Generative Design for Building Information Modeling

[Extended Abstract]

Bruno Ferreira
Instituto Superior Técnico
bruno.b.ferreira@tecnico.ulisboa.pt

ABSTRACT

Generative Design (GD) is a programming-based approach for Architecture that is becoming increasingly popular among architects. However, most Generative Design approaches were thought for traditional Computer-Aided Design (CAD) tools and are not adequate for the Building Information Modeling (BIM) paradigm, which is becoming mandatory in several countries. Rosetta is a portable GD environment that allows users to choose both the programming language in which they want to program and the CAD application in which they want to produce their model. Due to the usefulness of BIM there is a desire to explore BIM with GD, so we propose a solution that expands Rosetta. This solution allows the portable usage of BIM features, which does not limit the user to a specific Building Information Modeling (BIM) tool. To evaluate our solution, we combined Rosetta with Revit and developed several models as case studies, that show the usefulness of our approach.

1. INTRODUCTION

Computer-Aided Design (CAD) applications increased the efficiency of design activities and allowed architects to produce more accurate and precise drawings that could be more easily edited without the need of manually erasing and redrawing parts of the original design.

Nevertheless, modeling complex and creative geometry can still be a challenge in a CAD tool and changing the model continues to present some difficulties as the degree of flexibility provided by these tools is not sufficient. Generative Design (GD) is used as a solution to these problems [1]. GD can be described as form creation through algorithms [2]. This approach allows the generation of different solutions just by changing the constraints and the requirements implemented in a given program [3]. These programs can include complex algorithms that generate geometry that is very difficult to create by manual means.

Recognizing the advantages of the GD approach, many tools were developed that allow the creation of GD programs. These tools were also tailored for architects with basic programming experience, reducing the programming skills needed to use them.

Rosetta is one of those tools. It provides a programming environment for GD, supporting the development of scripts using different programming languages. These scripts can then be used to generate models in different CAD tools. Nowadays, BIM tools are replacing the traditional CAD tools. Firstly, because they offer a set of features that go beyond CAD, and, secondly, because many governments are making the use of BIM mandatory.

GD tools like Rosetta were developed for CAD applications, and thus, are not adequate for the BIM paradigm. This happens because CAD tools only deal with geometry while BIM objects are far more than that. They are defined by the parametric rules they contain as well as by their properties like materials, finishes, manufacturer specifications and even price.

These properties allow BIM tools to detect problems in the design, such as a pipe that is placed in the same location as a window. The parametric rules allow the object to adapt to its usage when inserted in a project.

Finally, all the information about the project life cycle is stored in these objects and can be used to create documents related to fabrication, cost estimation, and even building management [4]. CAD tools do not require this information but BIM tools do. This fact alone changes the way users interact with BIM and also changes how GD programs have to be written. GD tools that, currently, only work with CAD tools must provide all these BIM-specific features in order to communicate with a BIM tool but most of these GD tools do not have the proper support for that.

Developing a GD program for BIM is a problem that exists nowadays and the solution offered by BIM tools, like Revit, is an Application Programming Interface (API) that provides a way to use BIM with a programming approach. Unfortunately, the use of the API requires knowledge of programming languages, like C# or C++, and computer science concepts, such as transactions and polymorphism, which assumes considerable programming experience.

The objective of this thesis is to present a solution that allows architects with some programming experience to write GD programs for BIM applications.

2. RELATED WORK

Several tools were analyzed to guide us in the development of our solution. The main focus was on tools that allow users to write GD programs for BIM applications. This includes tools that are already available in BIM applications as well as plug-ins developed for the same purpose.

2.1 Grasshopper 3D and Lyrebird

Grasshopper 3D is a graphical programming language developed for architects as a plug-in for Rhinoceros 3D CAD. Programs written in this language represent a data flow graph that consists of a group of components and the connections between them. These components can be selected...
DesignScript is a programming language that is heavily influenced by design principles. It was created by Robert Aish to be, not only a production modeling tool, but also a full-fledged programming language and a pedagogical tool [7].

As a programming language, DesignScript is seen as an associative language as it maintains a graph of dependencies between the variables used in a program. Any change in one of the variables is propagated throughout the program. This means that if we have a variable \( a \), and if \( b \) is defined as \( a + 1 \), a change in \( a \) will also modify the value of \( b \). This is a change-propagation mechanism, similar to the update mechanisms available in associative CAD tools. The language is also a domain-specific language as it contains primitive operations for design and geometry.

As a modeling tool, DesignScript tries to introduce concepts that are easily understood by users that are not accustomed to design with the help of a programming language. This is achieved by allowing the developers to use its logical framework in order to produce the design models, and also by facilitating an exploratory approach to the tool involving refactoring of the produced models [7].

Finally, DesignScript aims to be a pedagogical tool, as it allows the evolution of the programming skills of its users. Users unfamiliar with programming are able to use a direct approach with a graph node diagramming interface that is simple and requires little to no understanding of programming concepts [8]. Figure 3 shows a program created with the graph approach. However, as their design becomes more complex, users might feel the need to learn more advanced programming concepts. The node-to-code functionality of DesignScript allows a transition between the graph representation and a script that initially presents a logic very similar to the original graph. For users that desire to produce more complex programs, this script might be changed into a normal script.

### 2.3 Dynamo

Dynamo is a plug-in for Revit that is strongly influenced by graphical programming languages such as Grasshopper for Rhino. Just as Grasshopper, users create a workflow by introducing nodes that are connected to each other through wires associated with the ports that each node contains. A port from an element can only be connected to another port of a matching type. This means that the input and output port must have compatible types. Nodes can represent several Revit elements, such as lines, or functions, such as

![Figure 1: An example of a complex parametric program in Grasshopper.](Image)

![Figure 2: Beams being created with Lyrebird using lines from Rhino.](Image)
mathematical functions. Users can also define custom nodes in order to extend the functionality provided by Dynamo [9]. Figure 4 shows a program written with Dynamo nodes and its result.

However, GDL only allows the creation of simple geometry and does not take advantage of the BIM functionalities of ArchiCAD.

2.5 GenerativeComponents

GenerativeComponents (GC) is a parametric and associative system developed for Bentley’s Microstation.

This system is propagation-based so the user has to determine the rules, relationships and parameters that define the desired geometry. This propagation-based system consists of an acyclic directed graph that is generated by two algorithms: one that is responsible for ordering the graph and the other that propagates values through it [11].

GC has several ways of user interaction, taking into consideration his skills. The first one is a Graphical User Interface (GUI) that allows direct manipulation of geometry. The second one is by defining relationships among objects with simple scripts in GCScript. The third and final one, is by writing programs in C#, allowing the definition of complex algorithms. GC shows that a graphical language might be easier to learn but as the user wants to produce more complex models, he will start to produce scripts in a textual programming language.

2.6 RevitPythonShell

RevitPythonShell is a plug-in developed for Revit that allows users to take advantage of the RevitAPI but using Python. This tool was implemented to simplify the workflow that is needed in order to develop with the RevitAPI [12].

This plug-in embeds IronPython as language and uses a code editor provided by Python Tools. With this editor, developers have access to a Read-Eval-Print-Loop (REPL) that will let them experiment the API in an easier approach. They only need to type a statement, hit enter, see the results and carry on [12].

2.7 Hummingbird

Hummingbird is a tool that allows the usage of Grasshopper programs to facilitate the creation of Revit’s native geometry [13]. This tool consists of a set of Grasshopper components and a Revit plug-in, and the workflow generates files in a language similar to Excel, where several tables describe the geometry to be created in Revit.

This plug-in allows the user to translate Rhino’s geometry to Revit. However, this tool introduces a number of components that take advantage of Revit’s BIM features, such as Levels, Masses and Topographic Surfaces (also known as TopoSurfaces).

Hummingbird is in many ways similar to Lyrebird, which also takes advantage of Grasshopper to communicate with Revit. Yet, Hummingbird allows a bi-directional communication between Revit and Rhino, that is based on CSV files [13]. These files are generated during the tool’s workflow and can be read or written by a special component.

2.8 Rhino-Grasshopper-ArchiCAD

Rhino-Grasshopper-ArchiCAD is a tool that allows users to combine these three tools in a single project. This tool is currently in beta testing [14].

By introducing special components in Grasshopper and a plug-in in ArchiCAD, users can exchange information between both applications in order to develop an algorithm that produces the desired outcome.
Table 1: A comparison between the analyzed tools, showing what kind of language support they have and their BIM support. The "✓"-" symbol indicates that it supports textual languages but only with small scripts.

For example, it is possible to create a curve in Rhino, extract the points that define it with Grasshopper, manipulate them, and use them to create BIM elements in ArchiCAD.

Since this tool is still in beta testing, it is not possible to know everything about its features and how it compares with other tools. However, the development of Rhino-Grasshopper-ArchiCAD shows the current need of a way to use GD with the BIM paradigm as well as the desire of making BIM a friendlier design tool for architects [14].

2.9 Comparison

Table 1 shows a comparison between the analyzed tools. As it is possible to see, most of these tools make use of visual programming languages, due to their easy learning process and simplicity of use. Although most of them also offer ways to use textual programming languages, they have limitations, since they only allow the creation of small scripts and do not offer access to all the BIM tool’s functionality.

It is also possible to see that most of them already offer support to BIM applications, which shows that GD for BIM is a real need. However, as seen in table 1, only three of the analyzed tools offer full support of the BIM application that they are connected to.

Finally, we can see that, despite some of these tools are already working with BIM applications, they still offer Geometric Operations, due to their usefulness when working with complex geometry.

2.10 Problems to Address

One disadvantage of all the analyzed tools lies in the fact that they use visual programming languages. Although easy to learn, visual programming languages do not scale well with the complexity of the programs, becoming difficult to use when programs grow [15].

Also, the textual language support these tools offer is not enough, since most of them offer support only to small scripts. Adding to this, almost all of them use programming languages that are inappropriate for beginners, due to being too complex, like C#, or obsolete, like BASIC.

Finally, all these tools are connected to only one application and work with specific programming languages. If the user wants to change to other application, his program is no longer viable, forcing him to write a new one from scratch.

These are some of the problems we solve with our approach.

3. GENERATIVE DESIGN FOR BUILDING INFORMATION MODELING

In order to allow users to develop programs that scale well, we decided to have a solution that uses textual programming languages. However, since these are typically harder to learn, we use languages that have known pedagogical qualities, like Python or Racket. Also, we want to provide BIM operations that are easy to use and understand, as well as portable between BIM applications.

For these reasons we divided the application in three components: (1) an Integrated Development Environment (IDE) where the user writes his programs; (2) an Abstraction Layer that provides all the necessary primitive operations and abstractions; and (3) a plug-in that allows the connection with BIM applications.

The architecture of our solution can be seen in figure 5.

Figure 5: Architecture of the Solution

In the next sections we explore each of the three components.

3.1 Integrated Development Environment

Our solution uses a programmatic approach based on textual programming languages since they are more flexible and scale better with the program’s complexity. For this reason users need an editor in which they write their code.

The language of the editor must be fit for a beginner. For example, the Python language is very popular and is being used to teach beginners how to program. Also, languages like Python have an IDE that helps users write and debug their programs. These features are very important for beginners since they have difficulties writing their first programs and these IDEs help them overcome some initial barriers.

For these reasons, we decided to use the same programming environment provided by Rosetta, since it is designed for beginners.

By using Rosetta, users are able to choose the language in which they want to program. Racket, Python, Javascript and Processing are some examples of languages that are already available. This helps users to overcome the need for
learning a new programming language if they already know one that is available. If they have yet to learn how to pro-
gram they can choose any of the languages available, since
most of them are pedagogical languages. The Rosetta IDE
also supports debugging and syntax highlighting for all the
languages available as front-ends.

3.2 Abstraction Layer

To write GD programs, users need functions that give ac-

cess to BIM functionality. These functions must allow users
to instantiate BIM objects and define all the information
needed in order to produce an accurate project.

For example, to produce a wall with a door in it, there
must be first a function that creates walls. This function
might receive the height, length, position and the type of
wall in order to create the correct BIM object. Then, a
function that creates the door is used and it must receive,
not only the position, height and type of door but also the
wall that will serve as host of this object.

In addition to this, to indicate positions and create lines
and arcs, abstractions for these concepts are needed as well
as functions that allow their creation and usage.

All these functions as well as others that create and ma-
nipulate BIM objects are included in an abstraction layer,
so that they can be used in all supported BIM tools.

However, some features are unique to certain BIM tools,
so we will offer them as functions that will be available only
to that specific BIM. The user can choose to use them,
giving up the portability of their programs, but gaining the
ability to take advantage of the specific features the tool has
to offer.

In order to implement this abstraction layer, we decided to
expand the abstraction layer currently available in Rosetta.
Rosetta already has abstractions for coordinate systems and
geometric operations that can be adapted to the BIM back-
ends, eliminating the necessity to implement all functional-
ity from scratch.

3.3 BIM Plug-In

Finally, in order to execute the users’ programs and gen-
erate the result in the desired BIM tool, a component that
communicates with the tool is needed.

Since most of the BIM tools have an API that exposes
their features in a programatic way, this component can be
a plug-in for the tools written with the aid of that API.

All the functions of the abstraction layer have a corre-
spondent one in this component. When one of them is used,
the needed information is sent to this component and then
the correspondent function is executed. By taking advan-
tage of the API, these functions can produce their results in
the BIM tools.

With our solution, we introduce Revit as the new BIM
back-end, so we developed a plug-in using the RevitAPI.

This plug-in is written in C# and was developed using the
Microsoft Visual Studio programming environment. The ab-
straction layer previously mentioned was developed to hide
the complexity of the API, providing primitive operations
that are easy to understand by beginners. These primi-
tive operations not only introduce abstractions that are well
known by the users, like walls and slabs, but also manage
the necessary transactions needed by the API to create and
visualize objects.

Figure 6 shows a diagram that depicts the invocation of
a BIM primitive operation and how each of the described
components work to produce the results.

4. PROGRAMMING FOR BIM

Our solution allows users to create GD programs with the
BIM paradigm, which is very different from the CAD one
that was previously used with Rosetta.

The differences between them not only impact how users
use the tools that support the paradigm, but also how the
programs are written for them.

In the next sections we explore these differences.

4.1 Semantics

When working with the BIM paradigm, the objects that
are used contain, not only geometry, but also information
about its properties, like price, materials and supplier. Some
of these properties allow the detection of collisions or struc-
tural errors in the model, something that is not easy to do
in CAD models.

Even if two objects have the same geometry, they might
be semantically different for the BIM tool. For this rea-
son, programs written for the BIM paradigm have to use
primitive operations that carry these semantics. Instead of
primitive operations like box that were used to create slabs
and walls in CAD models, programs for BIM use primitive
operations like create-slab and create-wall, that include
not only geometric information as input, but also additional
information, such as the type of slab or wall that is to be
instantiated.

4.2 Dependencies

BIM objects have more properties besides their parame-
ters. Several objects are connected to other objects, which
introduces dependencies between them.

A wall’s height depends on the elevation of the level in
which it is placed, and some objects depend on a host to
exist, like a door that needs to be placed on a wall.

These simple restrictions must be present in the GD pro-
gram that creates the objects, as well as in the primitive
operations available. For these reasons, primitive operations like `create-slab` need to receive a level as input.

Since objects need to be used as input, users have to write the code in a way that allows objects to be created in the correct order. The dependencies impose a structure in the code which did not exist when creating programs for the CAD paradigm, where the user was free to create objects in the order he wanted.

### 4.3 Simplification

In the BIM paradigm objects are usually associated in groups due to the semantics they possess. In Revit, for example, objects are grouped in families, that share common features, such as material and their overall form.

These are presented as libraries to the user. So, when creating the object with GD, the user can specify the type of object they want to use. Previously, in the CAD paradigm, this was not possible, and the geometry of the object needed to be implemented in the program as well.

This has a serious impact not only in the development effort but also when it is necessary to change the program. Previously, if the user wanted to change the shape of an object, he had to re-implement it in the code, while with the BIM paradigm, it is only necessary to use a different type of object as input. This simplifies the code, which in turn simplifies future changes. However, the complexity of the primitive operations might grow, due to all the possible information that could be added to certain objects.

### 4.4 Iterative Development

Creating BIM models has advantages when compared to CAD models due to the additional information that the model contains. However, as mentioned previously, this might introduce complexity in the primitive operations, since they might require a high number of parameters as inputs.

This is not only a disadvantage of the programmatic approach, but of the BIM paradigm itself, since in many situations users do not know all the information that is needed in the beginning of their projects.

To solve this problem, we implemented the primitive operations as functions with keyword parameters using default values. The default values are used when nothing is passed as input to a given parameter, which allow users to create objects of a default BIM family when they are not certain about the specific object they want or what properties to use.

As the project advances and more decisions are made, users can replace the default values for ones of their choosing. This allows for an iterative development, giving users the flexibility to start developing with as little details as possible, which can be incremented as they explore alternative designs.

### 4.5 From CAD to BIM

In the previous sections we compared the new BIM approach to the CAD one, since most users would have used the latter one. As such, it is then important to reflect on how these changes will affect programs previously developed for CAD, as well as the transition to BIM.

The first big difference is the usage of objects instead of geometry and the inclusion of semantics in the code. Although this seems like a big difference, most users already took advantage of intermediate abstractions to make the code more legible. These abstractions gave objects semantics, introducing concepts like walls and slabs in the code. For these users the transition to BIM is not as difficult, since the semantic information was already in the code and they only need to replace their abstractions with the new primitive operations.

Secondly, the dependencies and the structure of the program. The new paradigm imposes a well-defined structure in the created programs, which did not happen in CAD. However, due to the intermediate abstractions previously mentioned, the users’ code might already have a structure similar to the one needed to use the BIM paradigm. Nevertheless, there are certain elements, like levels, that would rarely be present in programs for CAD, and are mandatory now.

Finally, certain operations are no longer valid in BIM models, namely boolean operations. The usage of subtractions and intersections was extremely common when working with the CAD paradigm to unite elements and to create openings to insert elements. These operations are not available when working with the new paradigm, but they are also not necessary in many cases. For example, when we create a door it is necessary to use a wall host as input, and, as a result, the door is correctly introduced in the element, with the correct opening in the wall automatically done. This solves most of the problems, but it is important to note that boolean operations might be necessary to create complex shapes, and their absence might make this difficult in BIM. For these reasons, we include some of these boolean operations for BIM.

We can conclude that, although the difference between paradigms are considerable, good programming practices might help the transition and, in some cases, the user’s program might even be simplified. The dependencies between object might force users to restructure their programs, but the new structure follows a logical and architectural organization of elements, something that might be partially present in programs where users introduce these concepts as intermediate abstractions.

### 5. EVALUATION

In order to evaluate our solution, we used the new back-end we developed for Rosetta, Revit, which now allows users to write GD programs for the BIM paradigm. Using the new back-end, we developed a set of case studies that we present in the next sections.

#### 5.1 BIM features

As we mentioned previously, the biggest change with BIM is the amount of information available in the model due to its objects. So, users must be able to take advantage of BIM objects, in order to produce a correct model.

To test if this was possible with our solution, we used a model that was previously developed by architecture students for CAD as a case study: the Dubai Towers. This model was produced by a GD program but contained only geometry, so we tried to produce it in the new paradigm.

This leads to a model in Revit that can be seen in figure 7. This model is entirely composed by objects that were created with the primitive operations we introduced. This shows the capability of our solution regarding the creation of objects.
Adding to this, the program creates every slab associated with each level, something that is mandatory in Revit as well as other BIM applications. This shows that, not only can we reproduce the dependencies and associations created when interacting with the application with the GUI, but we can also produce a model with the correct properties.

5.2 Simplification

Another feature we gain when working with BIM is the ability to work with a library of objects. Since every element in a BIM model is an object with parameters, such as price, manufacturer and materials, there are several libraries available, some of them provided by suppliers.

In Revit, these objects are organized into families, some of which are installed with the application, while others can be downloaded from suppliers. Since every object has a family, the parameters of that family define the geometry of the object, and changing the family can completely change the object.

This can simplify users’ programs since they no longer need to define the object’s geometry. If the user wants to create a door in their program, they do not need to implement every component related to its shape and properties and can simply give a family as input. The code seen in listing 1 illustrates how simple a program that creates a door can be.

Listing 1: Racket code to produce a door in Revit.

```racket
(define door-family
  (load-family "C:\M_Single-Glass_3.rfa"))

(define door-default-element (family-element door-family))

(define w1 (create-wall (list (u0) (xyz (at 10) 0 0))))

(insert-door-relative (car w1)
  (at 5)
  0
  #:family door-default-element)
```

The user simply defines the family he wants to use, as well as the member of that family. To change the door, the user only needs to change the element or the family, which is much simpler than redefining the whole object.

5.3 Scripting

One of our objectives was to create a solution where scripting would be simple, with a programming language that is appropriate for beginners.

To evaluate this, we created a simple test that produces a matrix of box objects, and compared the necessary code to produce it with the code that the RevitPythonShell would require. We used the RevitPythonShell as a comparison element, since it uses a textual programming language that is also suited for beginners, Python.

When we compared the solutions, it was possible to see that our solution required less programming skills and was much simpler. RevitPythonShell required creation and management of transactions, as well as a high number of intermediate calculations, while our solution hides all that complexity from the user. Listing 2 shows the code written for Rosetta to produce the matrix for the test. It is possible to see that the code for Rosetta is simple, and does not require the management of transactions like RevitPythonShell.

Listing 2: Racket code for the test.

```racket
(for/list ((i (range 10)))
  (for/list ((j (range 10)))
    (box (xyz (+ 0 (* i 11)) (+ 0 (* j 11)) 0) 10 10 10)))
```

Adding to this, we also supply a REPL just like RevitPythonShell, that allows for an easier exploration of primitive operations and solutions.

5.4 Portability

When we analyzed the several tools in our Related Work, we pointed out that they were not portable, a feature that would be highly useful for users to explore several tools and alternative solutions.

Portability was the main objective of Rosetta, when it was developed for CAD tools, and since our solution expands it to BIM tools, we also wanted our solution to be portable as well.

All the primitive operations introduced were created with portability in mind, dealing with concepts that are introduced by the BIM paradigm and not of a single application. To test this, we used a case study developed by an architect, a model of the Absolute Towers, and ported her program to BIM back-ends. That program was then used with the Revit back-end, as well as with another BIM back-end that is currently being developed for ArchiCAD.

The same program produced equivalent models, as can be seen in figure 8.

In addition to portability between BIM tools, we also achieved a certain degree of portability between CAD and BIM tools. By implementing simplified versions of the developed BIM operations, which are similar to the intermediate abstractions that some users created, it was possible to create programs that would work in all back-ends. Figure 9 shows a building with sinusoidal facades, produced by the same program in all the available back-ends: OpenGL, Sketchup, Rhinoceros 3D, AutoCAD, Revit and ArchiCAD.

This feature allows users to take advantage of faster back-ends to explore several solutions and then change to a BIM
Figure 8: Absolute Towers, produced by a program in Revit and ArchiCAD.

Figure 9: Building with a sinusoidal facade, produced in all back-ends.

back-end to produce a more detailed model, with all the relevant information.

5.5 Performance

Although performance was never a main concern, as it is highly dependent on the performance of the back-ends themselves, we decided to test the performance of our solution. First, we used the model seen in figure 9 to see the performance of all back-ends. The results can be seen in figure 10.

Figure 10: A chart that shows the time to produce a building with a sinusoidal facade in several back-ends.

As it is possible to see, the BIM back-ends are slower than the others, possibly due to all the additional information that the model contains. This reinforces the need of portability, even with non-BIM back-ends, in order to quickly explore variations in an architectural model.

We then tested the performance of our solution in comparison with the RevitPythonShell, due to its similarities with our solution. We used a simple test that produced a matrix of cubes with variable number of elements. The results can be seen in figure 11.

Figure 11: A chart that shows the time to produce a matrix of cubes with 10x10, 15x15 and 30x30 elements, in RevitPythonShell and Rosetta.

As it is possible to see, our solution is approximately 5 per cent slower than RevitPythonShell, an acceptable trade off for all the flexibility and simplification that our solution offers.

Finally, we compared our approach with the traditional way of interacting with the BIM tool, through the GUI. One of the main objectives of this thesis is to give a programmatic approach for GD, since it simplifies and accelerate several phases of the development, so we must see if this goal is achieved. To evaluate this, we asked a Revit expert and an architect that is also a beginner programmer to develop the same model in a manual and programatic approach, respectively. The results can be see in table 2. As it is possible to see, the initial model took longer to develop in the programatic approach. However, when it is necessary to change the model, the programatic approach becomes much faster, since a simple change to a parameter value, might produce the desired modification, while the manual approach might require the user to change all the elements one by one.

Table 2: Time taken to produce and change a model of the Absolute Towers with the BIM Manual Approach and the Programming-Based Approach.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Manual</th>
<th>Programmed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production of the model</td>
<td>2h14min</td>
<td>14h</td>
</tr>
<tr>
<td>Change the slabs shape</td>
<td>1h25min</td>
<td>10min</td>
</tr>
<tr>
<td>Increase the number of floors</td>
<td>48min</td>
<td>7min</td>
</tr>
<tr>
<td>Change the slab size with height</td>
<td>1h52min</td>
<td>4min</td>
</tr>
</tbody>
</table>

5.6 Real Case Application

During the development of this thesis, we received a request from an Architecture Studio to produce a facade in Revit for one of their buildings. So, we took advantage of this request to test our solution.
Figure 12: Facade generated in AutoCAD, using Rosetta.

Figure 13: Facade generated in the new Revit back-end.

The concept was already decided, but the overall look of the facade was not decided yet, which signified that we were free to experiment and explore different designs.

Using our tool, an architect with programming skills started by developing a program that produces solutions in AutoCAD. Figure 12 shows one of the models generated in AutoCAD. Given that AutoCAD allows faster visualization than Revit, starting to program the facade in AutoCAD allowed the architect to quickly generate and evaluate different solutions.

When the overall design of the facade was defined, the development was changed for Revit. Since the program was portable between CAD and BIM, we did not need to do any major changes to the program, and only needed to change the back-end. The program produced the result seen in figure 13, in Revit.

Finally, with the facade generated in Revit, we delivered the project file to the Architecture Studio, so that they could incorporate the facade in their model. A render of the building with the facade can be seen in Figure 14.

This simple example allowed us to test our solution in a real case scenario, showing not only how the workflow of the architects is influenced by the use of Generative Design tools, but also how useful it can be to explore and finally produce an abstract element with BIM objects that would be very difficult to produce by manually using the GUI.

6. CONCLUSIONS

GD was introduced in Architecture as a new way to explore complex solutions using programming. Recognizing the potential of GD, several tools were introduced to explore programming in architecture, such as Grasshopper and Rosetta.

However, nowadays CAD tools are being replaced by Building Information Modeling (BIM) tools, creating the need for new GD tools that explore this paradigm. Several GD tools already explore this paradigm, such as Lyrebird, Dynamo, or Geometric Description Language. Some of these tools even offer several ways of interaction, like GenerativeComponents, while others are concerned with easy-to-use textual programming languages, such as RevitPythonShell.

The demand of GD environments for the BIM paradigm is a reality, and new tools are constantly being announced, such as the recent Rhino-Grasshopper-ArchiCAD connection. However, many of these environments do not offer flexibility or scalability, due to the usage of visual programming languages, while others are not fit for beginners due to complex or obsolete programming languages. In addition, none of the mentioned environments offer portability, locking developers to one language and one BIM application.

For these reasons, we propose an extension of Rosetta that introduces BIM tools as back-ends. This solution is based on: (1) an IDE that supports several textual programming languages, appropriate for beginners; (2) an abstraction layer that makes several primitive operations developed for the BIM paradigm available for users; and (3) a plug-in that connects the development environment to the desired BIM tool, in order to produce the models. With this architecture, we were able to introduce Autodesk Revit as the first BIM back-end.

The introduction of BIM back-ends led to a new way of creating GD programs that is considerably different from the way users write programs for CAD. The BIM paradigm introduces objects with semantics and dependencies between each other, in order to produce a model with all the correct information. By introducing appropriate primitive operations in the abstraction layer, we implemented these features in Rosetta, which in turn are reflected in how a GD program must now be written. Also, by having these features we can simplify the code the user has to write, since it is possi-
able to use pre-modeled BIM objects, available in the tools’ libraries. By having these objects, users do not need to implement all the objects’ geometry in the code. Finally, since developing for the BIM paradigm is more complex than for the CAD paradigm, we made iterative development possible through the primitive operations, by having several optional parameters that can be specified as the project grows. Furthermore, through the IDE’s REPL it is possible to easily explore alternatives and test the program as it grows. All these features can also make it easier to transition between the CAD and BIM environments.

In summary, our solution is able to explore BIM tools, maintaining the features that lead to the development of the original Rosetta. We also introduced a certain degree of portability between CAD and BIM paradigms, allowing for an easier transition for users of the original Rosetta, as well as a bigger flexibility to explore alternative designs through an iterative development. Rosetta is now able to accompany the demands of the GD community with support for BIM tools, which can easily continue to be developed and expanded.

7. FUTURE WORK

Currently, there is ongoing work developing another back-end, ArchiCAD, by exploring our architectural solution. In the future, we will introduce this and more BIM applications as back-ends, namely Bentley’s MicroStation and Digital Project.

Regarding our current back-end, we will continue to introduce more BIM features that are currently not available, as well as new primitive operations that will make more objects available to users. One of the features that we will develop is the creation of new families. Currently, we partially support this when a mass wall is created, but we want to extend this to allow the definition of the objects shape with geometric operations and then the definition of parameters and rules for the new object.

We will also continue to explore the combination of the CAD and BIM paradigm. Since CAD back-ends are efficient when producing geometry, we want to be able to produce complex geometry in a CAD back-end, read geometric data from it, and then change to a BIM back-end in order to use this information to instantiate complex BIM objects. This workflow is something we want to allow users to explore, and we are already working to make it possible.

Finally, we will also explore ways to introduce optimization in Rosetta, allowing users to combine it with their BIM programs to achieve solutions that meet demanding requirements. There are several plug-ins for Revit that evaluate and produce values that would optimize the model, and one of our objectives is to connect both plug-ins in order to have a program that iteratively tries new variations with the values produced by the optimizer.

8. REFERENCES