Session 13

Utilising Dynamo "Beyond" Computational Design

Andreas Dieckmann, Chair for Computer Aided Architectural Design, RWTH Aachen University

Class Description

Dynamo, the visual programming environment for the Revit platform, is marketed by Autodesk mainly as a computational design tool. Not interested in computational design? That doesn't mean that Dynamo is not for you because it really is so much more – a swiss army knife for automating all sorts of complex workflows as well as everyday menial tasks in Revit.

This class will provide an overview of how to use Dynamo for tasks other than computational design with a special focus on model checking and related operations. More specifically, we will look at the following topics:

- Creating model checking routines in Dynamo, ranging from simple office standard checks to more complex code compliance checks
- Methods of visualising and exporting model check results with Dynamo
- Tying model check results to actual tasks in Dynamo, i.e. fixing things based on check results

The class will be taught from the perspective of an architect with a programming background but no formal education in programming. Thus it will also focus on aspects of accessibility for non-programmers such as:

1. Dynamo's modular structure and how it facilitates changing parts of a workflow or checking routine
2. Using Python and the Revit API to create missing functionality in Dynamo

This class is targeted at Revit users that are interested in automating Revit but shy away from full-on programming.

About the Speaker:

Andreas Dieckmann has been working with CAD & BIM applications since the mid-nineties. He began using Revit in 2007 and since then has never wanted to do another project without it. Andreas teaches various classes on Building Information Modelling and parametric design at RWTH Aachen University[1] / Germany where he has been working as a teaching / research fellow for the chair of Computer Aided Architectural Design (CAAD)[2] for about a decade now. Since 2011, he has also been teaching BIM and parametric design workshops at conferences and other universities. Andreas is one of the founding partners of the Aachen-based architectural office IP arch[3], which completed its first major projects in 2010. When he is not busy working on his PhD thesis, he regularly contributes new packages to the Dynamo community.

1 http://www.rwth-aachen.de
2 http://caad.arch.rwth-aachen.de
3 http://www.ip-arch.de

Content

Utilising Dynamo "Beyond" Computational Design.....1
.....*Class Description*.....1
.....*About the Speaker:*.....2
Introduction.....4
.....*On Content Structure*.....4
.....*On Dynamo Versions*.....4
.....*On Custom Nodes*.....4
.....*Dynamo "Beyond" This And That (a.k.a. The Actual Introduction)*.....5
Extending Dynamo With Python And The Revit API.....6
.....*Why Coding In Dynamo Is Fairly Easy*.....6
.....*Meet Your New Best Friend, The Revit API*.....6
.....*Python Nodes – A Programming Environment Within A Programming Environment*
.....8
.....*The Anatomy Of A Python Node*.....9
.....*Additional Aspects Of Creating Python Nodes*.....12
Element Collections.....14
.....*OOTB Selection Scope*.....14
.....*Selection Scope Beyond OOTB*.....16
Object Relationships.....17
.....*Formal Relationships*.....17
.....*Informal Relationships*.....18
Managing And Checking Collections.....20
.....*Applying Conditions*.....20
.....*Sorting Collections*.....21
.....*Grouping Collections*.....23
.....*Filtering Collections*.....23
Managing Results.....25
.....*Examining Results Inside Dynamo*.....25
.....*Visualising Results*.....25
.....*Writing Results To The Model*.....26
.....*Exporting Results*.....27
.....*Fixing Things*.....27
Case Studies.....29
.....*Natural Lighting*.....29
.....*Egress Paths*.....37
A Rather Lengthy Conclusion.....43
.....*Further Reading*.....45
.....*Data Sets*.....45

Introduction

On Content Structure

One of the central points this document is trying to make is that extending Dynamo's capabilities with Python is a worthwhile endeavour. Since that is probably also the techiest part, we will start with that to get it over with. After that, we're going to go through the different techniques necessary to create a model checking routine in Dynamo. Lastly, we're going to have a look at a few case studies to examine how those techniques can be brought together in a more complex graph.

On Dynamo Versions

At the time of writing, Dynamo 0.7.2 has been available for just a few days. While Dynamo 0.7.x represents a giant step forward for the future of the Dynamo engine and offers a much broader set of out-of-the-box functionality, the current trade-off is, unfortunately, stability. Many of the workflows described in this document as well as a significant number of the custom nodes referenced herein could not be migrated to Dynamo 0.7.2 yet. In most if not all cases this is caused by two known bugs, namely #1698 ("Custom nodes return Null")[4] and #2401 ("Built-in categories not unwrappable")[5] which will surely be resolved eventually. Until then, Dynamo 0.6.3 can serve as a stable production environment for some of the workflows described below – unless you have already made the switch to Revit 2015. In that case, you're stuck with Dynamo 0.7.x, for better and worse.

On Custom Nodes

All custom nodes mentioned in this document are part of the package **Clockwork**[6] unless otherwise stated. Clockwork collects all the 200+ custom nodes in a single package that I had previously published in various packages. My reasoning is that this will make updates easier and reduce package dependencies. I had never set out to build so many custom nodes – somehow it just happened (most of them are by-products of my research and

4 https://github.com/DynamoDS/Dynamo/issues/1698
5 https://github.com/DynamoDS/Dynamo/issues/2401
6 Clockwork should be available via Dynamo's package manager by the end of October 2014. Find documentation (based on the 0.6.3 version) for what each node does here: https://github.com/CAAD-RWTH/ClockworkForDynamo/tree/master/package_samples (Documentation for 0.7.x under way)

teaching activities). At some point it just got out of hand and unmanageable, hence the recent aggregation into a single package. The irony is that not long ago I actually encouraged people to publish more packages[7] – oh well.

Dynamo "Beyond" This And That (a.k.a. The Actual Introduction)

Looking at the OOTB functionality available in Dynamo right now, it should be quite clear that the majority of its functionality is still directed towards computational design. A year ago, this was even more true. Up until January 2014, there wasn't even an OOTB way of selecting elements based on their component category – one of the major preconditions for browsing through the content of a BIM model.

So, how does that matter? It doesn't. One of the many beauties of Dynamo is its extensibility. You can use **Python** nodes (and more recently since 0.7.1: **Zero Touch DLL import**) to add your own functionality. Using Dynamo means going beyond OOTB functionality in Revit. Making custom functionality is just another step - it's going beyond OOTB functionality in Dynamo.

Dynamo is a swiss army knife. It has all sorts of tricks up its sleeve – when you least expect it you will discover another useful piece of functionality in the node browser. And it's a very flexible tool. You can bend it (and Revit, too, while you're at it) to your will through the power of coding, beyond computational design, beyond OOTB functionality, beyond this and that.

7 http://dynamobim.org/sharing-is-caring-feeding-the-package-manager/

Extending Dynamo With Python And The Revit API

Why Coding In Dynamo Is Fairly Easy

There seem to be a lot of “real” programmers out there that turn up their nose at visual programming. Just the other day I overheard a colleague from another faculty that shall remain unnamed say “Let the architects use visual programming, we'll do the *real* coding”. In my opinion, that kind of talk is an indicator for how well visual programming actually works: it's a shortcut to programming and apparently some people consider that unfair.

So why is coding in Dynamo so easy, even for beginners and non-programmers? Here are some reasons that come to mind – take your pick:

- You don't have to worry about **remembering the correct syntax** of a command – just drop a node onto the canvas. It'll (usually) have a description and you can immediately see what goes into it and what comes out.
- You don't actually have to **remember the name of a command** either. The node browser is structured rather well, so finding a particular node usually is a quick process.
- It's easy to **follow the data flow** of a Dynamo graph because you just have to follow the node connections one by one.
- Using the **preview bubbles** (that are available for most nodes) or the **Watch node** can help immensely when building a graph step-by-step.
- A lot of **workflows are often quite similar** making large parts of already existing workflows reusable.
- **Creating custom nodes** is a simple process and so is sharing them. No need to reinvent the wheel every time you start Dynamo.

Meet Your New Best Friend, The Revit API

If you want to use Dynamo to learn something about your Revit model or automate a task in Revit and there is no built-in node in Dynamo that will do just that, it's time to get yourself acquainted with the Revit API (“**A**pplication **P**rogramming **I**nterface”). The API allows you to query and alter a Revit model and follows the object-oriented programming[8] paradigm: Any object in a Revit model usually has some properties (i.e. readable attributes like name, level, height etc.) as well as some methods (i.e. things you can do with that object like e.g. mirror it).

8 http://en.wikipedia.org/wiki/Object-oriented_programming

In terms of resources, you won't need much. The **Revit SDK** ("Software Development Kit") ships with each version of Revit and updated versions are available online through ADN ("Autodesk Developer Network")[9]. The SDK contains code samples and, more importantly, the full API documentation (look for a Windows help file called **RevitAPI.chm** in the folder you extracted the SDK to).[10] Apart from the SDK, for me personally, the three most valuable resources for understanding how to work with the Revit API were Nathan Miller's **Revit API Notebook**[11] wiki, Jeremy Tammik's **The Building Coder**[12] blog as well as Harry Mattison's **Boost Your BIM**[13] blog.

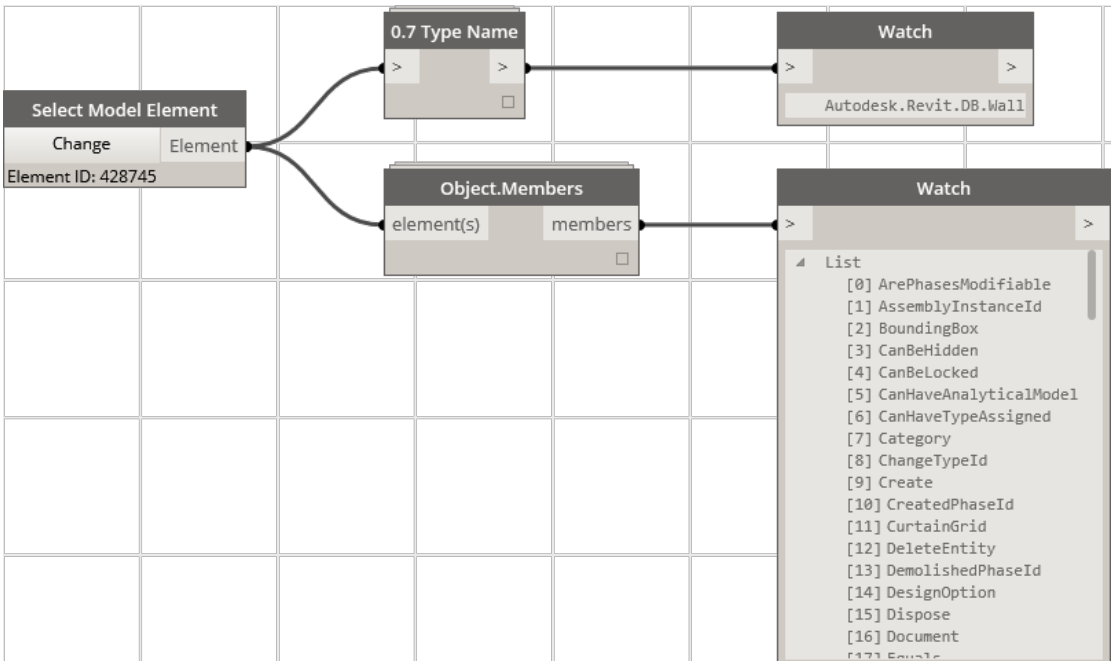

Fig. 1: Querying a Revit wall object with custom nodes

The most important thing of all, though, is curiosity. Using a tool like Jeremy Tammik's **RevitLookup**[14], you can examine any object in a Revit model and find out about its properties and parameters. Another way is to do this in

9 http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=2484975

10 Alternatively, you can use an online version here: http://revitapisearch.com/ (Currently only for the Revit 2014 API. Trivia: This website was developed by Peter Boyer who is part of the Dynamo developer team.)

11 http://wiki.theprovingground.org/revit-api

12 http://thebuildingcoder.typepad.com

13 http://boostyourbim.wordpress.com

14 https://github.com/jeremytammik/RevitLookup (a compiled version can be found here: http://revitcoaster.blogspot.de/2014/06/revit-lookup-2015-addin.html)

Dynamo directly (see Fig. 1). Use the custom node **Object.Members**[15] to display all properties and methods of an object.[16]

Python Nodes – A Programming Environment Within A Programming Environment

If you want to access the Revit API in Dynamo and you're not a hardcore coder, you will have to use a Python node for that. Python nodes provide a separate (script-based) programming environment within Dynamo.

Most Python aficionados will probably tell you that one of the nicest things about Python is its simplicity. I suppose they are right. Python[17] is very easy to learn in terms of syntax and code structure. It's quite an ideal language for beginners, I think. Dynamo uses **IronPython**[18] which allows Python to access the .NET[19] framework of Windows which in turn will grant you access to the Revit API. There are numerous resources[20] to help you learn Python, so it's not covered in this document. For more Revit-specific examples refer to Nathan Miller's Revit API Notebook mentioned above (see ““) or take a look inside any of the Revit-specific nodes in package Clockwork. You can also find some info about Python nodes in the official Dynamo learning material.[21]

Working with Python nodes is fairly easy because the operations to be performed are usually not all that complicated. Dynamo provides a lot of infrastructure in terms of list management and geometric operations, so in most cases a Python node will merely have to perform the minor (albeit

15 Previously called **“Get Members”**

16 Alternatively you can get an object's data type using either package **“Type Name”** by Stephen Elliott (in Dynamo 0.6.3) or **“Get Type Name”** by Julien Benoit (in Dynamo 0.7.x) and then look up that data type in the Revit API documentation mentioned above. Note that there is also a package called **“Revit API Tools”** available (albeit apparently only for Dynamo 0.6.3).

17 https://www.python.org

18 http://ironpython.net

19 http://www.microsoft.com/net

20 Some good starting points are these sites:
http://www.learnpython.org
https://www.python.org/about/gettingstarted/
https://wiki.python.org/moin/BeginnersGuide

21 For Dynamo 0.6.3: https://www.youtube.com/watch?v=NaRWKiVrO48 and “Dynamo: Visual Programming for Design”, pp. 46-49 (https://github.com/DynamoDS/Dynamo/raw/master/doc/distrib/Dynamo%20Visual%20Programming%20for%20Design.pdf)
For Dynamo 0.7.x: https://www.youtube.com/watch?v=iTWWnpURVEw

essential) task of communicating with the Revit API. You can build the rest of the logic of your script around that using built-in Dynamo nodes.
In my experience, there are mainly four typical use cases when accessing the Revit API via Python in Dynamo known in basic database theory as **CRUD**[22]:

- **C**reate new elements in the model (e.g. draw a line, place a family etc.)
- **R**ead element properties (e.g. get the level of an element, find windows within a wall etc.)
- **U**pdate element properties (e.g. change the height or name of an element etc.)
- **D**elete an element from the model[23]

As long as you are only reading data from the model, things are fairly easy. However, any action that will result in changes to the model (i.e. creating, updating and deleting) will require what is referred to as a transaction[24] to avoid model inconsistencies (basically like checking in your changes to the central file when working with workshared files in Revit).

The Anatomy Of A Python Node

Let's take a look at a typical Python node that reads data from the Revit API. The code shown below is of the custom node **Element.Host**[25] that takes a list of objects and returns for each object the element it is hosted in (e.g. the wall a windows was placed in) or an empty list (if it doesn't have a host object). It was written for Dynamo 0.7.x.[26]

```
### PART 1 – IMPORT REFERENCES ###
# Import the CLR (Common Language Runtime) of the .NET framework.
# This will allow us to reference other program libraries.
import clr
```

22 http://en.wikipedia.org/wiki/Create,_read,_update_and_delete
23 Use package **"Eraser By Name"** by Julien Benoit for deleting elements from the model
24 http://en.wikipedia.org/wiki/Database_transaction
25 Previously called **"Get Host"**
26 Note that Python nodes in Dynamo 0.6.3 work slightly different. For a direct comparison have a look at the same custom node in the 0.6.3 version of Clockwork. Using Python nodes in 0.7.x is a bit more complicated than in 0.6.3 but all the important differences are addressed step-by-step in this migration guide: https://github.com/DynamoDS/Dynamo/wiki/Python-0.6.3-to-0.7.x-Migration

```
# Add a reference to the Revit API.
# This will allow us to access properties and methods of Revit elements.
clr.AddReference('RevitAPI')
from Autodesk.Revit.DB import *
# This part is necessary for importing Dynamo's ToDSType(bool) method.
# We will need that method later to “re-wrap” any Revit elements
# before returning them to Dynamo.[27]
clr.AddReference("RevitNodes")
import Revit
clr.ImportExtensions(Revit.Elements)
# Depending on what you're doing,
# you might need to reference other libraries here as well.

### PART 2 – DECLARE VARIABLES ###
# In the next line, we're declaring the variable “faminsts”
# and assigning the content of the node's first input to it.
# There are a couple of things worth explaining about this.
# All inputs of the Python node get fed into the list IN.
# They can be accessed by their respective index
# (i.e. IN[0] contains the first input of the Python node).[28]
# Dynamo's UnwrapElement() function will remove the wrapper around the
# actual Revit element and allow us to access it via the Revit API.[29]
faminsts = UnwrapElement(IN[0])
```

27 Wrapping / unwrapping is one of the new concepts introduced to Python nodes with Dynamo 0.7.x that requires incoming Revit elements to be unwrapped and outgoing elements to be wrapped. Essentially, you'll need to wrap all elements you created in a Python node with ToDSType(False) and everything else with ToDSType(True). Read more about it on the Dynamo Wiki on GitHub (https://github.com/DynamoDS/Dynamo/wiki/Python-0.6.3-to-0.7.x-Migration#elements). Note that at present neither are all Revit element types unwrapped correctly (https://github.com/DynamoDS/Dynamo/issues/2401), nor can all types be wrapped (https://github.com/DynamoDS/Dynamo/issues/2460)..

28 Two subtle differences in Dynamo 0.6.3:
1. There are two types of Python nodes (single input and multiple inputs).
2. In single input nodes the input is stored in the variable IN. In nodes with multiple inputs, the respective inputs can be accessed via the variables IN0, IN1,IN2 etc.
https://github.com/DynamoDS/Dynamo/wiki/Python-0.6.3-to-0.7.x-Migration#input-variables

29 When working with Dynamo 0.7.x, make sure to use Dynamo 0.7.2 or later. UnwrapElement() can not process lists of elements in previous versions.

```
# Declare the variable “elementlist” as an empty list.
# We're going to use this list to collect the host elements.
elementlist = list()
### PART 3 – CYCLE THROUGH INPUT LIST ###
# Now cycle through each element in our input list “faminsts” with a “for” loop.
# While cycling through the list, each element in the list will be made
# available through the variable “item”.
# In Python - unlike in a lot of other programming languages - control
# structures like loops or conditions are indicated through indentation.
# The only delimiting element you'll need is the colon.
for item in faminsts:
        # Try/except is a control structure that lets us define:
        # 1) A default behaviour of the script (“try”).
        # 2) What should be done if the default command can not be executed
        # properly (“except”).
        # This device allows us to catch possible exceptions
        # and keep our script running.
        try:
                # There's a lot happening in one line here.
                # It's a good example of how concise your code can be in
                # object -oriented programming, but maybe not very readable.
                # Let's break this down into three smaller steps:
                “”””
                elem_hosts = item.Host
                elem_hosts_wrapped =  elem_hosts.ToDSType(True)
                elementlist.append(elem_hosts_wrapped)
                “”””
                # 1) Retrieve the host element of each item in the list.
                # Note that this is actually the only piece of code where we
                # interact with the Revit API (the “Host” property of an object).
                # 2) Use Dynamo's ToDSType() method to wrap the host
                # element for downstream use in Dynamo.
                # Note the difference of notation when calling a property of
                # an object – obj.Property – and a method of an object –
                # obj.Method(args).
                # 3) Add the wrapped host element to the element list we
                # declared earlier.
                elementlist.append(item.Host.ToDSType(True))
```

```
        except:
                # if we encounter an exception, we just add an empty list to our
                # elementlist.
                # This is to ensure that there are as many elements in the
                # output list as there are in the input list.
                # Downstream, this might be important for filtering elements.
                elementlist.append(list())

### PART 4 – WRITE RESULTS TO OUTPUT ###
# Finally: declare the output variable. It is always called OUT.
OUT = elementlist
```

Additional Aspects Of Creating Python Nodes

As described above (see “”), if you want your Python node to make changes to the Revit model (i.e. create, update, delete), you will need to wrap the portion of the code that actually exerts the change in a **transaction**.[30] In Dynamo 0.6.3 this was done by placing a built-in Transaction node *behind the Python node* while in Dynamo 0.7.x the transaction has to be *a part of your Python code*.[31]

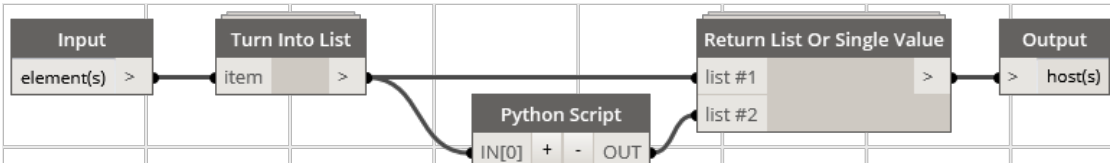

Fig. 2: Typical setup of my Revit-specific custom nodes: the input is converted into a flat list, then gets passed to the actual Python node, the output of which is subsequently returned either as a list or single value (depending on the original input).

Consider turning Python nodes into **custom nodes**. In my opinion, it has several advantages:

- Better documentation: Custom nodes have a name, description and category.
- Better reusability: They can be found in the node browser using the search tool.

30 If that portion of the code is nested inside a loop or condition it makes sense to wrap the entire structure into a transaction.

31 Note that transactions seem to work more smoothly in Dynamo 0.7.x – in 0.6.3 you sometimes need to place additional Transaction nodes in other portions of your graph as well to make it execute properly. Creating transactions inside Python nodes is documented on the GitHub wiki (https://github.com/DynamoDS/Dynamo/wiki/Python-0.6.3-to-0.7.x-Migration#transactions).

- They can be shared using the package manager.

Most importantly, you can add additional recurring functionality around the Python node and avoid having to code the same functionality into each Python node you make (see Fig. 2). My personal rule of thumb is to only use Python nodes for operations that in Dynamo would either be very difficult (or would need a lot of nodes) or impossible (i.e. requiring access to the Revit API). Everything else gets done in Dynamo. A good example for that is data conversion. It's good practice to do all the conversions that are readily available directly in the Python node as they are generally not available as Dynamo nodes.[32] However, for some Revit data types, there are no conversion functions (e.g. UV coordinates or colours),[33] so you might have to build something like that yourself. It would then make sense to build it as a separate custom node in order to make it reusable.

32 See the GitHub wiki for more details on available methods: https://github.com/DynamoDS/Dynamo/wiki/Python-0.6.3-to-0.7.x-Migration#geometryobjects

33 Use the custom nodes **RevitUV.ToDynamoUV** and **RevitColor.ToDynamoColor** for that (only in Clockwork for Dynamo 0.7.x)

Element Collections

Most operations on a model start with selecting the appropriate elements. While there are nodes available in Dynamo for the direct (i.e. manual) selection of elements (e.g. "Select Model Elements"), reviewing or checking a model usually requires the selection of elements based on element properties rather than manual selection. Generally, there are three types of entities available for selection: families, family/element types (often also referred to as family symbols, especially in the Revit API documentation) and family/element instances, plus some other things which we'll graciously ignore in the context of this document. In the context of a model review families mostly seem to be useful as a means of organising (i.e. grouping) family types while family types and instances provide the actual model information.

Operations that gather elements from the model in such a way are referred to as **FilteredElementCollectors** in the Revit API.[34] FilteredElementCollectors provide a lot of different methods of acquiring pre-filtered collections of elements from the model which can be applied in Dynamo using Python nodes and the Revit API.

OOTB Selection Scope

Let's have a look at the collections we can build with OOTB functionality in Dynamo. In many if not most cases the scope of the selection is at first determined by using Revit's built-in component categories as a filtering device. To that end, the **Categories** drop-down was introduced in Dynamo 0.6.3[35] – before that, custom nodes had to be used for individual categories[36]. The most typical downstream use for categories is the **All Elements of Category** node that will return family/element instances based on category.[37] Another commonly used approach is the **Element Types** drop-down in conjunction with the **All Elements Of Type** node. This method ignores the

34 http://revitapisearch.com/html/263cf06b-98be-6f91-c4da-fb47d01688f3.htm

35 Dynamo 0.6.3 was released in March 2014, but the Categories node was actually already present in the daily builds since January 2014. This is actually a great example of how responsive the Dynamo development team is and how they listen to the users as this functionality is based on a feature request (https://github.com/DynamoDS/Dynamo/issues/790) and was implemented within two months' time of the initial request.

36 Package **"Select By Category"** up to version 0.3.11.

37 Note that in Dynamo 0.6.3 All Elements Of Category will not only return instances, but also types. If you only want to retrieve instances, use the custom node **"Get Family Instances By Category"** instead.

component categories and instead builds a collection based on element types. The subtle differences may in some cases not be very obvious but the following is a good example (also see Fig. 3):

A collection of the category “Floors” will include all elements of the floor category (i.e. floors that were modelled using the floor slab tool *as well as* floors created as in-place families). A collection of the element type “Floor”, on the other hand, will give us all elements created as a floor slab regardless of their category (i.e. floor slabs *and* foundation slabs).

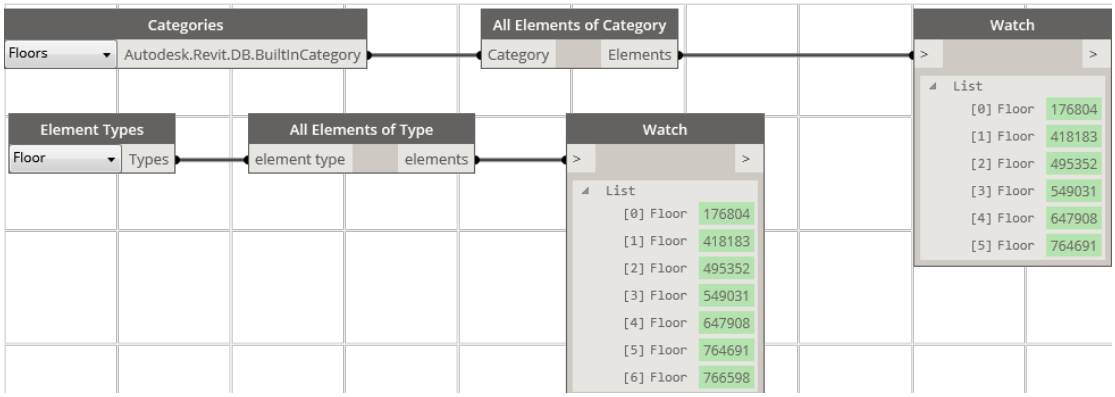

Fig. 3: The subtle differences of building collections based on category and element type

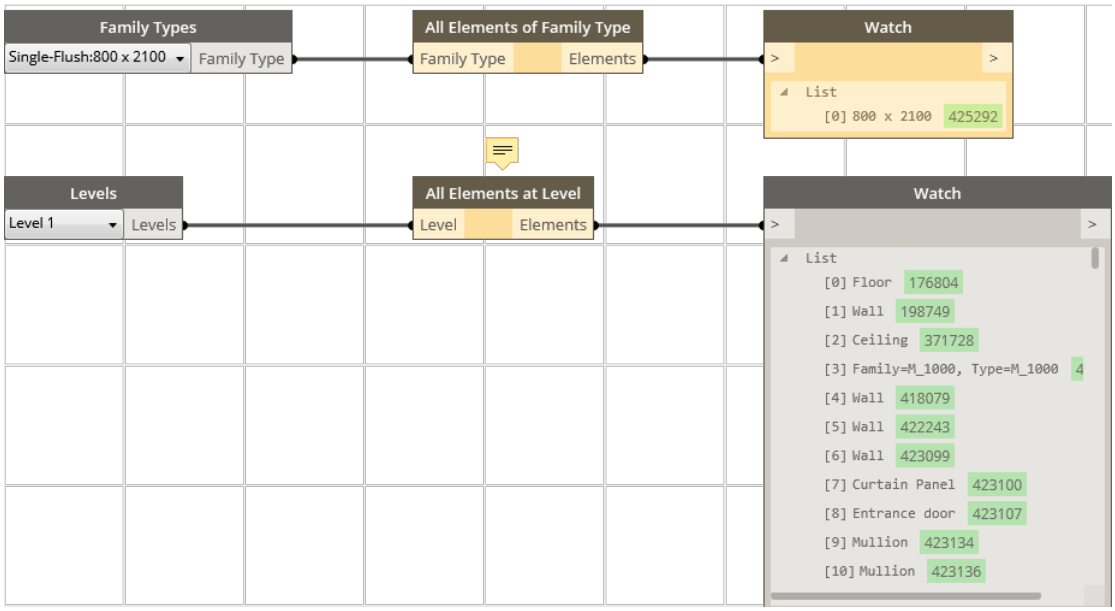

Fig. 4: More OOTB selection methods

You can also use All Elements Of Type to retrieve a collection of family types, but be aware that this will only work for system families (per category, e.g. “WallType”) or *all* family types regardless of category (using “FamilySymbol”).

Other OOTB options to retrieve collections are by using **All Elements At Level** or **All Elements Of Family Type** (see Fig. 4).

Selection Scope Beyond OOTB

Since FilteredElementCollectors can do a lot of useful things, they can be used to provide some additional functionality within Dynamo. Again, using Python nodes, this is fairly easy to accomplish. Here are some examples (also see Fig. 5) that illustrate some of the possible uses for FilteredElementCollectors[38]:

- **Get Families By Category**: Retrieve a collection of all families of a given category from the model.
- **Get Family Symbols By Category**: Retrieve a collection of all family types of a given category from the model.
- **Get View-Dependent Family Instances By Category**: Retrieve a collection of all view-dependent family instances of a given category from the model (e.g. dimensions).

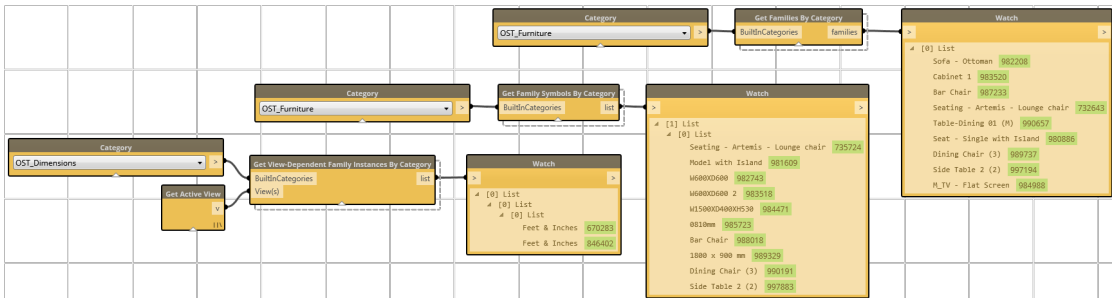

Fig. 5: Some examples of custom-built element collections

38 All of these are part of Clockwork for Dynamo 0.6.3 – have a look at the Python code within these nodes to see how to build a FilteredElementCollector yourself. As soon as issue #2401 (already mentioned above) is resolved, these nodes will make it into Clockwork for Dynamo 0.7.x, too. Jeremy Tammik provides a great overview of use cases on his blog: http://thebuildingcoder.typepad.com/blog/2010/12/filtered-element-collector-sample-overview.html

Object Relationships

When reviewing a model, gathering a simple collection of elements will not always be precise enough. The more specific the question the more likely it is that we will need to follow a chain of relationships between elements. As an example, consider the following scenario: You want to find all exterior doors on the ground floor of a building. There is no OOTB way of doing this, so it would have to be done by examining the relationships between objects.

Formal Relationships

A Revit model is essentially a relational database of building elements. Using the Revit API, we can retrieve those relationships from the model. A simple example would be the retrieval of the level of a wall. Dynamo provides a few nodes that make element relationships accessible, e.g. the **Sheet.Views** node that will return all views placed on a given sheet. Here are some more examples of custom nodes retrieving element relationships:

- Finding inserts (e.g. doors and windows that are hosted in walls, see Fig. 6) or hosts (e.g. the wall a given window or door is hosted in, see Fig. 7)[39]
- Finding the bounding elements of a room (see Fig. 8) or finding rooms by the doors and/or windows that lead to them (see Fig. 9)[40]

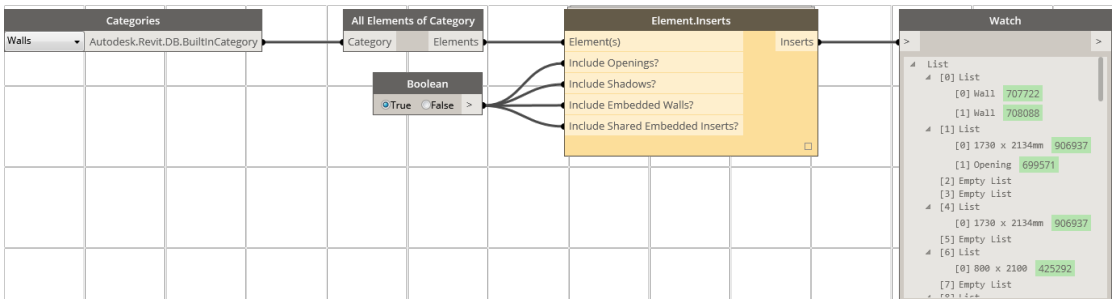

Fig. 6: Finding inserts in walls

39 Using custom nodes **Element.Inserts** (formerly **"Get Inserts"**) and **Element.Host** (formerly **"Get Host"**)

40 Using custom nodes **"Get Room Boundaries"** and **"Rooms From Doors And Windows"** - currently not part of Clockwork for Dynamo 0.7.x due to issue #1698

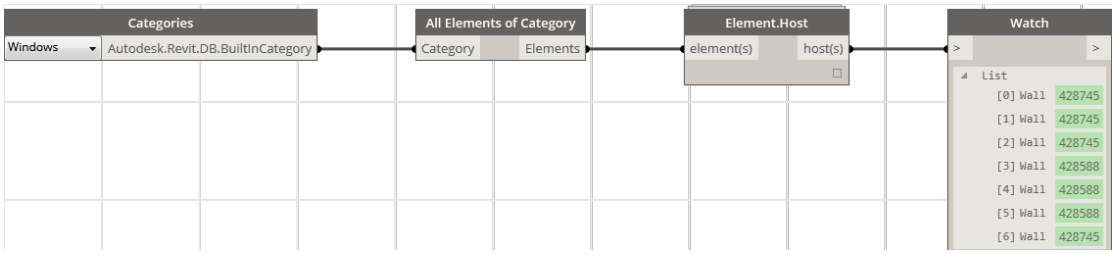

Fig. 7: Finding the walls a collection of windows is hosted in

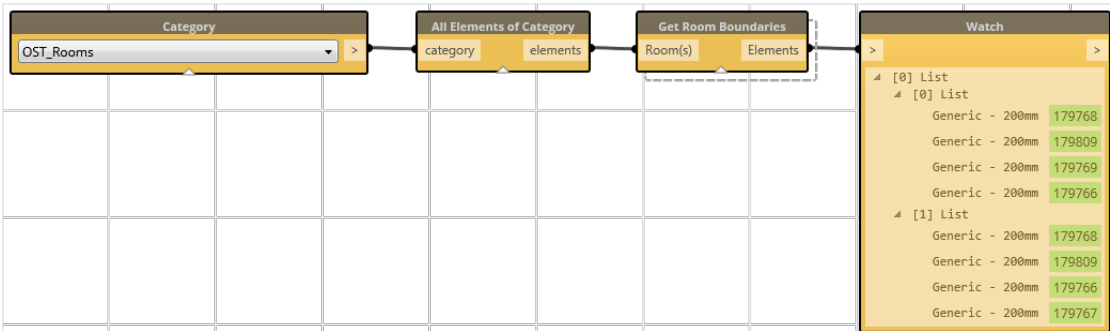

Fig. 8: Getting the bounding elements of a list of rooms

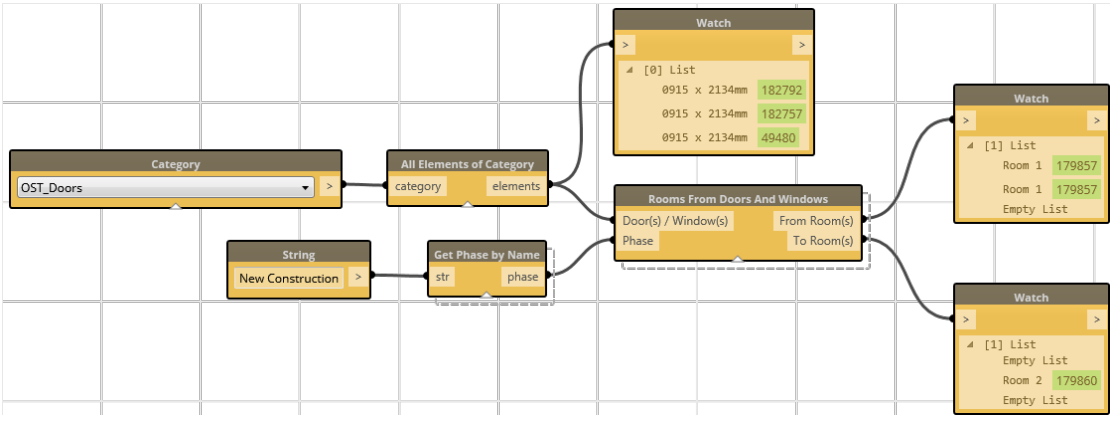

Fig. 9: Getting rooms from doors

Informal Relationships

Besides formal object relationships, informal relationships exist in the model, too, i.e. relationships that cannot be directly retrieved from the Revit API in terms of object properties that link to other objects. We can use other methods (some available in the Revit API, some in Dynamo itself) to find out about topics like e.g. spatial relationships between objects. One example is the **Reference Intersector By Category** custom node (see Fig. 10). It retrieves a collection of all family instances of a given category from the model

that are intersected by a given vector, a great tool for line-of-sight related problems.[41]

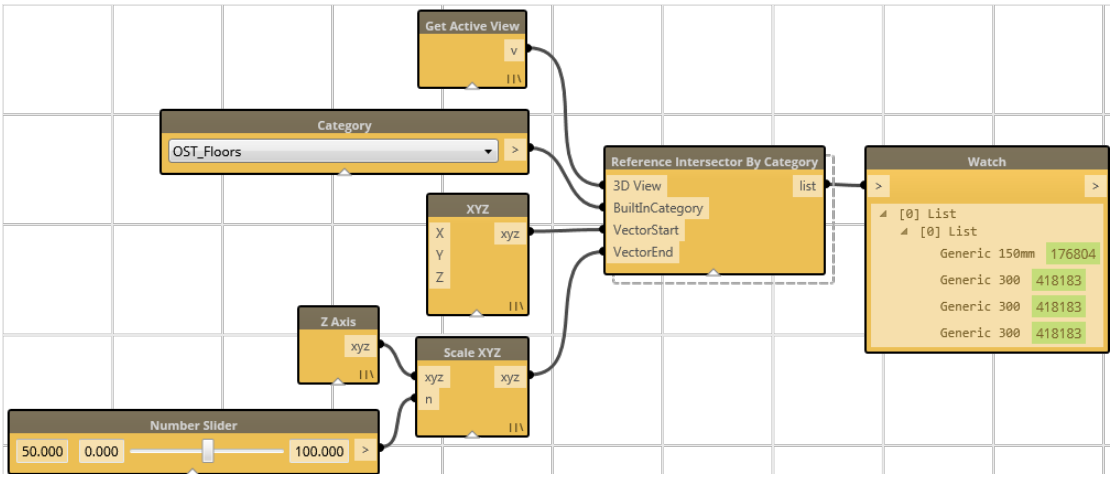

Fig. 10: A reference intersector in action, finding all floor slabs intersecting the project Z axis

41 Currently only in Clockwork for Dynamo 0.6.3 until issue #2401 is resolved. This uses another method available in the Revit API, namely the ReferenceIntersector: http://revitapisearch.com/html/36f82b40-1065-2305-e260-18fc618e756f.htm

Managing And Checking Collections

Having acquired a collection of elements from the model, the next step usually is to further manage that selection. The three most typical use cases probably are sorting, grouping and, most importantly, filtering. When checking a model, these actions are in many if not most cases based on certain conditions we are interested in, i.e. does an element satisfy certain criteria. An example would be a filter that only retrieves walls that have a certain thickness. The key to all of the techniques below is, of course, that the information in the model is correct – but if you are using Revit for more than just generating 3D geometry this should be a given.

Applying Conditions

When using Dynamo for checking or reviewing Revit models, conditions basically form the rules we check the model against. There are several ways of working with conditions in Dynamo. What they all have in common is their output: Conditions will always return boolean values (True/False).[42] Conditional statements (see Fig. 11) can be created using the **Formula** node (all Dynamo versions) or **Code Blocks** (Dynamo 0.7.x only).[43] Using logical operators like “and” and/or “or”, parentheses and nested conditional statements, you can create complex rules.

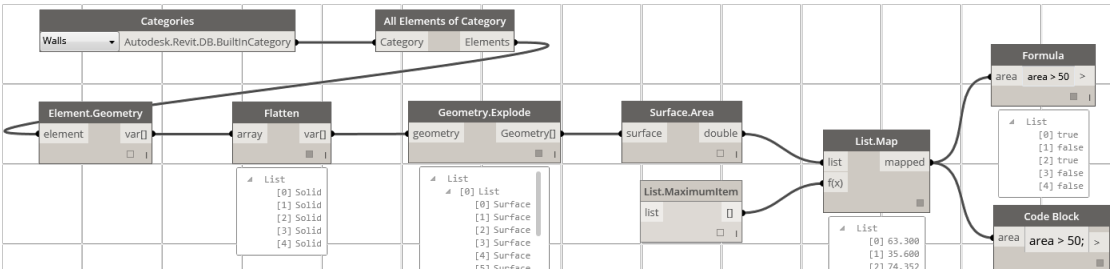

Fig. 11: Using conditional statements in Formula nodes and Code Blocks to evaluate wall areas

Another way of creating a conditional statement is, again, by using the Python-based access to the Revit API, querying an element property and

42 Note that in Dynamo 0.6.3 conditions return 1 and 0 instead of True and False.

43 Be aware that the syntax for conditional statements differs between Formula nodes and Code Blocks. Formula nodes are based on **NCalc**, the syntax of which is documented on the NCalc project page (http://ncalc.codeplex.com/). Find out more about conditional statements in Code Blocks in the Dynamo Language Guide, pp. 34-37 (http://dynamobim.org/wp-content/uploads/forum-assets/colin-mccroneautodesk-com/07/10/Dynamo_language_guide_version_1.pdf)

returning a boolean value, e.g. finding out if a certain wall type is a curtain wall (see Fig. 12).[44]

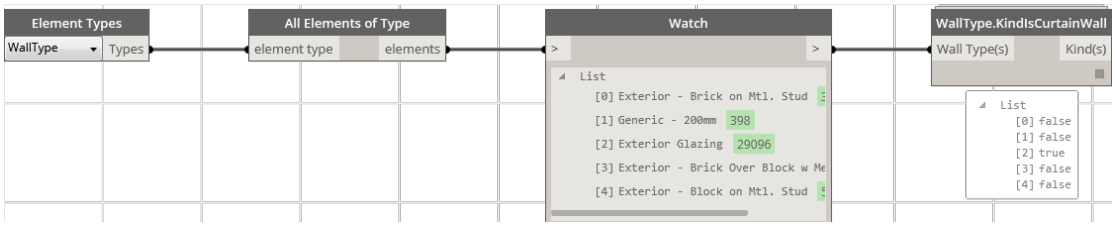

Fig. 12: Checking for curtain wall types using a Python-based custom node

Sorting Collections

Sorting a list of elements is a task that is commonly used in a lot of workflows (think schedules). When it comes to sorting Revit elements in Dynamo, the most useful node is **SortByKey** (see Fig. 13). It takes a list and a so-called key projector. The key projector can be any node that returns a property for each item in the list, e.g. the element name.

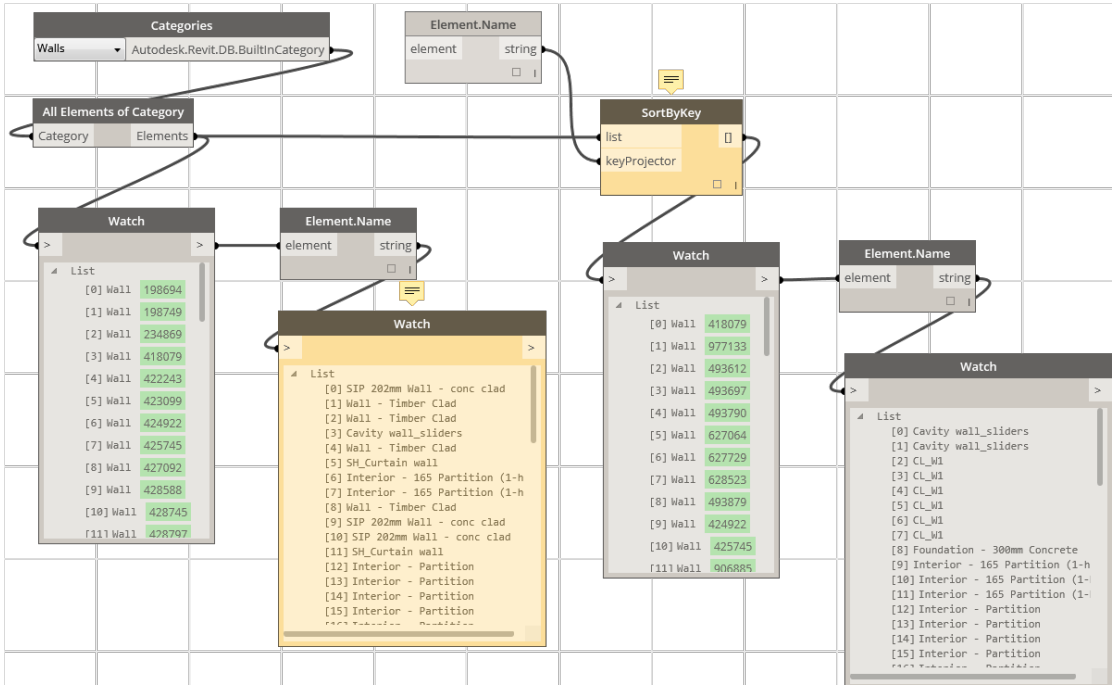

Fig. 13: Sorting walls by name

The tricky thing is that the key projector needs to be a single node. Consider the following workflow: You want to sort a list of walls by the elevation of their respective level. This will require that you actually use two nodes to obtain the

44 Using custom node **WallType.KindIsCurtainWall**

elevation: the custom node **Element.Level**[45] and the built-in node **Level.Elevation**. If you were to use SortByKey, you would have to wrap the two nodes in a custom node first (calling it “Element.ElevationOfLevel” perhaps) in order to have a single node as the key projector. An alternative approach that also lends itself well to sorting any type of nested list is using the custom node **List.SortListOfLists** (see Fig. 14).[46]

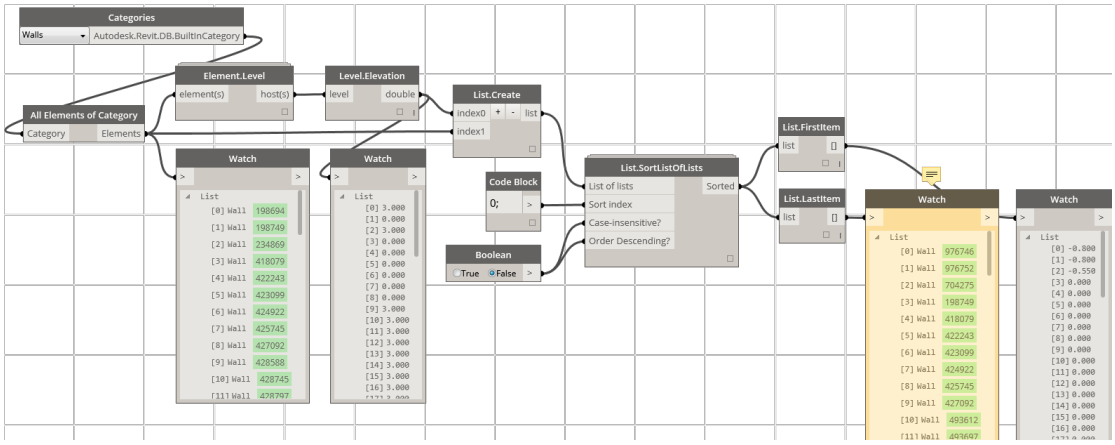

Fig. 14: Sorting a nested list

Also, keep in mind when trying to sort by multiple criteria that you need to reverse the sort order in your graph (see Fig. 15). For example, if you wanted to sort a number of point coordinates by X and Y, you would need to sort them by Y first, then by X[47]. Since Dynamo processes lists item by item, it will preserve the order of the previous sort operation as best as possible.

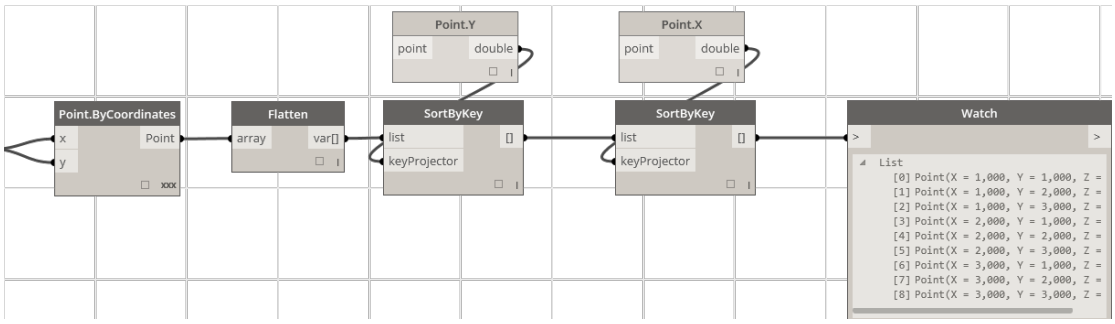

Fig. 15: Reverse sorting order when sorting by multiple criteria

45 Not included in Clockwork for Dynamo 0.6.3

46 Called **“Sort List Of Lists”** in Clockwork for Dynamo 0.6.3 – this custom node uses Python's powerful list handling abilities.

47 Hint: In real life you might want to round the X and Y values a bit first.

Grouping Collections

Grouping elements is very similar to sorting (see Fig. 16). The key difference is that grouping will add one level of nesting to the list as all items that share the same characteristic (provided by the key projector) will be stored in a common sublist. For grouping, of course, the same rules regarding the key projector apply. Like with sorting, grouping preserves previous sort orders as best as possible.

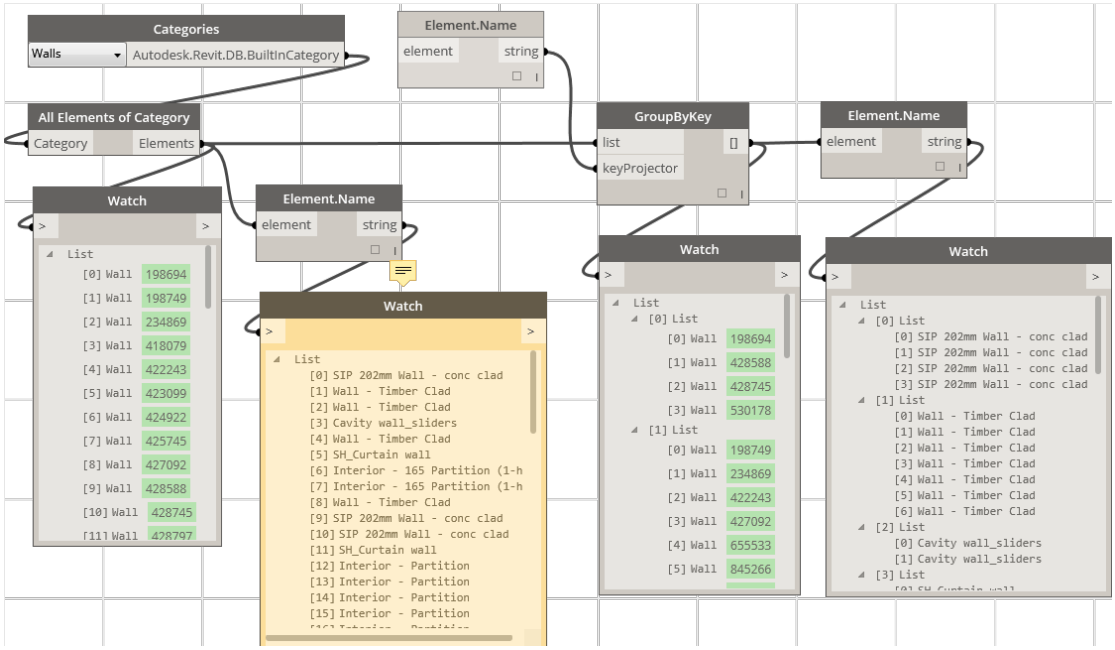

Fig. 16: Grouping walls by name

Filtering Collections

The ability to filter lists according to different criteria is probably the most important part of managing element collections, at least when reviewing a model. Fortunately, Dynamo provides a decent infrastructure in that department. The **List.Filter** node works just like the SortByKey and GroupByKey nodes: it takes a list and a function as its input (see Fig. 17). In this case, the function needs to be a condition (i.e. it has to return a boolean value). The condition is then applied to each element in the list to filter that element in or out.

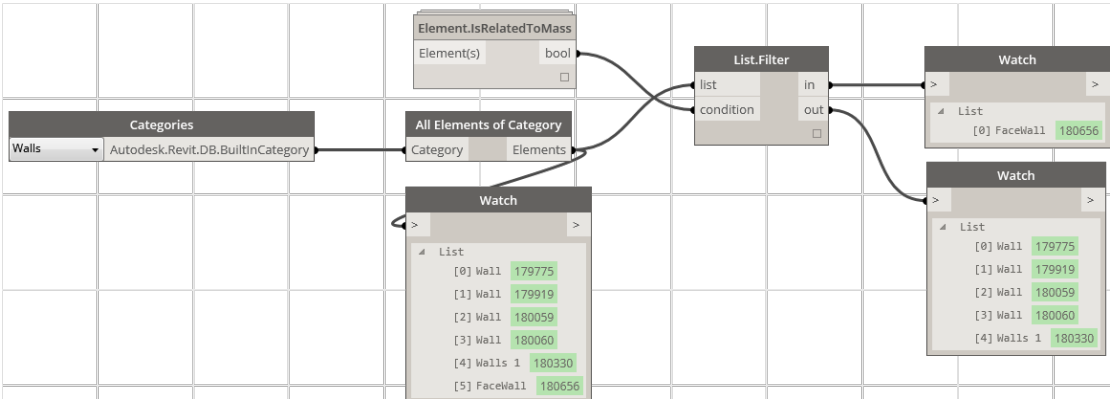

Fig. 17: Using the List.Filter node with a custom condition

As with sorting and grouping, the function needs to be a single node, so either wrap a more complex condition in a custom node or use **List.FilterByBoolMask** to build more complex filters without creating a custom node every time (see Fig. 18). List.FilterByBoolMask lets you use any list of booleans as a filter for your original list.

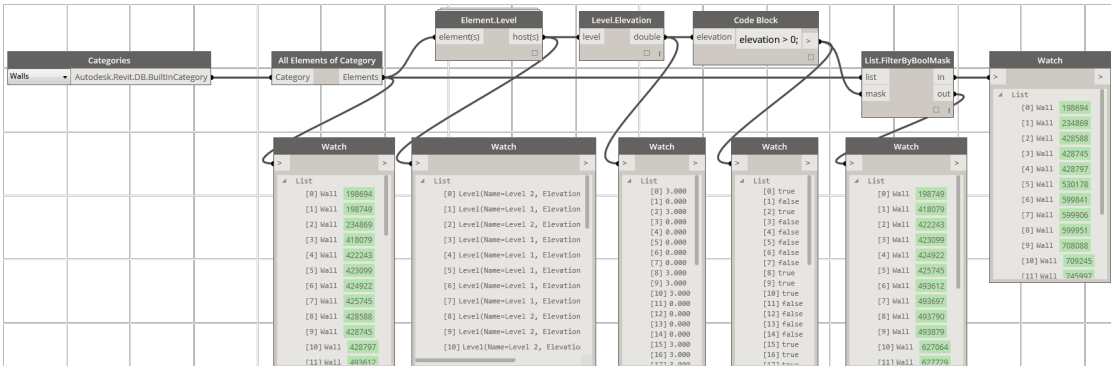

Fig. 18: List.FilterByBoolMask allows for more complex filters

Note that this node will not work well with nested lists. For filtering nested lists you will have to use the **List.Combine** node in order to have each nested list filtered separately according to a nested list of booleans (see Fig. 19).[48]

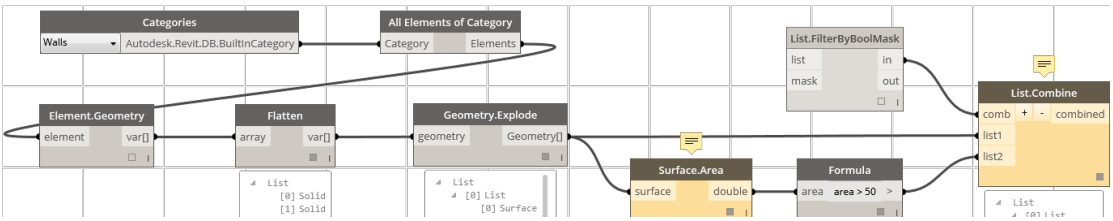

Fig. 19: Filtering a nested list using the List.Combine node

48 This technique only works in Dynamo 0.7.x – in 0.6.3 you would have to cycle through the nested lists using Python. Clockwork for Dynamo 0.6.3 contains two such custom nodes: **Combine List With Nested List** and **Combine Two Nested Lists**.

Managing Results

After having acquired a collection of elements from the model, having refined that collection further and having obtained the desired results by checking the collection against a rule, what can be done with those results? Since they are not (yet) part of the model, there are several possible approaches: Simply visualise them in the model, bring them into the model by altering element properties, export them or even fix certain conditions found in the model based on the obtained results.

Examining Results Inside Dynamo

Just for the sake of completeness, it should be mentioned that there are several ways of examining results inside Dynamo:

- Preview bubbles
- Watch node
- Watch 3D node
- Background preview

When working with Watch 3D nodes, remember that you can disable the preview status of individual nodes or entirely disable the preview of upstream geometry. Also note that Revit elements in Watch nodes can be clicked on to zoom into the respective element in Revit.

Visualising Results

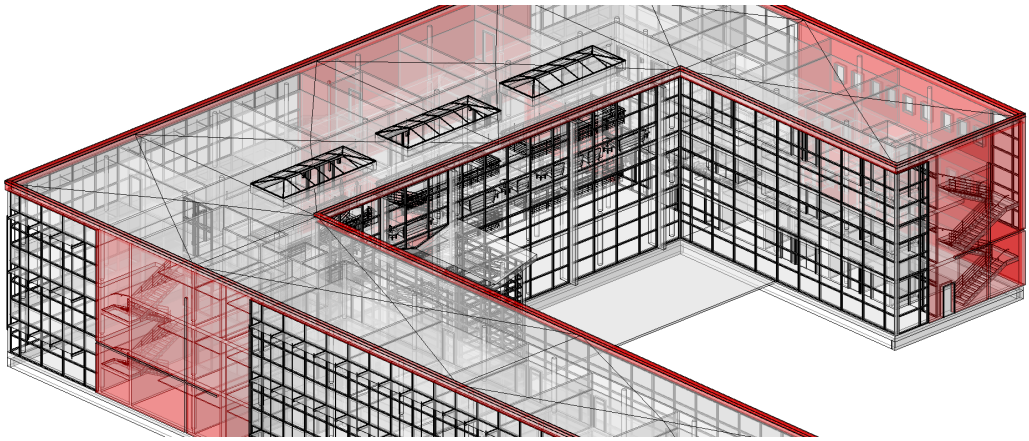

Fig. 20: Element overrides in view

If you want to visualise your results but shy away from writing data into your model elements, it is best to use the built-in **Element.OverrideColorInView** node. The example shown below (see Fig. 21) highlights all walls that contain a certain material. This will, of course, write the color (and fill pattern)

overrides for the affected model elements to the current Revit view. However, if your project file is not based on an English language template, this node will not work for you.[49]

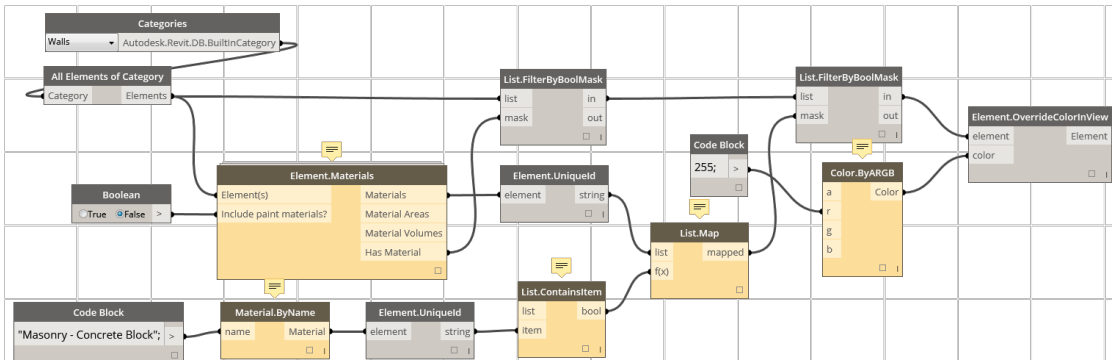

Fig. 21: Overriding the surface colour (and pattern) of all walls that contain a certain material

Writing Results To The Model

If you're like me, you might not like element overrides that much and prefer to work with actual data in your model using view filters or schedules with conditional formatting.[50] After having set up the necessary shared parameters in your project and/or your families, you can use the built-in **Element.SetParameterByName** node to write the desired data to the respective model elements for further downstream use with the aforementioned OOTB Revit functionality.

If you aren't comfortable with the those options (element overrides or writing data into model elements), selection sets might be a more suitable approach for you. Personally, I don't much like doing manual work with them in Revit, but recently I've come to rely on them quite regularly when bringing selections from Dynamo to Revit. The nice thing about selection sets is that they can be used like view filters in Revit, but without having to define any criteria (i.e. rules) regarding the membership of elements in a given selection set. Also, it is extremely easy to get rid of them again. To that end, I made a custom node

49 Read more about what is causing this on GitHub (https://github.com/DynamoDS/Dynamo/issues/880). Also note that there is a package by Julien Benoit for Dynamo 0.6.3 that circumvents this bug: **Override Element in View**.

50 Since this document was written for RTC attendees, I will assume that readers are familiar with the concepts of shared parameters, view filters and conditional formatting of schedules in Revit.

called **SelectionSet.ByElements** that will simply create a selection set from a list of elements (see Fig. 22).[51]

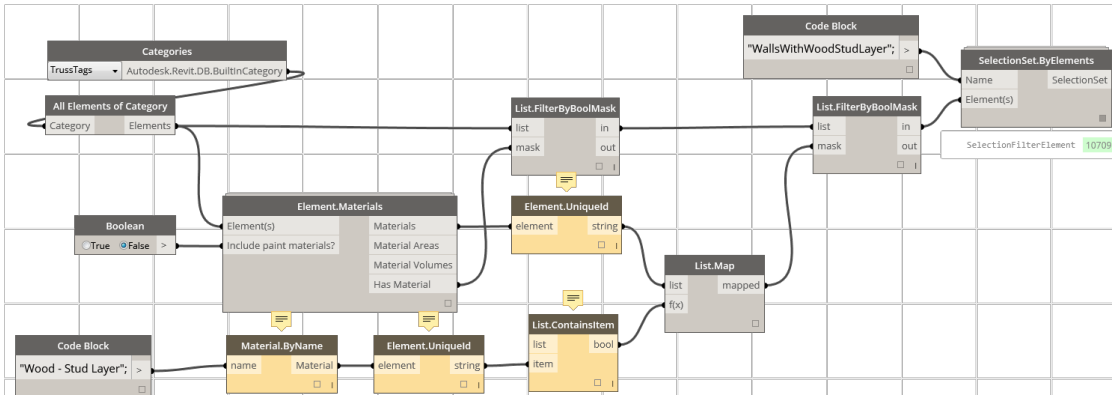

Fig. 22: Saving a filtered list to a selection set in the model

Exporting Results

Again, just for the sake of completeness: Dynamo also has some built-in export options. For exporting tabular data export, use **File.ExportToCSV** or **Excel.Write**. Find tutorials for those in the samples that ship with Dynamo. Phil Lazarus a.k.a. the BIM Troublemaker recently posted two articles on Excel round-tripping with Dynamo that are definitely worth reading.[52]

Fixing Things

One thing “classic” model checking usually cannot do for you is fix issues in your model automatically. With Dynamo, there really is no significant difference between creating a rule-based check and a rule-based automation. This allows you to feed the results of a model checking routine into another set of nodes that exert some form of change on the model. Obviously, it might be a challenge to automate tasks that would usually require a designer, but simple corrections can easily be included at the end of a model checking routine. If you need those types of correction depends largely on the skill and, more importantly, discipline of your users. Consider the following two simple examples for fixing model and annotation elements:

- **Fixing model elements:**
  Walls and their inserts (i.e. windows and doors) both have an orientation property that returns the vector of the direction they are facing towards. If a window is placed into an exterior wall correctly,

51 Not available in Clockwork for Dynamo 0.6.3

52 http://bimtroublemaker.blogspot.com/2014/09/practical-dynamo-excel-linking.html and http://bimtroublemaker.blogspot.com/2014/09/practical-dynamo-excel-linking-2.html

both the wall and the window should face into the same direction. Using a few custom nodes, it is easy to check if this is indeed the case and otherwise flip the facing direction of those windows that face into the building.[53]

- **Fixing annotation elements:**
  Most likely you will have rules regarding the use of annotation tags or dimensions with relation to different view scales and/or project phases. Dynamo could help you enforce those rules, e.g. by checking the appropriateness of the dimension styles used in each view and switching their types if they do not comply with the office standards.

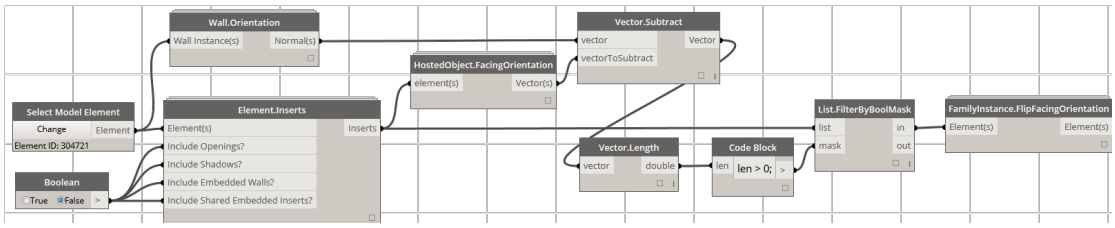

Fig. 23: Flipping the orientation of filtered list of family instances

53 Using the custom nodes **Wall.Orientation** (formerly “Get Wall Orientation”), **HostedObject.Orientation** (formerly “Get Hosted Object Orientation”) and **FamilyInstance.FlipFacingDirection** (not available in Clockwork for Dynamo 0.6.3).

Case Studies

My personal interest in task automation with Dynamo began with my (still unfinished) PhD thesis on model checking as a design tool. One of my objectives is to assess the feasibility of including model checking in the toolset of the designer as opposed to limiting it to the work domain of specialists. In that regard, one approach would be to offer model checking tools within the comfort zone of the designer, i.e. a BIM authoring environment like Revit. For the purposes of conveying the main ideas about model checking with Dynamo, the case studies below are, at least to some extent, grossly simplified when compared to actual building code. They should be understood as a learning resource at proof-of-concept level rather than a polished piece of software. They represent, however, typical cases of code-related model checking.

Natural Lighting

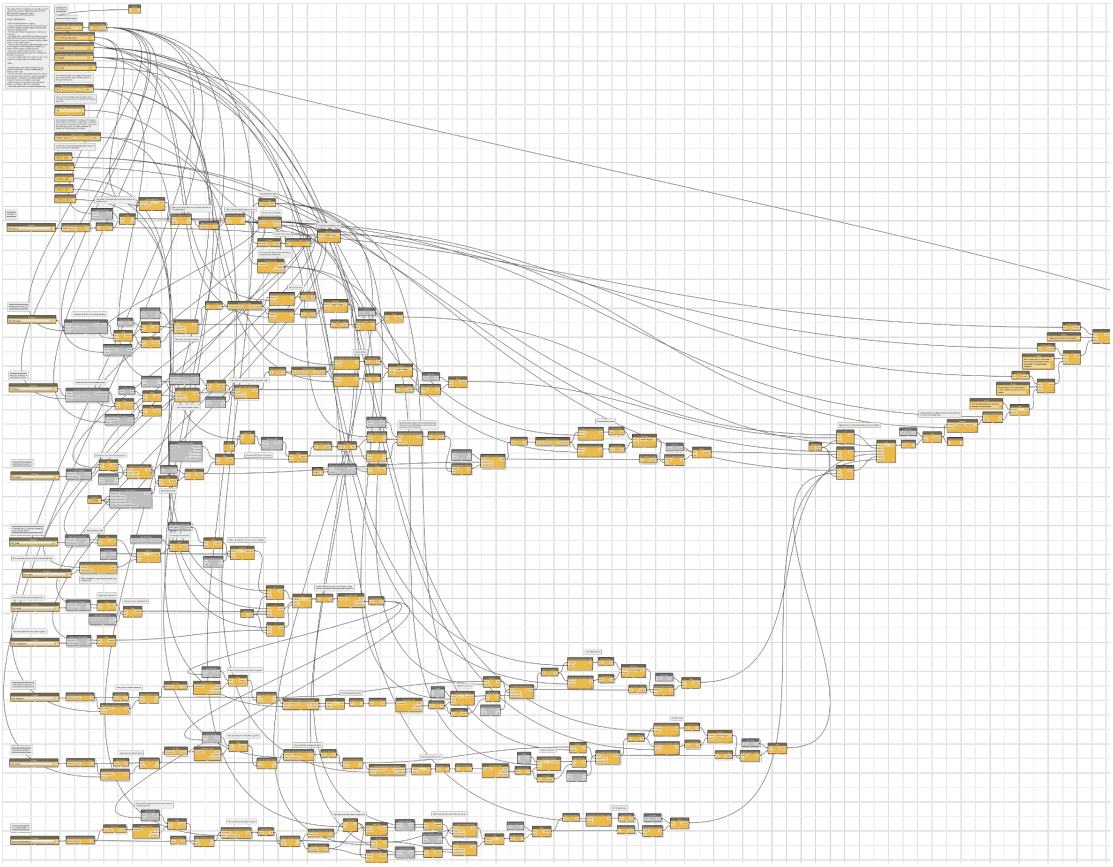

Fig. 24: Checking glazing area versus room area ratio

Many building codes include rules regarding the natural lighting of building. These rules typically only apply to occupied space, i.e. rooms that are suited or intended for the permanent presence of humans (as opposed to rooms like

storage, toilets, utility rooms etc.). For instance, in the building code of my region of Germany, these types of rooms need to have a glazing area that is at least one eight of their respective floor area. Since the Revit API doesn't offer a universal method to get this kind of data, there is no easy OOTB way in Revit to get a collection of glazed elements for a room. This case study describes the various approaches taken in order to obtain the information necessary to evaluate a room's glazing area and compare it to its floor area.

One of the fundamentals of model checking is that the information in the model should be as accurate as possible or rather: the quality of the model information will directly impact the quality of the model check results. Besides a decently modelled building we will obviously need rooms present in the model and we need to **identify those rooms that are classified as occupied space**. This can be easily done using a shared Yes/No project parameter for the rooms category (or any other classification system using a shared project parameter). Further, we will need to set Revit's **area and volume computation** to “Areas and Volumes”. This is necessary due to the fact that curtain wall panels do not carry any information as to what room they are bordering and we will need to establish a connection between them and the room volumes in order to find out which room they belong to. Therefore it is also imperative that the **upper limit (and or limit offset) of all rooms** is set in such a way that they extend to roofs and/or floor slabs that act as room bounding elements or in other words: we need the actual room volume. Besides being relevant for matching curtain wall panels to rooms, this might also be essential for finding windows hosted in roofs, as – depending on the Revit version used – they might not contain any room-related information either. Furthermore, all windows, doors and curtain panels need to be **set to their respective categories** – otherwise these elements will be ignored when creating collections. It is also assumed that for all doors and windows that are hosted in walls, the **widths and heights are type-based**, not instance-based. Finally, doors and curtain wall panels will need to have a **sufficiently transparent material** assigned to them in order to be taken into account.

Looking at the graph, everything starts with retrieving a collection of rooms that are classified as occupied space. From that collection we can derive several pieces of information that will come in handy for further downstream use:

- The number of rooms to be checked
- Their respective boundaries[54]

54 Using **“Get Room Boundaries”** (**Room.Boundaries** in Clockwork for Dynamo 0.7.x)

- Their respective area[55] and, based on that and a configurable setting, the required glazing areas
- Their Unique IDs[56]

Based on that information, we can begin creating collections for the various component categories that could qualify as glazing.

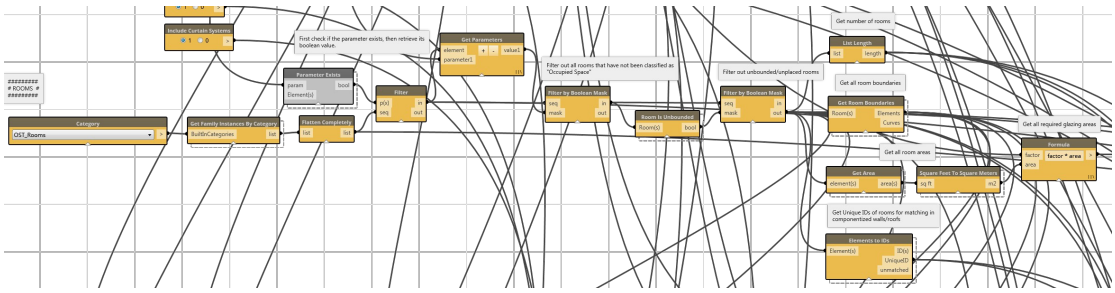

Fig. 25: Retrieving the room collection

We start by creating room-based collections of windows[57] (making use of their FromRoom and ToRoom properties) and subsequently filtering out all windows that are located between two rooms.[58]

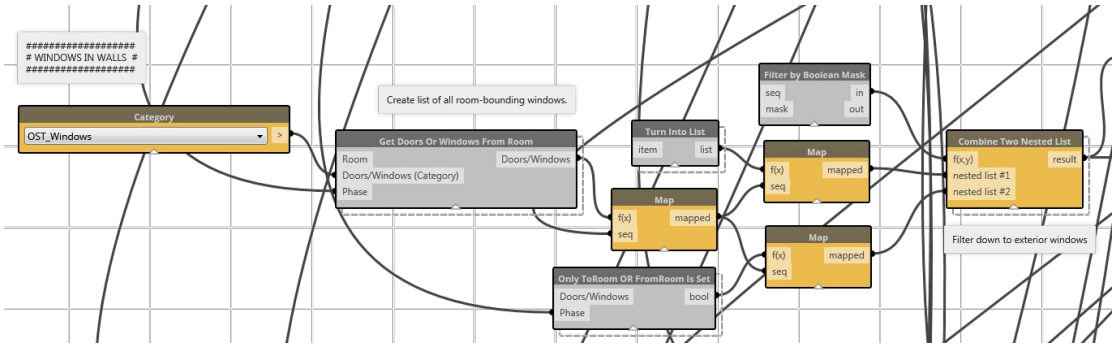

Fig. 26: Building a collection of exterior windows only

Based on this filtered list, we can then start pulling the width and height parameters of the windows, calculate their respective area and create subtotals for each room.[59] In this part of the graph, I am using a technique that I often find quite useful when dealing with nested lists. I record the number of elements in each sublist and subsequently flatten the nested list. Now I can

55 Using **"Get Area"** (**Element.Area** in Clockwork for Dynamo 0.7.x)

56 Using the custom node **"Elements to IDs"** - or the built-in node **Element.UniqueId** in Dynamo 0.7.x

57 Using **"Get Doors Or Windows From Room"** (currently only in Clockwork for Dynamo 0.6.3)

58 Using **"Only ToRoom OR FromRoom Is Set"** (currently only in Clockwork for Dynamo 0.6.3)

59 Using **"Sum List"** by Stephen Elliott in Dynamo 0.6.3 – in Dynamo 0.7.x use the built-in node **Math.Sum** instead

easily perform all sorts of operations on the list without using the Map[60] node. When finished, I simply rebuild the original nested list structure.[61]

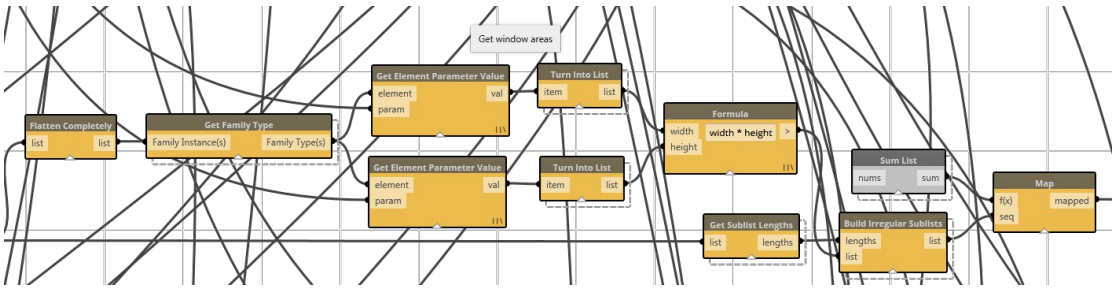

Fig. 27: Flattening and rebuilding a nested list structure while calculating window areas

The approach for doors is almost exactly the same as for windows except that an additional filter is used for creating the collection. We check if each door actually contains any transparent elements.[62]

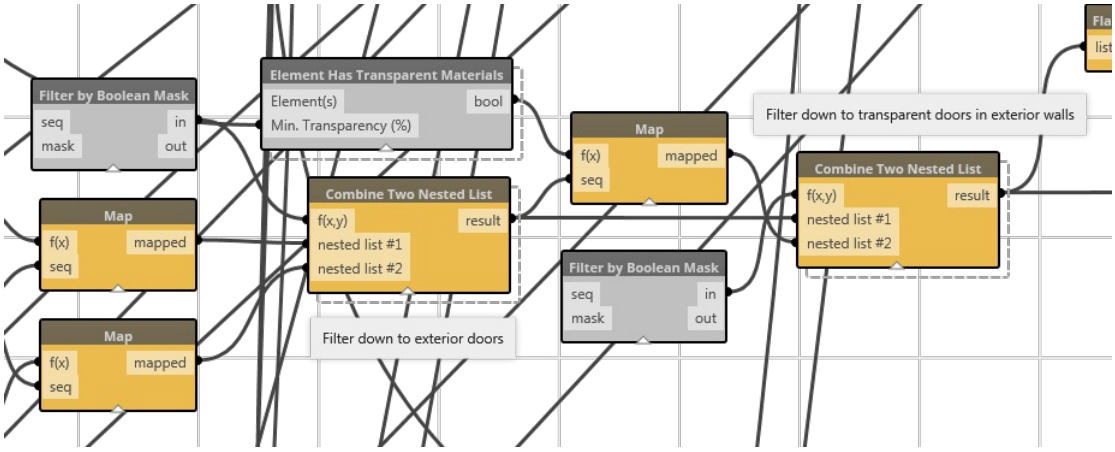

Fig. 28: Filtering out non-transparent doors

Next are windows hosted in roofs. As mentioned previously, depending on the Revit version your model was created in, a roof window will not have any information in its FromRoom and/or ToRoom properties.[63] We can easily find all roof hosted windows by filtering the list of room-bounding elements down

60 **List.Map** in Dynamo 0.7.x

61 Using **"Get Sublist Lengths"** and **"Build Irregular Sublists"** (**List.Sublist.Lengths** and **List.SublistsByLengths** in Clockwork for Dynamo 0.7.x)

62 Using **"Element Has Transparent Materials"** (currently only in Clockwork for Dynamo 0.6.3)

63 Using a window schedule, this can be easily checked. If the information is indeed available, both properties will most likely contain the same value. Note that this will in all likelihood not impact the workflow described here.

to only roofs and subsequently pulling all their inserts.[64]

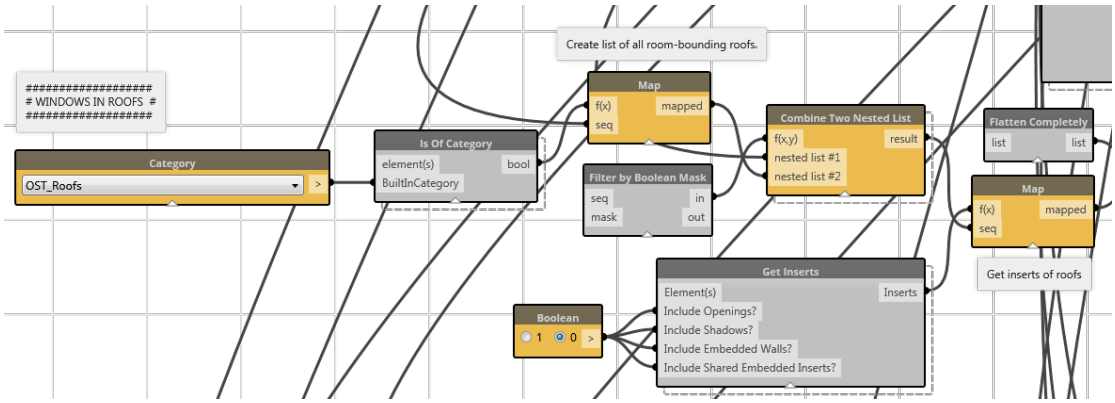

Fig. 29: Finding all room-bounding roofs and their respective inserts

Based on the window location[65] we can now create a point slightly below the window.

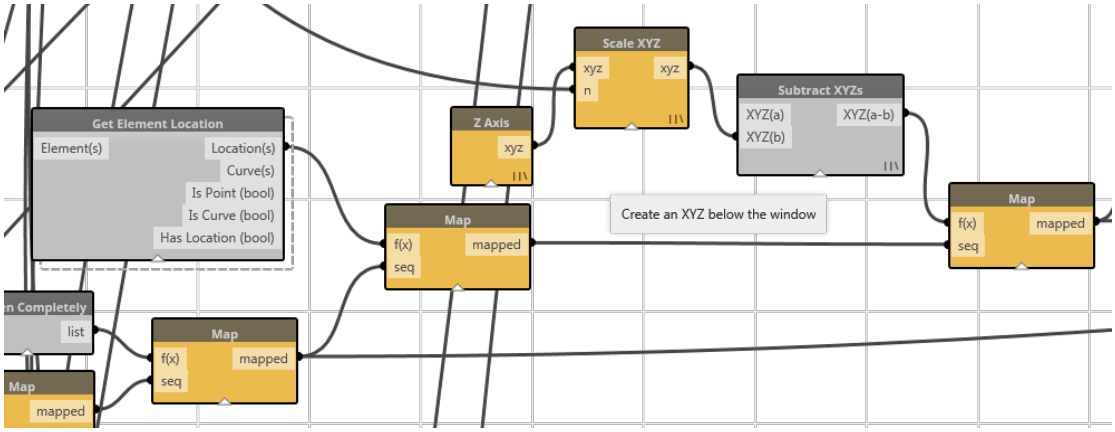

Fig. 30: Creating a point below each roof-hosted window

Using these points, we can find out if a given window is located in any of the rooms.[66] Since a roof can obviously span across multiple rooms, a window hosted in such a roof would have to be checked against multiple rooms, hence the slightly complex part of the graph here. After that, the area is calculated using the same process as previously with windows and doors hosted in walls.

64 At this point, the inserts should be filtered down to windows only, but this part has been omitted from this graph.

65 Using **"Get Element Location"** (**Element.Location** in Clockwork for Dynamo 0.7.x). I originally made this node because the built-in **Get Family Instance Location** could not handle locations for some elements such as lines, but in this case it would do the job just as well.

66 Using **"Is Point In Room"** (currently not in Clockwork for Dynamo 0.7.x)

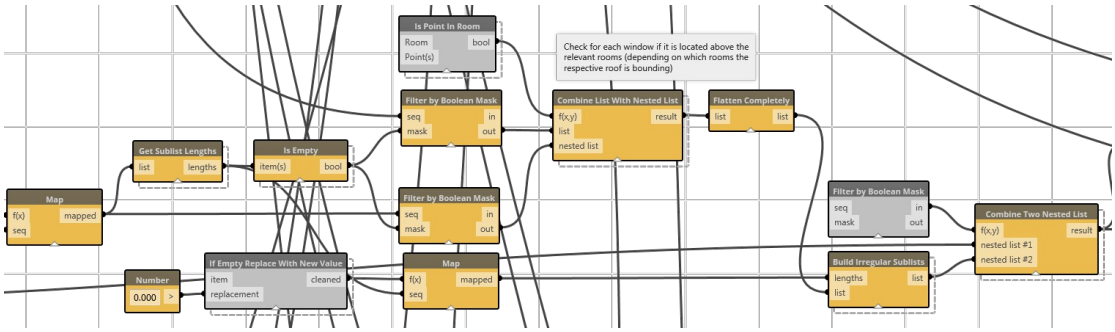

Fig. 31: Matching roof-hosted windows to rooms using the previously generated points

All that is left unprocessed now are elements hosted in curtain walls, curtain systems and glazed roofs. Also, walls might contain embedded curtain walls which also need to be taken into account. To that end we need to create collections of all of these object classes. All of these should have registered as room-bounding objects in our original collection of room boundaries. Therefore we can filter that collection down to the respective category. For the walls we first need to create a unified collection of walls and embedded walls which we can then filter by whether they are actually curtain walls.[67]

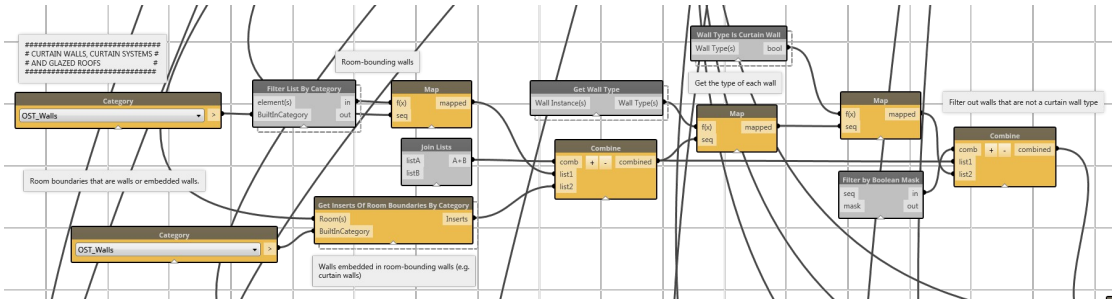

Fig. 32: Creating a unified list of all room-bounding walls and embedded walls and filtering down to curtain walls

Similarly, we need to filter out all non-glazed roofs[68] while we can just take all curtain systems from the boundaries collection.

67 Using **"Get Wall Type"** and **"Wall Type Is Curtain Wall"** (**Element.Type** and **WallType.KindIsCurtainWall** in Clockwork for Dynamo 0.7.x)

68 Using **"Filter Roofs By Kind"** (**Roof.KindIsGlazed** in Clockwork for Dynamo 0.7.x)

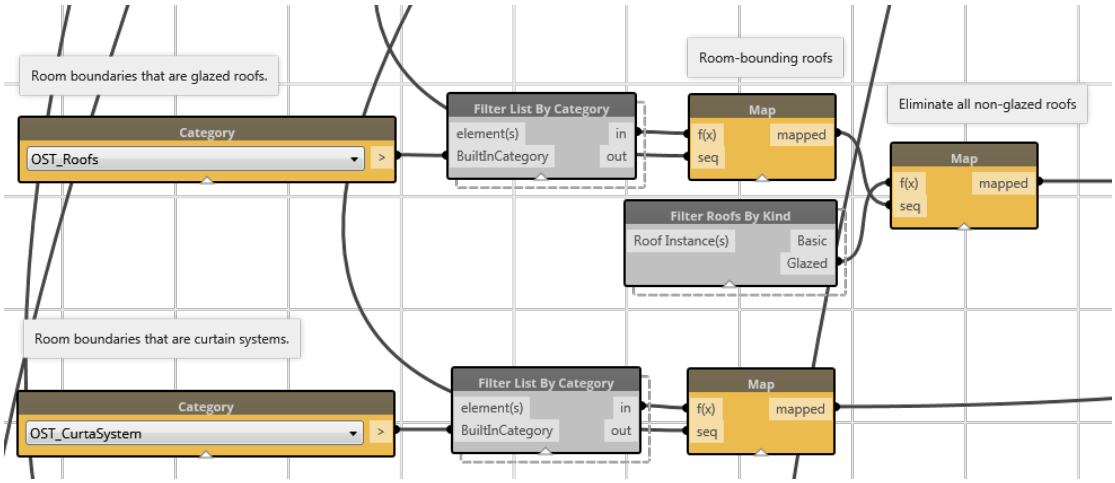

Fig. 33: Building collections of glazed roofs and curtain systems

Since most elements hosted in these components will not contain room-related information, we create a flat list of unique components (curtain walls, curtain systems and glazed roofs) using their Unique IDs as identifiers for future reference.

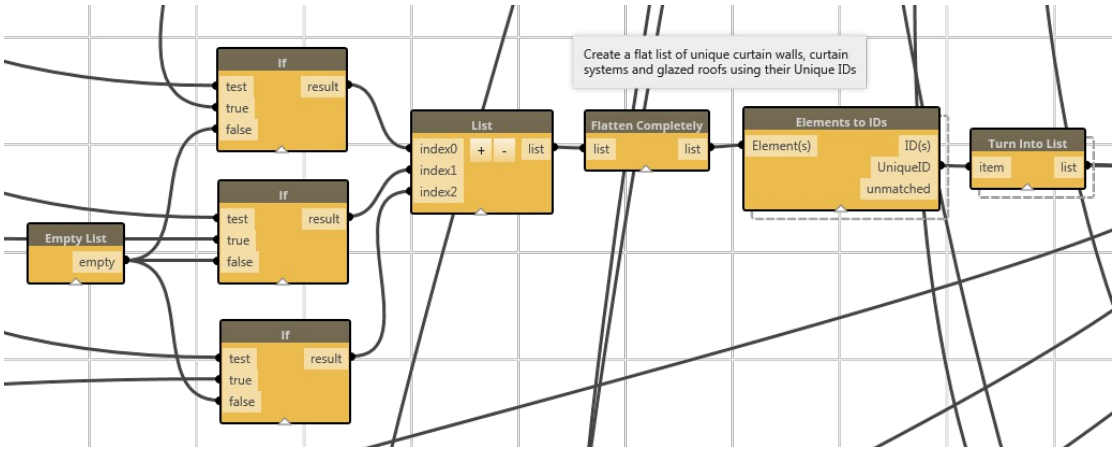

Fig. 34: Building a flat list of Unique IDs of curtain walls, curtain systems and glazed roofs

Since these systems may not only contain curtain wall panels, but also windows and doors, we will continue with these elements before dealing with curtain wall panels. For the windows, we first create a filtered list of exterior windows only (i.e. windows that only exit to one room). After that we can filter them by host, using the unique list of curtain systems etc. we created previously.[69]

69 Using Stephen Elliott's **"List Contains"** - in Dynamo 0.7.x use the built-in node **List.ContainsItem** instead.

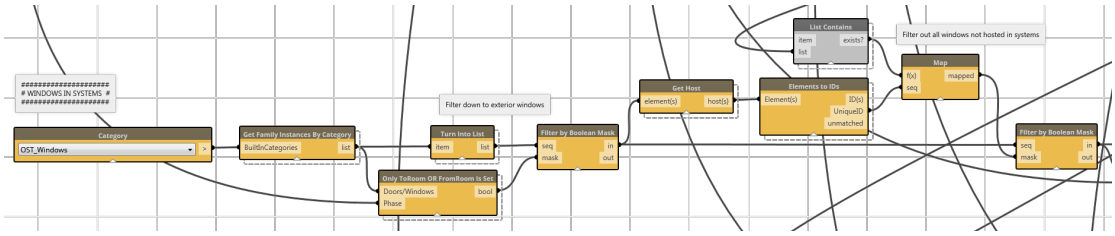

Fig. 35: Building a collection of windows hosted in room-bounding systems

We can then match them to the corresponding rooms by their FromRoom / ToRoom properties.

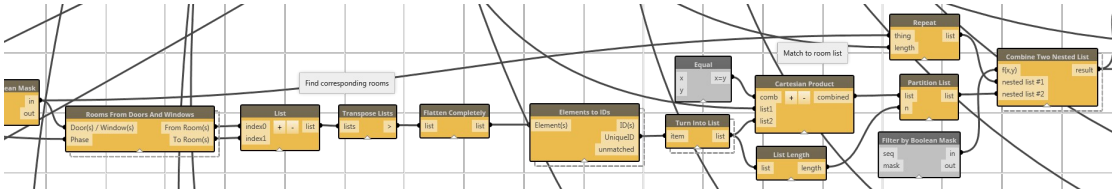

Fig. 36: Matching windows to rooms

As opposed to windows hosted in walls, In the case of windows hosted in systems the width and height are instance properties. The process for doors is almost exactly the same, but there is an additional filter again that checks for material transparency. Last are the curtain wall panels. We first collect all curtain panels from the entire models and then filter out those that are not hosted in the list of room-bounding systems we created previously. Since curtain panels do not contain any room-related information, we need to use an approach similar to that used for windows hosted in roofs. Using their facing orientation[70] and location[71] we can project points to both sides of each panel looking for rooms.

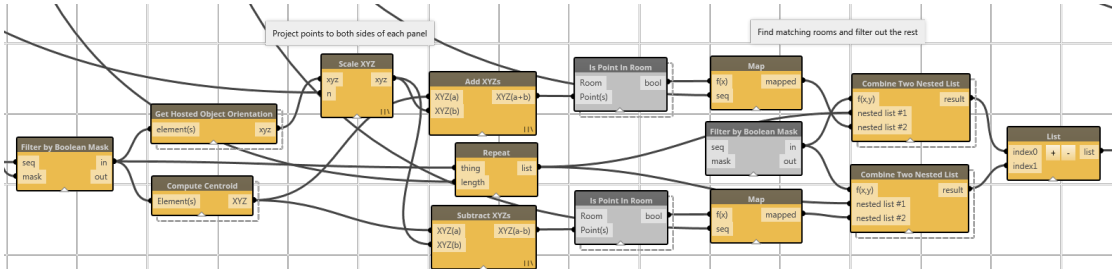

Fig. 37: Matching curtain panels to rooms

70 Using **"Get Hosted Object Orientation"** (**HostedObject.FacingOrientation** in Clockwork for Dynamo 0.7.x)

71 Using Julien Benoit's **"Compute Centroid"**. Unlike curtain panels hosted in divided surfaces or repeaters, classic curtain panels do not have a Location property. Julien's fiendishly clever node converts an object into a solid and then uses the Revit API's ComputeCentroid() method to find its location.

Depending on their host system, curtain panels might not have values assigned to their Width and Height properties. What all of them have in common, though, is the Area property – less calculating for us.

Aggregating the results is actually the easiest part. All of the areas retrieved for the respective component types are then summed up in a room-based list and compared to the required glazing area for each room. Using a simple condition we can then filter the list of rooms down to those that do not have sufficient glazing.

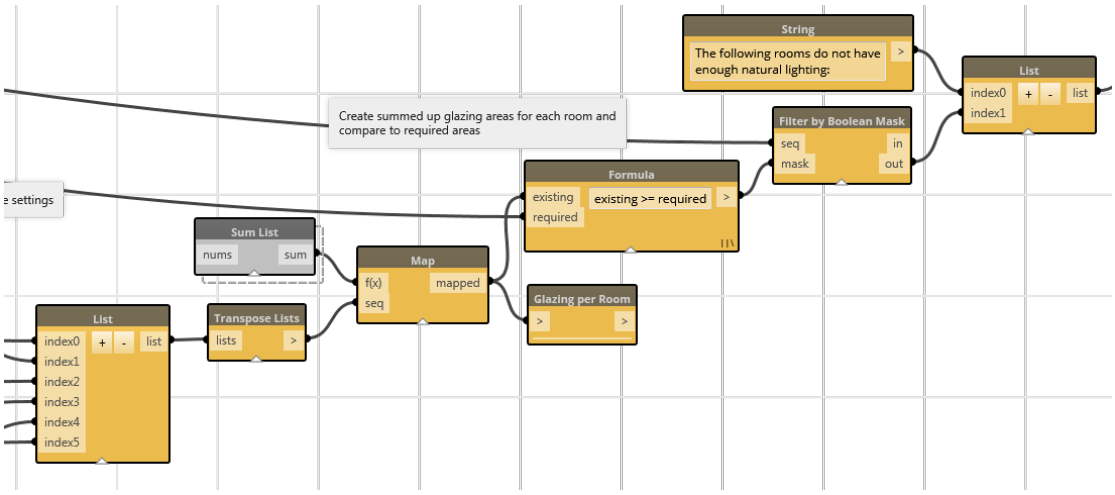

Fig. 38: Aggregating the results and checking for rule compliance

Since this graph is a work in progress, it still has some issues. For one thing, it currently doesn't exclude curtain **panels that have rooms on both sides** (which would be an indication for an interior curtain wall). It also does not take into account the **percentage of transparent material** present in a door. And finally, it completely ignores **panels hosted in divided surfaces or repeaters** (or placed freely using Dynamo).

Egress Paths

The second case study is a bit more simplified than the first since it represents an older state of the graph still developed in Dynamo 0.6.3 – its current 0.7.x version unfortunately still has a few stability issues (see “”).

Egress paths represent another aspect common to most if not all building codes. Much has been written about how to create egress paths in Revit, but

all of those methods require an obscene amount of manual work.[72] Enter Dynamo. All doors in a Revit model know the rooms they exit to and – as was shown in the previous case study – we can access those properties using custom nodes in Dynamo. This allows us to build a graph of room connections, e.g. from any given room to a fire escape. If we have that kind of graph, it becomes what is known as a shortest path problem[73] in graph theory[74] or, in our case – since we are dealing with spatial data – topological graph theory.[75] It's a classical computation problem, perfectly suited for Dynamo.

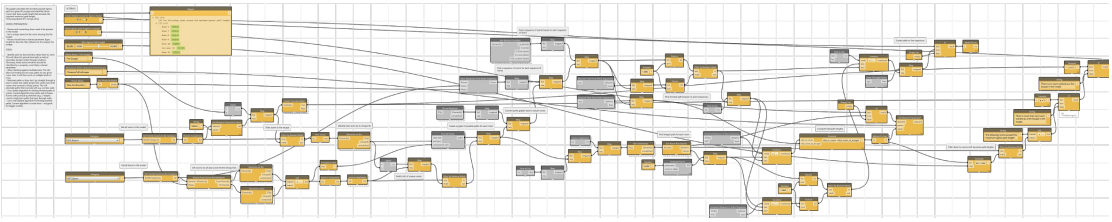

Fig. 39: Finding the shortest possible egress path for all rooms

Again, model preparation is key. In this simplified example, the requirements are fairly straightforward. You'll need rooms and doors connecting them, obviously. The room that houses the fire escape should have a unique name so we can identify it. And finally, there should be a shared project parameter (type: length) for the room category that can later be filled with the shortest distance to a fire escape.

We start off with retrieving a collection of rooms from the model. Using the room names, we can identify the fire escape for which we then retrieve its Unique ID.

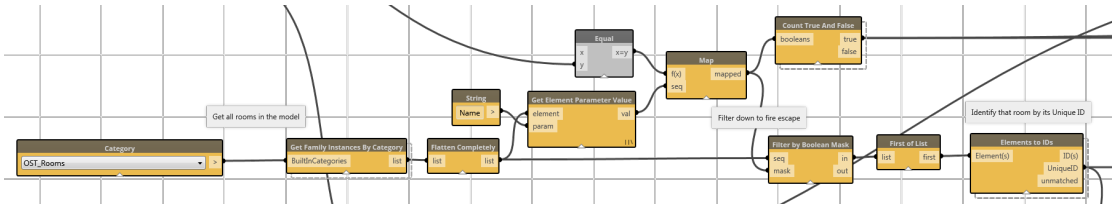

Fig. 40: Finding the fire escape

72 Start with this list of egress-related posts on Steve Stafford's blog: http://revitoped.blogspot.com/2012/08/egress-post-summary.html
73 http://en.wikipedia.org/wiki/Shortest_path_problem
74 http://en.wikipedia.org/wiki/Graph_theory
75 http://en.wikipedia.org/wiki/Topological_graph_theory

Next, we build a collection of doors and retrieve the rooms they lead to/from. We pull Unique IDs for all rooms which will later facilitate building the room graphs. We also use that list of rooms to build a unique list of rooms as we need to build a separate room graph for each room.

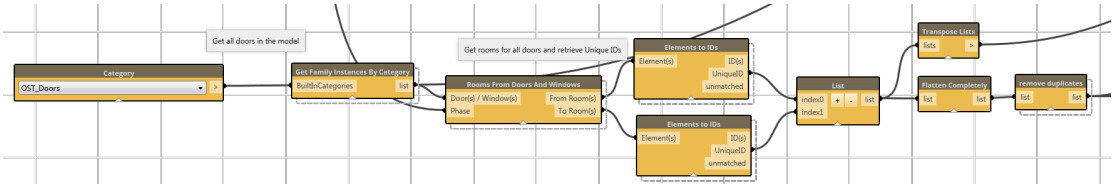

Fig. 41: Building a unique list of rooms from a collection of doors

Using those two lists (the unique list of rooms and the list of all room connections) we can build a graph of possible paths for each room[76] (always setting the fire escape as the end of the path) and convert the room IDs in those paths back to proper rooms.

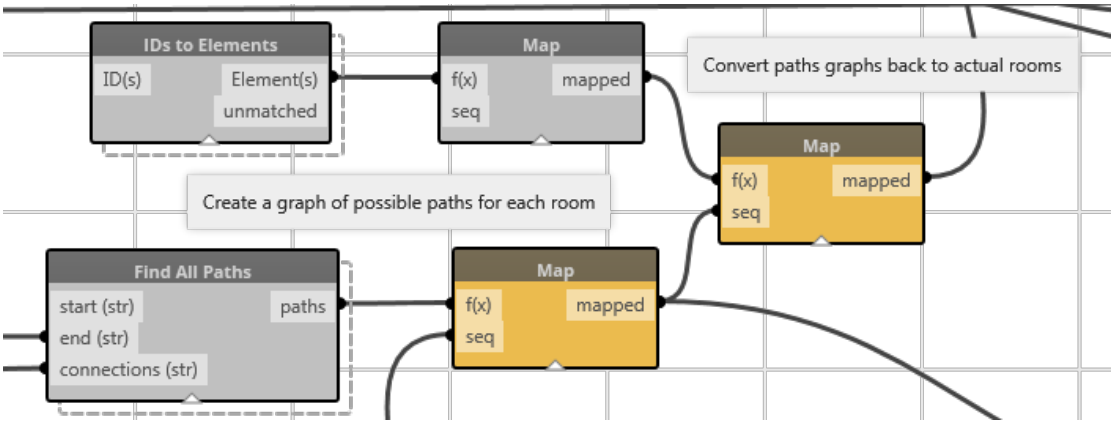

Fig. 42: Building path graphs for each rooms and converting them into room sequences

With the information available in the model we can convert a sequence of rooms into a sequence of doors that connect those rooms.[77] Those door sequences can, in turn, be converted into sequences of points.

76 Using **"Find All Paths"** (currently not in Clockwork for Dynamo 0.7.x)

77 Using **"Door Sequence From Room Sequence"** (currently not in Clockwork for Dynamo 0.7.x). Note that this node assumes that each pair of rooms is only connected by one door that would act as a means of escape.

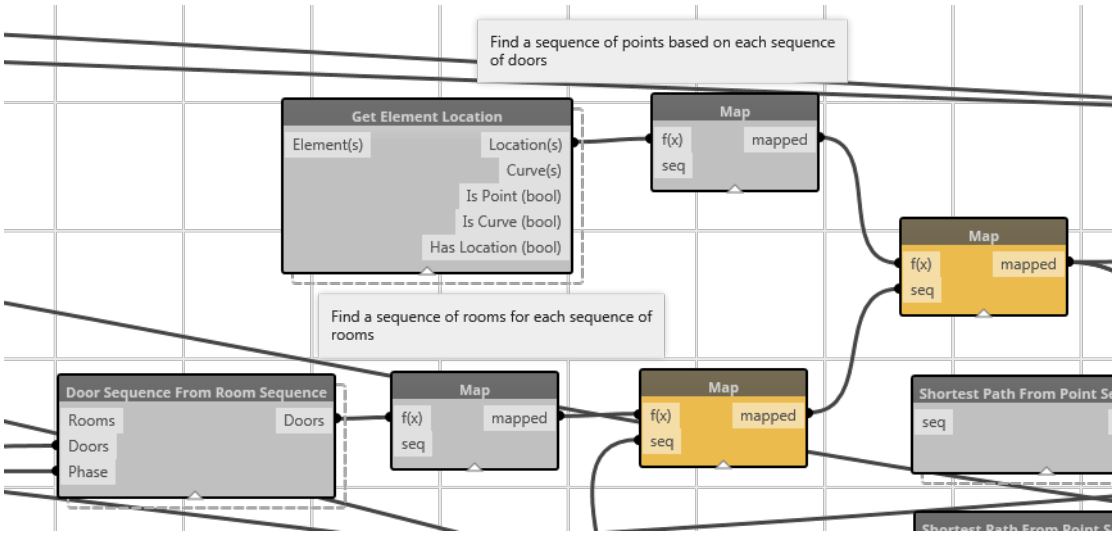

Fig. 43: Turning room sequences into door sequences and door sequences into point sequences

We can then compute the respective path length of each point sequence and find the shortest path for each room.[78]

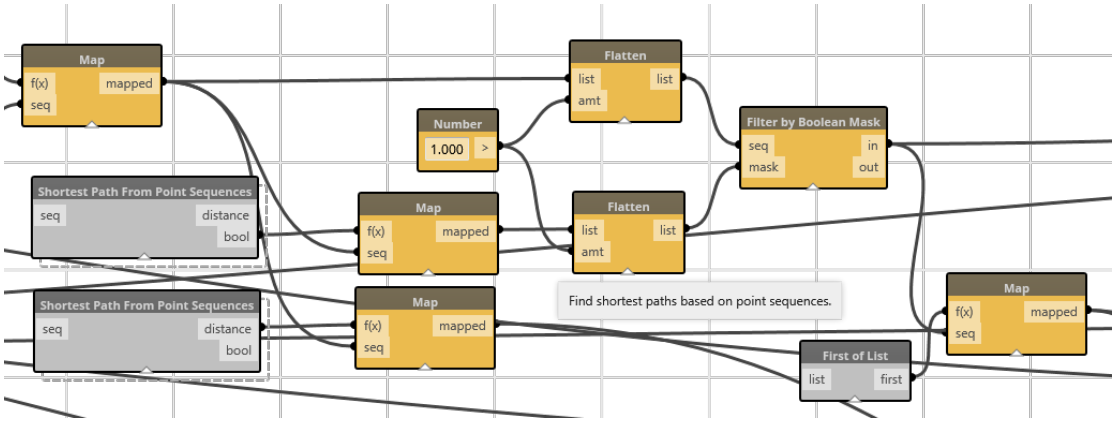

Fig. 44: Finding the shortest path for each room

That leaves us with one more path to compute: we need to find the path length inside the first room of each sequence. In order to do this we need to get the boundaries of each room and find the boundary point with the furthest distance to the first door that is used by the shortest path for that particular room.[79]

78 Using **"Shortest Path From Point Sequence"** (currently not in Clockwork for Dynamo 0.7.x)

79 Using **"Greatest Distance in Point Sequence"** (currently not in Clockwork for Dynamo 0.7.x)

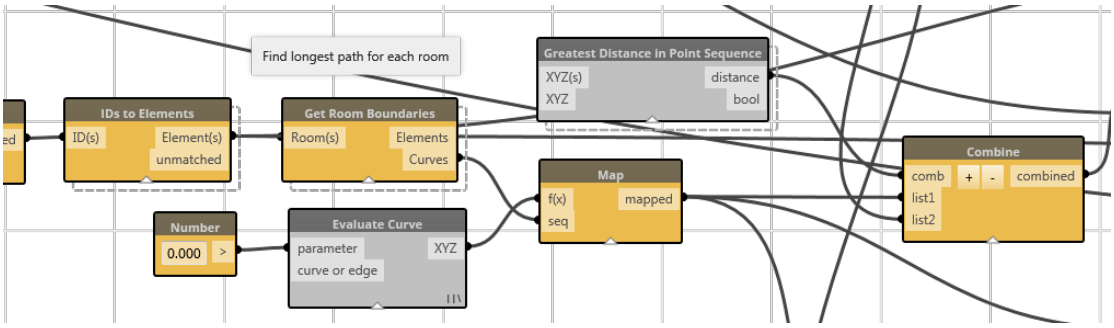

Fig. 45: Finding the furthest corner of each room

With that information available we can now compute the total egress path length for each room consisting of the shortest path length between the first and last room of each graph and the longest path inside each first room. This information can then either be viewed in the Dynamo graph itself or written to each room in the model using a shared parameter. Visualising it with a colour scheme for rooms should be child's play. Another option would be plotting each path with model lines based on the point sequences created earlier.

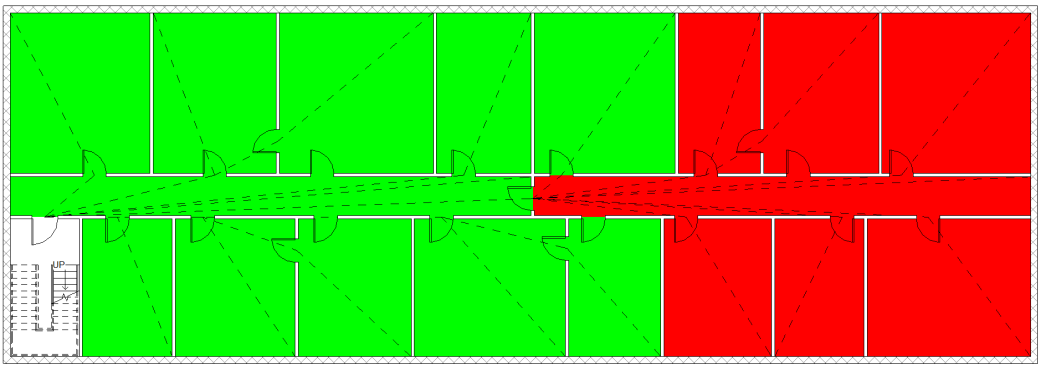

Fig. 46: Egress paths plotted from Dynamo over color scheme displaying whether a given room exceeds the maximum path length

Granted, the presented method is raw at best. Typically we would need to be able to **identify several possible exits** – therefore it would make sense to **identify those exits by doors and windows**, not by rooms. This change would allow the graph to work with ground level exits as well as secondary exits like windows. It would also allow computing results for a whole building and not just a single floor. The pathing itself is another matter that doesn't stand up to much scrutiny yet. For one thing, paths shouldn't be computed directly through a door's center point, but rather through **offset points on both sides of the door** – this would eliminate paths that coincide with e.g. corridor walls. The current in-room algorithm is as simple as it gets and will most likely **fail under more complex spatial conditions** as it always

computes the direct route between two doors (think L-shaped rooms). And finally, the algorithm computing the shortest path is a **simple brute force approach**. It would be much more elegant to use a real Dijkstra algorithm[80] that takes into account the cost of travelling along a path. Nonetheless, I think that this case study shows the potential of egress pathing inside a Revit model.

80 http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

A Rather Lengthy Conclusion

Dynamo is more than “just” a computational design tool. It's a tool to make tools. It *wants* you to make your own tools. Anything that is not available OOTB can be added to it – and shared. Which other BIM platform has that kind of infrastructure? Python nodes that access the Revit API are one way to do this. Another exciting possibility is the new **Zero Touch DLL import** feature in 0.7.x – it lets you import third party DLLs[81] which are then instantaneously available as nodes in the node browser, e.g. teach Dynamo to read IFC.[82] A great recent example is Adam Sheather's **DynaWorks**[83] project, accessing the Navisworks API from within Dynamo.

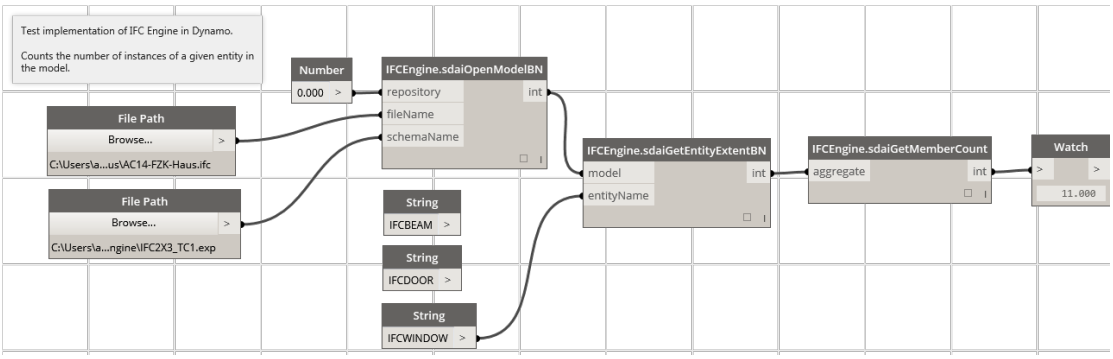

Fig. 47: Implementation of the IFC Engine DLL in Dynamo with Zero Touch Library Import, counting the number of windows in an IFC model

Automation is, of course, not an end in itself. We automate processes in order to save time, make money or, ideally, both. The time we save either has to outweigh the time we invest to achieve a higher degree of automation or there has to be some other gain, e.g. in precision or quality.

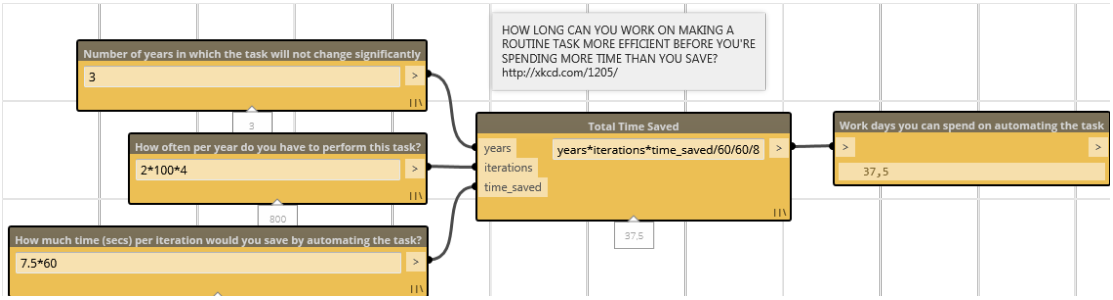

Fig. 48: Have trouble deciding whether it's worth it to automate a process? Try this simple Dynamo graph to help you decide.

81 http://en.wikipedia.org/wiki/Dynamic-link_library

82 The graph in Fig. 47 uses a custom wrapper DLL around the IFC Engine DLL, imported with Zero Touch Library Import. IFC Engine is available here: http://rdf.bg/ifc-engine-dll.php

83 http://stuffandbims.blogspot.de/2014/10/dynaworks-is-here-navisworks-library.html

Fortunately enough, I have experienced very tangible proof of the feasibility of process automation with Dynamo in my daily work on a number of occasions. My largest Dynamo project to date is a model checking routine that helps us review student project submissions in our introductory Revit class (see Fig. 49). With well over 200 students taking that class each year, this project has been a real time-saver.[84]

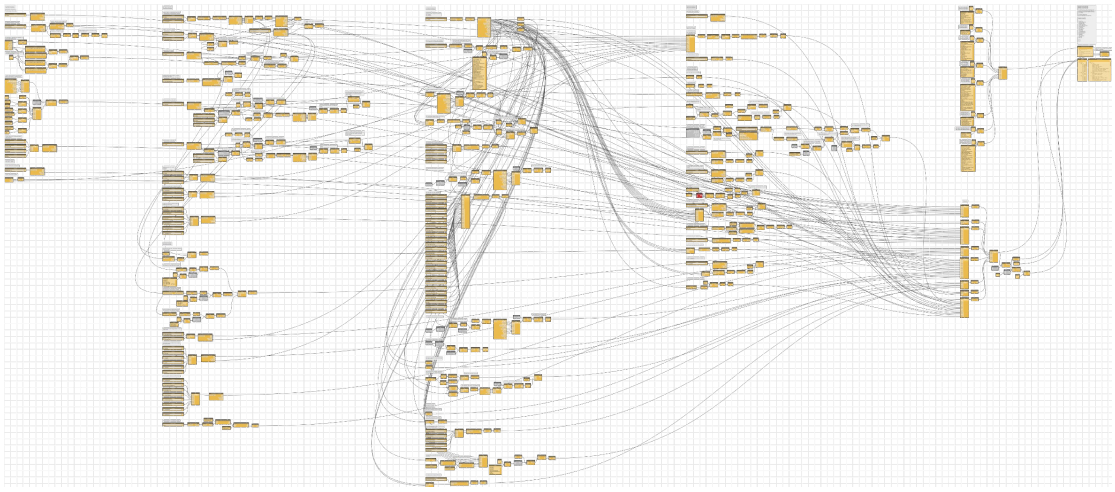

Fig. 49: Dynamo-based model checker for final submission of introductory Revit class projects at Chair for CAAD, RWTH Aachen University

And finally: I am an architect, not a full-fledged programmer. And I don't plan to become one any time soon. My personal Dynamo journey which started two years ago has definitely had its highs and lows – especially with the move from Dynamo 0.6.3 to 0.7x – but it has always been a constant and rewarding learning experience. During that time, I learned enough about coding with the Revit API in order to be able to make Revit do the things I want it to do. I like the Dynamo community. And the one thing I probably like best about Dynamo is the package manager. If you don't want to share an entire workflow you can still share the components it's made of. We don't all need to be geniuses. But pooling our resources might help a little.

And most importantly, to quote Sean Burke[85]: *"Dynamo excites me in a way Revit did ten years ago."*

84 Andreas Dieckmann & Peter Russell, "The Truth Is In The Model – Utilizing Model Checking To Rate Learning Success in BIM Software Courses", 2014, in: "Fusion – Proceedings of the 32nd eCAADe Conference Vol. 2", pp. 417-426, Newcastle/UK. Presentation (including zoomable Dynamo graph) can be found here: http://prezi.com/kaagf4w6iq-d/the-truth-is-in-the-model/

85 https://twitter.com/seandburke/status/449982913082191872

Further Reading

If you plan on using Dynamo for automating menial tasks, there is more material available online. If you haven't done so yet, I would recommend you follow the blogs of Julien Benoit[86] and Konrad Sobon[87]. Julien and Konrad have both been blogging about all kinds of useful automation you can achieve with Dynamo for a while now.

Data Sets

After the conference, all workflows shown here will be available on GitHub:

https://github.com/CAAD-RWTH/ClockworkForDynamo/tree/master/workflow_samples/rtceur2014

Some of the workflows shown here currently do not work in Dynamo 0.7x yet. If you want to be notified of an updated (0.7.x) version of these workflows, just follow me on Twitter (@a_dieckmann) or contact me via email (dieckmann@caad.arch.rwth-aachen.de). I will announce an updated version as soon as it becomes available.

86 http://aecuandme.wordpress.com/
87 http://archi-lab.net/