and manipulating networks. International AAAI Conference on Weblogs and Social Media.

Bowker, G.C. and Starr, S.L.: 1999, Sorting Things Out: Classification and Its Consequences. MIT Press, Cambridge, MA.

Coons, S.A.: 1963, An outline of the requirements for a computer-aided design system, Proceedings AFIPS, ACM, 299-304.

Curcin, V. and Ghanem, M.: 2008, Scientific workflow systems - can one size fit all?, CIBEC 2008, Cairo, 1-9.

Deelman, E., Gannon, D., Shields, M. and Taylor, I.: 2008, Workflows and e-science: An overview of workflow system features and capabilities, Future Generation Computer Systems, 25, 528-540.

Kilian, A.: 2006, Design innovation through constraint modeling, International Journal of Architectural Computing, 4(1), 87–105.

Kiviniemi, A.: 2006, Ten years of IFC development - why we are not there yet. Proceedings CIB-W78, Montreal.

McPhillips T., Bowers S., Zinn D. and Ludaescher B.: 2009, Scientific workflow design for mere mortals, Future Generation Computer Systems, 25(5), 541-551.

Pazlar, T. and Turk, Z.: 2008, Interoperability in practice: geometric data exchange using the IFC standard, ITcon, 13, 362-380.

SafeSoftware: 2008, FME Desktop Under the Hood, Whitepaper available at:
http://cdn.safe.com/resources/whitepapers/FME-Desktop-Under-the-Hood.pdf

Stouffs, R.: 2001, Visualizing information structures and its impact on project teams: an information architecture for the virtual AEC company, Building Research & Information, 29(3), 218–232.

Toth, B., Boeykens, S., Chaszar, A., Janssen, P., & Stouffs, R.: 2012, Custom digital workflows. A new framework for design analysis integration. Beyond Codes & Pixels. 17th CAADRIA Conference Proceedings. Chennai (India), 163-172.

[1] http://apps1.eere.energy.gov/buildings/energyplus/

[2] http://radsite.lbl.gov/radiance/

[3] http://www.altova.com/mapforce/xml-mapping.html

[4] http://www.tableausoftware.com/products/desktop

[5] http://www.safe.com/fme/fme-technology/fme-desktop

[6] http://gremlin.tinkerpop.com

[7] https://kepler-project.org/

[8] http://sidefx.com

[9] https://github.com/tinkerpop/blueprints/wiki/GraphSON-Reader-and-Writer-Library

[10] https://gephi.org/