# Custom Digital Workflows with User-defined Data Transformations via Property Graphs

**Patrick Janssen**[1], **Rudi Stouffs**[2], **Andre Chaszar**[3], **Stefan Boeykens**[4], **and Bianca Toth**[5]
[1,2]*National University of Singapore, Singapore*
[2,3]*Delft University of Technology, Netherlands*
[4]*KU Leuven, Belgium*
[5]*Queensland University of Technology, Australia*

Custom digital workflows aim to allow diverse, non-integrated design and analysis applications to be custom linked in digital workflows, created by a variety of users, including those who are not expert programmers. With the intention of introducing this in practice, education and research, this paper focuses on critical aspects of overcoming interoperability hurdles, illustrating the use of property graphs for mapping data between AEC software tools that are not connected by common data formats and/or other interoperability measures. A brief exemplar design scenario is presented to illustrate the concepts and methods proposed, and conclusions are then drawn regarding the feasibility of this approach and directions for further research.

## Introduction

The persistent lack of integration in building design, analysis and construction calls for new approaches to information exchange. We argue that bottom-up, user-controlled and process-oriented approaches to linking design and analysis tools, as envisaged by pioneers of CAD [1,2], are indispensable as they provide degrees of flexibility not supported by current top-down, standards-based and model-oriented approaches.

We propose a platform to support a design method where designers can compose and execute automated workflows that link computational design

*Design Computing and Cognition DCC'14*. J.S. Gero (ed),
pp. xx-yy. © Springer 2014

tools into complex process networks [3,4]. By allowing designers to effectively link a wide variety of existing design analysis and simulation tools, such custom digital workflows support the exploration of complex trade-offs between multiple conflicting performance criteria. For such explorations, these trade-offs often present a further set of questions rather than a final set of answers, so the method is envisaged as a cyclical process of adaptive workflow creation followed by iterative design exploration.

The adaptive-iterative design method requires a platform for designers to effectively and efficiently create and execute workflows. The remainder of this paper first gives a general overview of the proposed platform and then focuses on critical aspects of overcoming interoperability hurdles, specifically the creation of mapping procedures that map data between tools with incompatible data representations. We explore the feasibility of a data mapping approach that allows end-users to define their own customized mappers, applying it to an example scenario focusing on a digital design-analysis workflow linking parametric design tools to performance analysis tools.

The research method consists of building a test workflow comprising a geometric design model and analysis tools to evaluate lighting and thermal performance, and applying customized data mappings between these applications via property graphs. The data collected from this experiment includes observations of the types of data mappings required, the complexity of the mappings, and the modifiability of the mappings when editing of the workflow is needed. We conclude that linking design tools via customized data mappers is a feasible approach that can complement other existing mapping approaches, and we discuss future research directions.

## Adaptive-iterative design platform

In order to support the adaptive-iterative design method, a platform for creating and executing workflows is proposed. This platform is based on existing scientific workflow systems that enable the composition and execution of complex task sequences on distributed computing resources [5].

Scientific workflow systems exhibit a common reference architecture that consists 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 [6,7]. Nearly all workflow systems are visual programming tools in that they allow processes to be described graphically as networks of nodes and wires that can be configured and reconfigured by

users as required [8]. Nodes perform some useful function; wires support the flow of data, linking an output of one node to an input of another node.

Each workflow system acts to accelerate and streamline the workflow process, but each system also varies greatly in specific capabilities [9]. We aim to identify the functionality needed to develop a flexible, open and intuitive system for design-analysis integration, building on the capabilities exhibited by scientific workflow systems, and further capitalizing on recent advances in cloud computing.

## Actor model

The proposed platform is a type of scientific workflow system using an actor model of computation [10]. Nodes are *actors*, and the data that flows between nodes is encapsulated in distinct data sets, referred to as *data objects*. The actor model allows for a clear separation between actor communication (dataflow) and overall workflow coordination (orchestration).

We consider three types of actors: process actors, data actors and control actors. *Process actors* define procedures that perform some type of simulation, analysis, or data transformation. They have a number of input and output ports for receiving and transmitting data objects, as well as meta-parameters that can be set by the user to guide task execution. *Data actors* define data sources and data sinks within the workflow, including the data inputs and data output for the workflow as a whole. *Control actors* provide functionality related to workflow initiation, execution and completion. We focus here on the role of process actors in workflows, and the development of an approach to support custom data mapping procedures.

Process actors can be further classified into tool actors and mapping actors. Tool actors define procedures that wrap existing applications to make their functionality and data accessible to the workflow; while mapping actors define procedures that transform data sets in order to map the output from one tool actor to the input for another.

Figure 1 shows a conceptual diagram of an example network in which a parametric CAD system actor is connected to three evaluation actors: Revit for cost estimating; EnergyPlus for heating/cooling load simulation; and Radiance for daylighting simulation. The CAD system actor encapsulates a procedure that starts the CAD system, loads a specified input model and a set of parameter values, and then generates two output models. One of the models is a generated as an IFC file, which is directly consumed by the Revit actor (using the Geometry Gym Revit IFC plugin [11]), while the other model is generated as a text file. This latter model then undergoes two separate transformations that map it into both EnergyPlus and Radi-

ance compatible formats. The simulation actors then read in this transformed data, run their respective simulations, and generate output data consisting of simulation results.

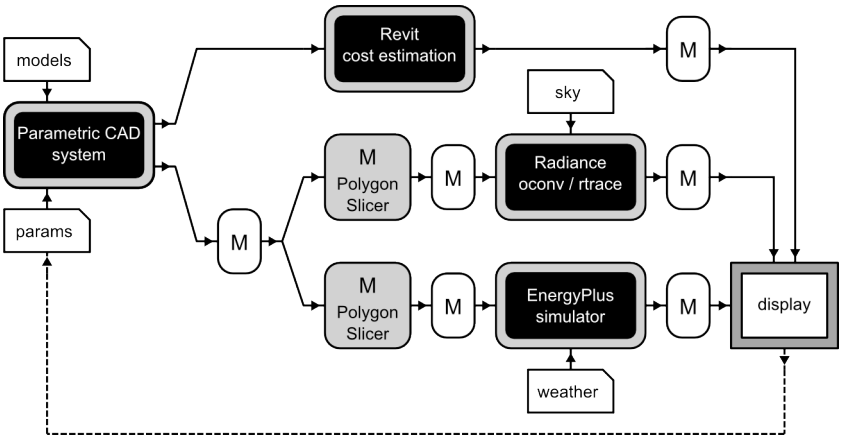



**Fig1.** Example network of actors. A parametric CAD system is linked to Revit, Radiance and EnergyPlus via various mappers (M). End users contribute white components, while actor developers build grey components and wrap black components representing existing tools.

The output results may be merged and displayed to the user within a graphical dashboard in an integrated way in order to support decision making. Furthermore, such a dashboard may also allow users to manipulate the input parameters for the workflow. Each time such inputs are changed, the execution of the network will be triggered, resulting in a new set of results. It is envisaged that more advanced data analytics tools could be developed in which multiple sets of results from the adaptive-iterative process could be analyzed and compared.

## Platform architecture

In order to support both the collaborative creation of workflows and their parallel execution, the proposed platform can be implemented as a web application deployed on a cloud or grid based infrastructure. Workflows (such as the one shown in Figure 1) can be interactively developed by users within the web browser interface. When the user runs a workflow, it is uploaded to a workflow server as a workflow *job* consisting of a set of computational *tasks*, where it is executed in a distributed manner.

For scalability, both the procedures and the associated data objects are stored in an online distributed key-value database. Key-value databases are highly optimized for speed and scalability when storing and retrieving many documents. Although such documents may use a standardized format (e.g. JSON), they are typically used as containers for storing other data, with no restrictions on data models and data schemas. For example, a document may contain any number of serialized objects, text files or binary blobs. Both data objects and actor procedures are stored in documents in the key-value database. No restrictions are imposed on the types of data objects and actor procedures that can be stored.

## Data models for data mappers

Data mappings aim to overcome interoperability problems between tools that cannot be linked using existing approaches. We examine two common approaches for creating such mappers, and propose a complementary approach.

The various approaches to overcoming interoperability problems rely on three distinct levels of data representation: the *data model*, the *data schema*, and the *data file*. A data model organizes data using generic constructs that are domain independent. Due to this generic nature, the range of data that can be described is very broad. It offers a way of defining a data structure that is very flexible but relies on human interpretation of semantic meaning. For example, Tsichritzis and Lochovsky [12] distinguish seven types: relational, network, hierarchical, entity-relationship, binary, semantic network, and infological. Data models will often be coupled with highly generic languages for querying and manipulating data, variously called *data query languages* and *data manipulation languages*.

A data schema represents domain-specific information using semantic constructs related to a particular domain. 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. The data schema is often built on top of a data model, by formally defining constraints that describe a set of allowable semantic entities and semantic relationships specific to a particular domain. This data schema is defined using a specialized language, variously called a *data definition language*, a *data description language,* or a *data modeling language*. Note that the data schema itself is distinct from the language used for describing the schema.

A data file uses syntactic constructs to describe how data is stored, either in a file or any other form of persistent storage. Going from the data file to the data schema is referred to as *parsing*, while the reverse is known as *serializing*. For any given data schema, there may be many different types of data files.

For example, consider the Industry Foundation Classes (IFC) representation for the exchange of Building Information Modeling data in the AEC industry [13]. In this case, the data model is an entity-relationship model, the data schema is an IFC schema defined in the EXPRESS modeling language, and the two main data files use a STEP or XML based file format. Note the use of XML for one of the file formats does not mean that an XML-based data model is being used for data manipulation. The data file is only used as a convenient way of storing the data.

**Tool interoperability**

To link two tools with incompatible data representations, a formalized data mapping needs to take place [14]. The mapping is defined as a one-way transformation process, where the data representation of the source tool is mapped to the data representation of the target tool. In existing scientific workflow systems, this mapping process is called 'shimming' [15].

In most cases, the incompatibility exists at all levels: at the data model level, the data schema level, and the data file level. One approach in overcoming this type of incompatibility is to create direct file translators for each pair of tools to be linked. An alternative approach is to create indirect file translators that read and write to an intermediate representation. Within the AEC industry, these are the two main interoperability approaches being pursued, which we call *direct file translation* and *indirect file translation*.

The direct file translation approach has two key weaknesses. First, since separate data file translators need to be created for each pair of tools, the number of translators required increases rapidly as the number of tools increases. Second, the required generality of the translator means that no assumptions can be made regarding the sets of data to be processed, so these translators are complex to develop and maintain (as file formats are continuously updated and changes are often undocumented), and tend to be plagued by bugs and limitations.

With the indirect file translation approach, a range of different levels exist depending on the type of data schema. At the low-level end of the spectrum, data schemas may be limited to describing only geometric entities and relationships, while at the more high-level end of the spectrum, schemas may also describe complex real-world entities and relationships.

The AEC industry has been using low-level geometric translators for many decades, based on file formats such as DXF, IGES and SAT. Such low-level geometry based file translation approaches are clearly limited, since they do not allow for the transfer of non-geometric semantic data.

At the more high-level end of the spectrum, the AEC research community has, since the 1970s, been working on ontological schemas that combine geometric information with higher level semantic constructs that describe entities and relationships in the AEC domain. Until recently, this approach was often conceptualized as a single centralized model containing all project data and shared by all project participants. However, this is now increasingly seen as being impractical for a variety of reasons, and as a result such high-level ontologies are now being promoted as a 'smart' form of file translation [16]. The latest incarnation of these efforts is the STEP-based IFC standard for BIM data storage and exchange.

**Open interoperability**

A number of platforms are being developed that enable users to apply both direct and indirect file translators, focusing in particular on BIM and IFC. For example, the SimModel aims to allow BIM and energy simulation tools to be linked via a new XML based model aligned with IFC [17]; D-BIM workbench aims to allow a variety of tools for analyzing differing performance aspects to be tightly integrated with BIM tools [18].

However, there are many sets of tools for which such translators have either not yet been developed or are no longer up-to-date. We therefore propose a more flexible and open type of platform that also supports linking tools for which direct and indirect file translators are not available. Such tools are linked using an approach that allows users to define their own customized data mappers tailored to the data exchange scenarios they require. Figure 2 shows the three approaches to linking tools. Note that this figure omits other approaches that do not require the neutral file format approach, e.g., agent-based interoperability [19]. The proposed platform will allow users to create automated workflows that mix-and-match all three tool-linking approaches, applying the most expedient approach for each pair of tools. For example, in cases where IFC-based mappers are already available, the user may prefer to simply apply such mappers, while in cases where no translators are available, customized data mappers can be developed.

End users will need to be provided with various tools and technologies in order to support them in this endeavor of developing mappers. This research focuses 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. Since users are focused on one specific data exchange scenario, these mappers can potentially be restricted to only a subset of the data schemas of the design tools being linked, thereby resulting in mappers that are orders of magnitude smaller and simpler when compared to general-purpose file translators.

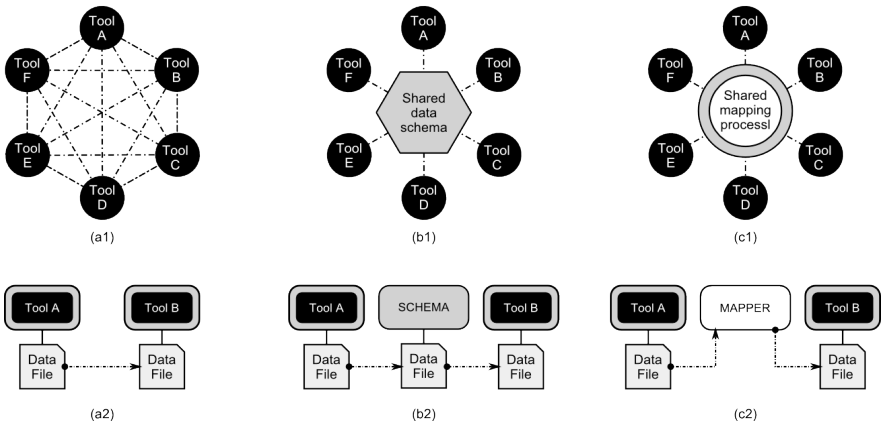



**Fig 2.** Interoperability approaches. Linking individual tools via file translation (left); via a shared data schema (middle); or via a shared mapping process (right).

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 strategies to a wide range of data exchange scenarios. It must also be user-friendly so that it supports users with limited programming skills in the process of creating and debugging mappers.

The complexity of creating such mapping is to a significant extent related to the levels of data incompatibility between the source tool and target tool. So far, the assumption has been that for most tools, data incompatibility will exist at all three levels: data model, data schema, and data file. This creates many difficulties for end-users attempting to create mappers. In particular, the user is required to deal with two different data models, each of which will have its own languages for data query, data manipulation, and schema definition. However, a simpler data exchange scenario can be imagined, where the input and output data for the mapping both use the same data model. In such a case, the user would be able to develop a mapping within one coherent data representation framework, including the option of using a single model based language for querying and manipulating data. In addition, given a single data model, it then be-

comes feasible to provide visual tools for developing such mappers, as will be discussed in more detail in the next section.

The simpler data exchange scenario with a common data model is clearly more desirable. The proposed approach is therefore to transform data exchange scenarios with incompatible data models into the simpler type of data exchange scenario where both input and output data sets have a common data model. With this approach, input data sets are created that closely reflect the data in the source data file, and output data sets are created that closely reflect the data in the target data file. The mapping task is then reduced to the simpler task of mapping from input data sets to output data sets using a common data model. Figure 3 shows the relationship between the mapping procedure and the source and target data files.

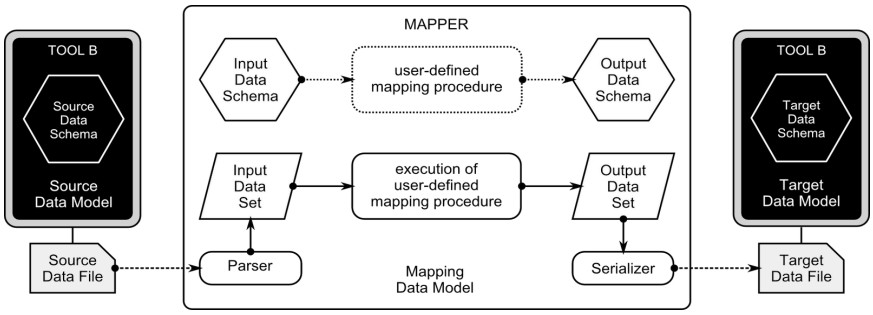



**Fig 3.** The mapping of a source data file to a target data file involves three stages of data transformation: parsing, mapping, and serializing.

These input and output data sets may have data schemas, either informally defined in help documentation or formally defined using a schema definition language. In the latter case, the schema may be used to support visual mapping tools. In most cases, these schemas will be 'reduced' schemas in that they will only cover a sub-set of the data that might be represented in the source and target data files. For example, while the source and target tools may each have large and complex schema, a user may decide that for the task at hand, they will only be using a small sub-set of the entities and relationships defined in those schemas. As a result, these schemas may typically be small and highly focused on the task at hand.

The input and output data sets can be generated in various ways. One approach is to use a parser to convert the source data file into the input data set and a serializer to convert the output data set into the target data file. In both reading and writing the data files, the data is converted with the minimum amount of change, and neither the parser nor the serializer performs any actual data mapping. As long as a generic and flexible mapping

data model is selected, the conversions between data file and data model should be relatively straightforward. In some cases, it may even be possible to automatically generate parsers and serializers. We focus here on the data mapping procedure, which requires more in-depth interpretation.

**User defined data mappings**

Given a pair of automated parsing and serializing procedures, the task for the user is then to define the data mapping. This mapping procedure needs to be able to process any data set that adheres to the input data schema, and produce a data set that adheres to the output data schema. For creating such mapping procedures, a number of techniques can be identified, differing in complexity and power. For the simplest type of mappings, which we refer to as *declarative equivalency mapping*, user-friendly visual tools already exist, while for more complex types of mappings, which we refer to as *scripted mappings,* the user is forced to write code. In this research, we propose a powerful intermediate approach using flexible and highly generic visual programming tools. This intermediate level we refer to as *procedural query mapping*.

With declarative equivalency mappings, the input and output data schemas are formally defined, and the user defines a list of semantically equivalent entities between these schemas. Based on this user-defined information, a mapping procedure can then be automatically generated that will transform the input data set to the output data set. In some cases, it may be possible to define such mappings using visual tools [20]. 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 input entities to be processed in some way in order to be able to generate another set of target entities.

With procedural query mappings, the user creates data mapping rules using visual programming languages specialized for particular types of data models. These specialized 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 generic languages for retrieving and manipulating data include XQuery / XPath for data stored in XML data models, SPARQL for data stored in Resource Description Framework (RDF) data models, and jaql for data stored in JSON data models. Although such languages are special-

ized for certain data models, the languages themselves are semantically still highly generic.

With scripted mappings, the user develops mapping scripts using a programming language. Such scripted mapping may be appropriate in cases where complex mappings need to be defined. 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, the PolygonSlicer (Figure 1).

For creating user-defined mappings within workflows, either declarative equivalency mappings or procedural query mappings approach are seen as being more appropriate, since these approaches do not require advanced programming skills. However, if more complex types of mappings are required, then scripted mappers can be created. Ideally, such scripted mappers should be developed to apply to a wide variety of situations and contexts, so as to be easily reusable.

For declarative equivalency mappings, a number of tools already exist, and for scripted mappers, query and manipulation languages abound. However, visual tools for procedural query mapping are rare. In addition, this approach is seen as being particularly crucial, since it is not subject to the limitations of the simpler declarative equivalency mapping approach, while at the same time it does not require the more advanced programming skills for the more complex scripted mapping approach. This research therefore specifically focuses on the development of a set of visual tools for developing procedural query mappings.

## Data models for mapping

It is envisaged that these various tools, parsers, serializers, and mappers could be developed and shared through an online user community. Users could download diverse sets of actors developed by different groups from a shared online repository, and then string these together into customized workflows (e.g., Figure 1). This process would ideally emerge in a bottom-up manner with minimal restrictions being placed on developers of actors. It is therefore important that no specific data model is imposed, but instead that actor developers and other users are able to choose preferred data

models. For a particular pair of tool actors, various parser-serializer pairs may be developed allowing users to choose to generate mappings based on alternative data models. For example, one parser-serializer pair might use a hierarchical XML data model, allowing users to create mappings with their preferred declarative equivalency mapping tool, while another might use a relational data model, allowing users to create a mapping by writing SQL scripts. Ideally, a diverse ecosystem of actors would emerge.

For procedural query mapping, various data models and query languages were considered from the point of view of applicability and ease of use. Below, an example scenario is described in which the property graph data model was used as the mapping data model. A property graph 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. The query language used for querying and manipulating data in the property graphs is called Gremlin [21].

## Example scenario

In order to demonstrate the feasibility of the proposed approach, we have implemented part of the example scenario shown in Figure 1 using Kepler [22], an open-source workflow system based on an actor-oriented software framework called Ptolemy II [23]. Kepler workflows can be nested, allowing complex workflows to be composed from simpler components, and enabling workflow designers to build reusable, modular sub-workflows that can be saved and used for many different applications.

Kepler is used to create a workflow connecting various tools, including SideFX Houdini as a parametric CAD system to generate a building model and EnergyPlus and Radiance as energy analysis simulation program and lighting analysis program, respectively, to evaluate building performance. Any other (parametric) CAD software and simulation tools could also be considered for this purpose.

A simplified design is used for testing the workflow, consisting of only two spaces stacked on top of each other, each with a window in one wall. The Kepler workflow is shown in Figure 4; here the actors created wrap the Houdini application, the EnergyPlus program, and the two Radiance programs (using Python as the programming language). In total there are 14 polygons, and each polygon is assigned a set of attributes that are used

for generating the property graphs. The model is shown in Figure 5, together with key attributes defined for each polygon.

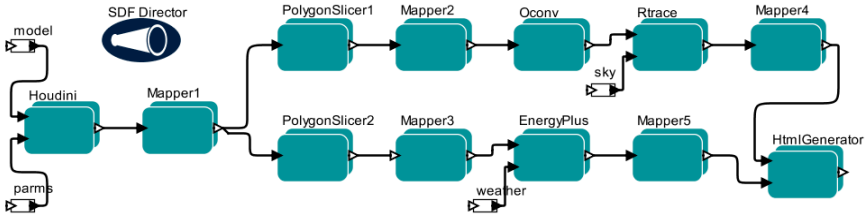
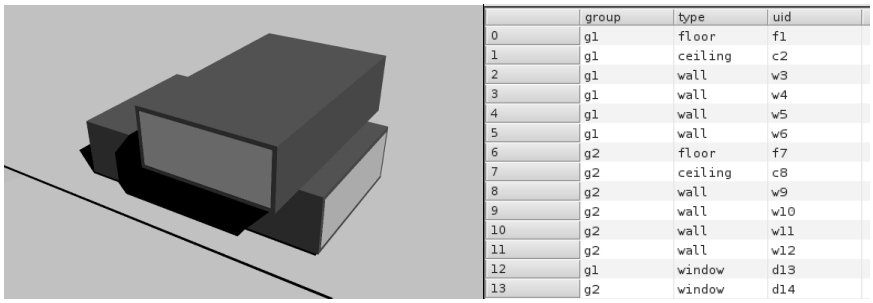



**Fig 4.** The Kepler workflow. See Figure 8 for the contents of the Mapper3 actor.



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

**Fig 5.** CAD model of 14 polygons, each with 3 attributes ('uid', 'type' & 'group').

### Workflow mappers

For all the mappers, property graphs are used as the mapping data model. For Houdini, EnergyPlus, and Radiance, parsers and serializers are created for stepping between the native file formats and the property graphs.

The workflow contains two scripted mappers and five composite mappers containing sub-networks of mapping actors. The two scripted mappers are actually two instances of the PolygonSlicer mapper, the scripted mapper performing the polygon slicing using Boolean operations. This has been implemented in a generalized way to be used for various slicing operations by users who do not have the necessary programming skills to implement such a mapper. In this case, Radiance and EnergyPlus require the polygons to be sliced in different ways. The PolygonSlicer mapper has a set of parameters that allows users to define the polygons to be sliced, and the Boolean operation to be performed. It also allows users to specify certain other constraints, such as the maximum number of points per polygon. Finally, it also identifies adjacency relationships between polygons, for example if two polygons are a mirror image of one another. The input and

output files for this mapper use a standard JSON property graph file format, and the parser and serializer using the existing GraphSON library.

The five composite mappers each contain sub-networks that perform a procedural query type of mapping operation. These sub-networks are defined using Kepler mapping actors that provide a set of basic functions for mapping graph vertices and graph edges. Users are able to build complex customized mapping networks by wiring together these basic mapping actors. Each mapping actor has various parameters that allow the mapping function to be customized. When these actors are executed, Gremlin mapping scripts are automatically generated based on these parameter settings.

Three highly generic mapping actors are defined that can be used to create a wide variety of mappings. The Merge actor is used for merging input graphs into a single output graph; the Spawn actor is used for adding new vertices to the input graph; and the Iterate actor is used for iteration over vertices and edges in the input graph while at the same time generating vertices and edges in the output graph. This last actor is powerful and very flexible, as it allows for copying, modifying, or deleting vertices and edges from the input to the output. 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.

### A mapping example

In order to understand the mapping process, the mapping from Houdini to EnergyPlus will be described in more detail. The first step is for the user to create 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 save it as a geometry data file (using Houdini's .geo format, though other formats could be used too, e.g., .obj).

Two mappers are then applied. The first mapper maps the output from Houdini to the input of the PolygonSlicer, and the second mapper maps the output of the PolygonSlicer to the input of EnergyPlus. For the first mapper, a parser is provided for stepping from the Houdini geometry file to the property graph, and a serializer is provided for stepping from the property graph to the JSON graph file. For the second mapper, a parser is provided for stepping from the JSON graph file to the property graph, and a serializer is provided for stepping from the property graph to an EnergyPlus input file (using EnergyPlus' .idf format). As already mentioned above, these parsers and serializers just mirror between the data file and the data model, and as a result they can be implemented in a way that is highly generic.

Although implementing these parsers and serializers will require someone with programming skills, it needs to be done only once, after which end-users can simply select the required parsers and serializers from a library. Implementing the parser and serializer for the JSON graph files is trivial since a library already exists. For Houdini and EnergyPlus, the ASCII data files have a clear and simple structure, resulting in a straightforward implementation for the parser and serializer.

Given a library of parsers and serializers, the task for the end-user is then reduced to the transformation of the input property graph into the output property graph using the three Kepler mapping actors. In anticipation of this mapping process, the user can define additional attributes in the geometry model that can be used during the mapping. 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. In this case, the polygons in the parametric model are each assigned three attributes: a 'uid' attribute is used to define a unique name for each polygon, a 'type' attribute is used to define the type for each polygon, and a 'group' attribute is used to define the group to which the surface belongs, with groups corresponding to zones (see Figure 5). When the parser reads the geometry data file, it will convert these attributes into properties, so that a polygon with attributes in the geometry file will become a graph vertex with properties in the property graph.

The user then needs to create the graph mappers using the graph mapping nodes. Figure 6 shows the overall structure of these input and output property graphs, and Figure 7 shows the properties associated with three of the vertices in each property graph. The PolygonSlicer and the EnergyPlus actors 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.

In the first mapping, where the output of Houdini is mapped to the input of the PolygonSlicer, two new vertices are added for the two groups, and edges are added from the new group vertices to the polygon vertices. These vertices and edges are created using an Iterate actor. 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.

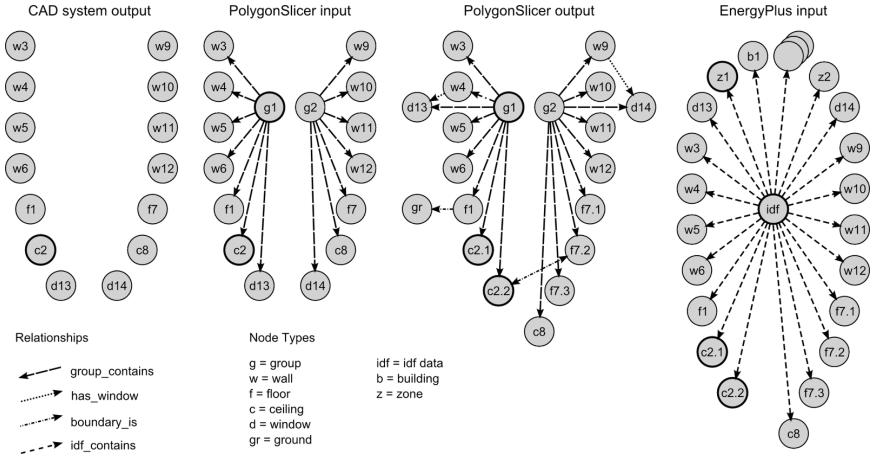



**Fig 6.** Simplified diagrammatic representation of the property graphs for. (Point data is not shown in order to reduce the complexity of the diagrams. Before the PolygonSlicer there are 24 points, while afterwards there are 28 points.)

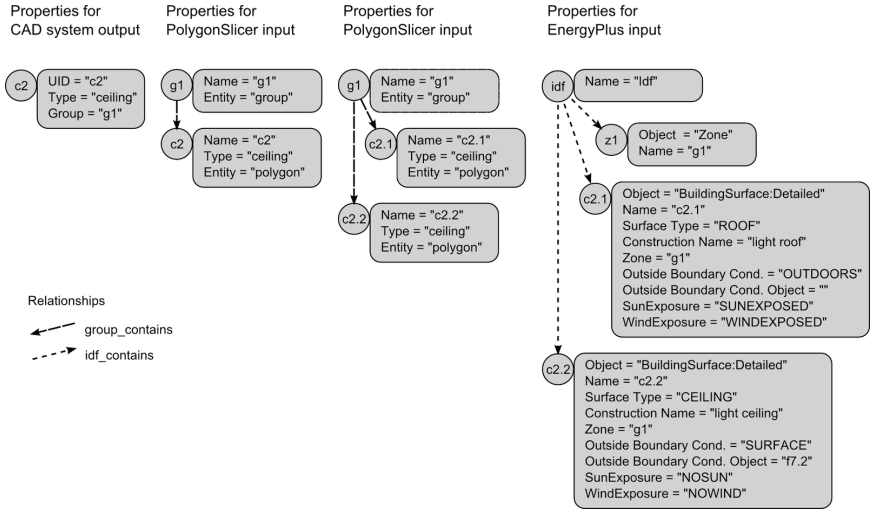



**Fig 7.** An example of the property data for a few of the vertices in the property graphs. Typically, the property graphs will undergo a process of information expansion, where data is gradually added to the model as needed.

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. The Kepler network for this mapper is shown in

Figure 8. The Spawn actor is used to create the additional vertices, and Iterate actors are used to copy and modify existing vertices. The groups are mapped to EnergyPlus zones, and the polygons to EnergyPlus surfaces. In the process of mapping, the Iterate actor 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 vertex. 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.

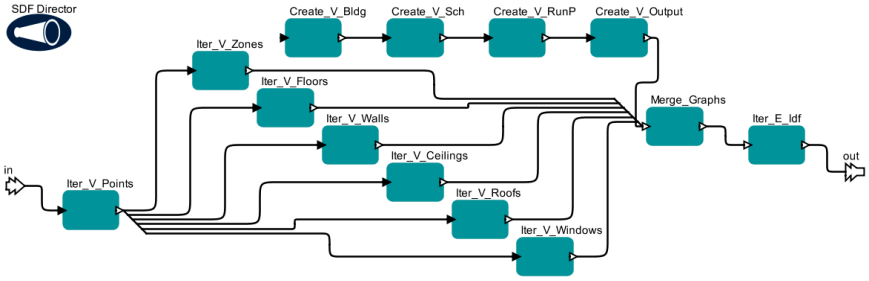



**Fig 8.** The Kepler mapper that maps the output of the PolygonSlicer actor to the input of the EnergyPlus actor. See the 'Mapper3' actor in Figure 5.

**Table 1** The parameter names and values for the Iter_V_Ceilings actor. 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** |
|---|---|
| Select | *g.V.has('Entity','polygon').has('Type','ceiling')*<br>*.as('result').out('boundary_is')*<br>*.has('Entity','polygon').back('result')* |
| Vertex properties | Object : 'BuildingSurface:Detailed'<br>Name : *x.Name*<br>Surface_Type : 'CEILING'<br>Construction_Name : 'light ceiling'<br>Zone : *x.in('group_contains').Name*<br>Outside_Boundary_Cond : 'SURFACE'<br>Outside_Boundary_Cond_Object :<br>*x.out('boundary_is').Name*<br>Sun_Exposure : 'NOSUN'<br>Wind_Exposure : 'NOWIND'<br>Points : *x.Points* |

For each different surface type, a separate Iterate actor is defined. For example, consider the 'Iter_V_Ceilings' actor in Figure 8. This node generates the ceilings of the two zones. Table 1 shows the two main parameters for the actor. 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 'Vertex properties' parameter then defines a set of name-value property pairs. For each polygon in the filtered input graph, the iterator actor will generate a vertex in the output graph with the specified properties.

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

### Adaptive-iterative exploration

Once a workflow has been developed, it can be used to iteratively explore design variants. The example workflow can be used without modification to evaluate any design variants that consist of a set of stacked zones. If other spatial configurations of zones need to be evaluated not limited to stacking, then the workflow may need to be adjusted in minor ways. For example, if zones are configurations so that they are adjacent to one another, then the 'Iter_V_Walls' actor in Figure 8 would be modified to allow common boundary walls between zones.

## Conclusions

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 common data models simplifies the process for end-users to develop customized mappings. 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 grow 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 visualize, interrogate and debug property graph data during the mapping process [24]. Schema management tools could let actor developers define formal graph schemas for input and output data for their actors. This could then let end-users identify and isolate subsets of large schemas relevant to their particular design scenario.

## Acknowledgments

The approach proposed in this research emerged out of discussions at the "Open Systems and Methods for Collaborative BEM (Building Environment Modeling)" workshops held at the CAAD Futures Conference in July 2011 and at the eCAADe Conference in 2013. We would like to thank the participants of these workshops for their contributions. Additionally, we would like to thank the reviewers for their valuable comments.

## References

1. Coons SA (1963) An outline of the requirements for a computer-aided design system. Proc. AFIPS. ACM. 299–304
2. Sutherland I (1963) SketchPad: A man-made graphical communication system. Proc. of the Spring Joint Computer Conf. Detroit. 329–346
3. Janssen P, Stouffs R, Chaszar A, Boeykens S, Toth B (2012) Data Transformations in Custom Digital Workflows: Property Graphs as a Data Model for User-Defined Mappings. Proceedings of the Intelligent Computing in Engineering Conference ICE2012, Munich
4. Toth B, Janssen P, Stouffs R, Boeykens S, and Chaszar A (2013) Custom digital workflows: A new framework for design analysis integration. International Journal of Architectural Computing 10(4):481–500.
5. Deelman E, Gannon D, Shields M, Taylor I (2008) Workflows and e-science: An overview of workflow system features and capabilities. Future Generation Computer Systems 25:528–540
6. Yu J, Buyya, R (2005) A taxonomy of scientific workflow systems for grid computing. SIGMOD Record 34:44–49
7. Curcin V, Ghanem M (2008) Scientific workflow systems - can one size fit all? CIBEC 2008. Cairo. 1–9

8. McPhillips T, Bowers S, Zinn D, Ludaescher B (2009) Scientific workflow design for mere mortals. Future Generation Computer Systems 25:541–551
9. Janssen P, Chen KW (2011) Visual Dataflow Modelling: A Comparison of Three Systems. Proceedings of the 14th International Conference on Computer Aided Architectural Design Futures, Liege, Belgium. 801–816
10. Bowers S, Ludäscher B (2005) Actor-oriented design of scientific workflows. Proc. of the Int. Conf. on Conceptual Modeling ER. 369–384
11. Mirtschin J (2011). Engaging Generative BIM Workflows. Proceedings of LSAA 2011 Conference Collaborative Design of Lightweight Structures, Sidney
12. Tsichritzis DC, Lochovsky FH (1982) Data Models. Prentice-Hall
13. Eastman CM, Teichholz P, Sacks R, Liston K (2011) BIM Handbook - A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors. 2nd ed. John Wiley, Hoboken
14. Kilian A (2011) Design innovation through constraint modeling. International Journal of Architectural Computing 4: 87–105
15. Altintas I (2011) Collaborative Provenance for Workflow-Driven Science and Engineering. PhD Thesis. University of Amsterdam, Amsterdam
16. Bos P (2012) Collaborative engineering with IFC : new insights and technology. In Gudnason G, Scherer R (eds) eWork and eBusiness in Architecture, Engineering and Construction, 9th ECPPM Conference Proceedings. 811–818
17. O'Donnell J, See R, Rose C, Maile T, Bazjanac V, and Haves P (2011) SimModel: A Domain Data Model for Whole Building Energy Simulation. Proceedings of the 12th Conference of International Building Performance Simulation Association, Sydney
18. Srinivasan, R, Kibert C, Thakur S, Ahmed I, Fishwick P, Ezzell Z, Lakshmanan J (2012) Preliminary Research in Dynamic-BIM (D-BIM) Workbench Development. Proceedings of the 2012 Winter Simulation Conference (WSC). 1–12
19. Kannengiesser U (2007) Agent-based interoperability without product model standards. Computer-Aided Civil and Infrastructure Engineering, 22(2):96–114
20. Bellahsene Z, Bonifati A, Duchateau F, Velegrakis Y (2011) On Evaluating Schema Matching and Mapping. In Bellahsene Z, Bonifati A, Rahm E (eds) Schema Matching and Mapping. Springer, Berlin and Heidelberg, 253–291
21. Gremlin (2014) Retrieved 1st April 2014 from https://github.com/tinkerpop/gremlin/wiki
22. Ludascher B, Altintas I, Berkley C, Higgins D, Jaeger E, Jones M, Lee A, Tao J, Zhao Y (2006) Scientific workflow management and the Kepler system. Concurrency and Computation: Practice & Experience 18(10), 1039–1065
23. Eker J, Janneck J, Lee EA, Liu J, Liu X, Ludvig J, Sachs S, Xiong Y (2003) Taming heterogeneity: the Ptolemy approach. Proceedings of the IEEE, 91(1):127–144
24. 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