Generative Design for Building Information Modeling

Guilherme Henrique Aciaioli de Sousa Barreto

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão
Members of the Committee: Prof. Daniel Jorge Viegas Gonçalves

November 2016
Acknowledgments

I would like to thank my parents and my brother for their friendship, encouragement and support throughout my life.

I would also like to thank my colleagues and friends of the GAC group that made this project possible. In special, Bruno Ferreira and Sofia Feist due to their closer support and fellowship.

A special thanks to professor António Leitão for giving me the opportunity to be part of the GAC group, and for his insight, support and sharing of knowledge in guiding me through the development of this thesis.

Finally, I would like to thank Fundação para a Ciência e Tecnologia (FCT) and to INESC-ID for attributing me a research grant, with reference UID/CEC/50021/2013.

To each and every one of you – Thank you.
Abstract

Generative Design (GD) is an algorithmic-based approach to design that allows the generation of forms and shapes through algorithms. It has been vastly explored with the Computer-Aided Design (CAD) paradigm, but not to the same extent with the Building Information Modeling (BIM) paradigm. Therefore, we propose a solution that allows the exploration of GD using the BIM paradigm, taking full advantage of its capabilities. The solution is an extension of Rosetta, a GD environment that supports a wide range of back-ends, namely, OpenGL and CAD applications. We expand Rosetta’s abstraction layer with modeling operations capable of producing models on a BIM application, ArchiCAD, having into consideration portability between the already supported back-ends. Furthermore, alongside the development of our solution, another BIM application was also being added to Rosetta, Revit, allowing us to test portability between the two back-ends. Finally, we evaluate our solution in terms of its adequacy, portability, performance, and support of ArchiCAD-specific operations.

Keywords

Programming; Architecture; Generative Design; Computer-Aided Design; Building Information Modeling; Portability; ArchiCAD.
Resumo

Generative Design (GD) é uma abordagem de desenho baseada em algoritmos que permite a geração de formas. Tem sido vastamente explorada em conjunto com o paradigma de Computer-Aided Design (CAD), mas não tão extensivamente com o paradigma de Building Information Modeling (BIM). Com isto em mente, propomos uma solução que permite a exploração de GD usando o paradigma BIM, tirando partido das suas capacidades. A solução é uma extensão do Rosetta, um ambiente de desenvolvimento para GD que possui um leque abrangente de back-ends, nomeadamente, OpenGL e aplicações CAD. Expandimos a camada de abstracção do Rosetta com operações de modelação capaz de produzir modelos numa aplicação BIM, ArchiCAD. Durante o desenvolvimento da nossa solução, um outro back-end BIM também estava a ser adicionado ao Rosetta, o Revit. De tal forma, tivemos em consideração a portabilidade não só entre os back-ends já suportados pelo Rosetta mas também com o Revit. Finalmente, avaliamos a nossa solução em termos de adequação em produzir modelos BIM, portabilidade, desempenho, e suporte de operações específicas do ArchiCAD.

Palavras Chave

Programação; Arquitectura; Generative Design; Computer-Aided Design; Building Information Modeling; Portabilidade; ArchiCAD.
# Contents

1 Introduction 1
   1.1 Computer-Aided Design & Building Information Modeling 3
   1.2 Generative Design 5
   1.3 Objectives 6
   1.4 ArchiCAD & Revit 6
   1.5 Summary 8

2 Related Work 11
   2.1 Grasshopper 13
      2.1.1 Lyrebird 13
      2.1.2 Hummingbird 14
      2.1.3 Rhino-Grasshopper-ArchiCAD 14
   2.2 Dynamo 15
   2.3 RevitPythonShell 16
   2.4 Generative Components 16
   2.5 ArchiCAD API 17
   2.6 Geometric Description Language 20
   2.7 Analysis 21
   2.8 Summary 22

3 Architecture 23
   3.1 Rosetta 25
   3.2 Abstraction Layer 27
   3.3 ArchiCAD’s Plug-In 29
   3.4 Communication Channel 29
   3.5 Interaction 29
   3.6 Summary 30
List of Figures

2.1 A Grasshopper program that shows how the readability and maintainability of a program become an issue when dealing with complex problems. (source: www.digitalcrafting.dk) ........................................... 14
2.2 A Grasshopper program that uses the Lyrebird plug-in. The added component is in green, and the result is seen on the right. (source: http://www.food4rhino.com/) 15
2.3 Columns created using R-G-A. The green component is the one that creates the columns in ArchiCAD, the others are required to obtain the points of each column. 16
2.4 Example of a program in Dynamo. It is possible to see the nodes and the wires that connect them. (source: http://dynamobim.com) ......................... 17
2.5 An example of a program in GC that manipulates the position of points. (source: http://communities.bentley.com) ............................. 18
2.6 A GDL script that shows how the 3D model of a chair can be created. .............................. 20
3.1 A diagram depicting the overall architecture of the solution. ..................... 25
3.2 A sequence diagram that shows the creation of a door in a wall. ............... 31
4.1 The door produced by the code seen before. ....................................... 39
4.2 The door on the left was created using the default type of door. The other two doors were created by specifying the type of door but still using the same parameters as before. ................................. 40
4.3 We changed the length of the railing modifying an attribute of the element. In order to adjust to the new dimension, the element automatically created additional posts. ...................................................... 41
4.4 Due to the limited set of attributes that the railing has, we are not able to apply an angle only to the top handrail. The only attribute available applies an angle to the overall railing. ...................................................... 42
4.5 By taking advantage of our library part we are able to change the top handrail’s angle by changing a single parameter.

4.6 A complex project that was created by traditional means.

4.7 The result of making a selective change to the walls of the project.

4.8 A simple tower, which is the result of our program that manipulates levels.

4.9 The walls created from the code above, which use different materials.

4.10 A profile being applied to a wall.

5.1 The tower produced by Rhino-Grasshopper-ArchiCAD (R-G-A).

5.2 On the left: vertical beams. On the right: columns.

5.3 Our version of the example tower.

5.4 The Absolute Towers of Canada, developed by the MAD Architects. (source: https://www.daniels.utoronto.ca)

5.5 Slabs of the two Absolute Towers.

5.6 On the left: the floor slab, the inner-slab and the walls. On the right: the result of intersecting the walls with the inner-slab.

5.7 The structural walls of an Absolute Tower, showing how the intersection with the inner-slab produces the final shape.

5.8 A render of a slice of the building, showing some of the added details, such as stairs, landings and railings.

5.9 A render of the Absolute Towers created by the program developed for the case study.

5.10 A render of two towers that have different slab shapes.

5.11 A render of the building with a sinusoidal facade.

5.12 The produced model in ArchiCAD (left) and in Revit (right).

5.13 The Absolute Towers in ArchiCAD (left) and in Revit (right).

5.14 The time that each back-end took to create each of the case studies. The difference in performance is possibly attributed to the applications themselves, and to the languages that each Application Programming Interface (API) uses, C++ for ArchiCAD and C# for Revit.

5.15 The same program, which contains Building Information Modeling (BIM) information, generated the model in different back-ends. Starting from the top left: OpenGL; Rhino; ArchiCAD; SketchUp; AutoCAD; and Revit.

5.16 To test the performance of Rosetta’s back-ends we created the model seen in Figure 5.15 and compared the time each one took. The results are shown in this graph, where it is visible that the BIM applications have the worst performance.
5.17 The Abu Dhabi shell (source: http://openbuildings.com/). 65
5.18 The several layers that compose the shell. 66
5.19 The element that is repeated throughout each layer, and the profile of each beam.
(source: http://www.thenational.ae/) 66
5.20 The profile of a beam produced by our program. 66
5.21 A render of eight layers produced by our program. 67
5.22 A render of a more detailed view of the shell produced by our program. 67
5.23 The Market Hall of Rotterdam. (source: http://www.archdaily.com/) 68
5.24 A render of the Market Hall model produced by our program in ArchiCAD. 68
5.25 All the walls of the building that have a custom profile. 69
5.26 The detail of a rotating door of our model. 69
5.27 The process of trimming the walls using a shell. 70
List of Tables

1.1 A table that compares ArchiCAD's and Revit's features. . . . . . . . . . . . . . . . . 9

2.1 A comparison between the analyzed tools, showing what kind of language support they have, if they support geometric operations, and the level of supported BIM operations, which is related to the number of check marks. . . . . . . . . . . . . . . . . 21

2.2 A detailed analysis of the operations supported by the several tools. . . . . . . . . . 22
Listings

2.1 A function that creates a slab using the ArchiCAD C++ API. .......................... 18
3.1 Creating a slab using static back-ends in Rosetta. ................................. 26
3.2 Creating a slab using dynamic back-ends in Rosetta. ......................... 27
3.3 The slab operation of our solution. The guide is the only required parameter, while the remaining ones have default values. ....................... 28
3.4 Creating a slab using our solution. ......................................................... 28
4.1 A program that creates a floor, composed of a slab and four walls, using only boxes. ................................................................. 35
4.2 The same program as before, but taking advantage of abstractions to be more legible and easier to understand. .................................. 35
4.3 The program modified to work with BIM operators. We needed to change the slab operation to one that produces a rectangular slab. The wall operation remained the same, as the remaining parameters, such as material, take default values. . 36
4.4 The code needed to create the door in a Computer-Aided Design (CAD) tool . . 38
4.5 The code to create the door in the BIM application ArchiCAD. We first create a wall and only then create the door. We take advantage of the properties of the door objects to manipulate the structure of the door. ......................... 39
4.6 Changing a door’s type using our solution. ........................................... 40
4.7 The operation that allows the creation of custom library parts. It is responsible by sending the Geometric Description Language (GDL) code and additional information such as type to the ArchiCAD plug-in. ......................... 42
4.8 Instantiating our created railing. It uses the added parameter to control the inclination of the top handrail. ................................. 42
4.9 The walls operation of Revit (line 1) and ArchiCAD (line 6). .................... 45
4.10 An example of the usage of levels. .................................................. 47
4.11 The creation of several walls using different materials and type of materials. . . 48
4.12 A profile being applied to a wall, in its creation process. ......................... 49
5.1 The intermediate abstraction for the facade element of the original CAD program. 62
5.2 The BIM version of the facade element. It was needed to specify the family of the element (line 4) and use the correct building component (lines 10 and 16). . . . . 62
5.3 Changing the doors’ type by using the modeling operation’s type parameter. . . . 69
**Acronyms**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BASIC</td>
<td>Beginner’s All-purpose Symbolic Instruction Code</td>
</tr>
<tr>
<td>BIM</td>
<td>Building Information Modeling</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer-Aided Design</td>
</tr>
<tr>
<td>GC</td>
<td>Generative Components</td>
</tr>
<tr>
<td>GD</td>
<td>Generative Design</td>
</tr>
<tr>
<td>GDL</td>
<td>Geometric Description Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>VB</td>
<td>Visual Basic</td>
</tr>
<tr>
<td>R-G-A</td>
<td>Rhino-Grasshopper-ArchiCAD</td>
</tr>
<tr>
<td>RPS</td>
<td>RevitPythonShell</td>
</tr>
<tr>
<td>REPL</td>
<td>Read-Eval-Print Loop</td>
</tr>
</tbody>
</table>
# Introduction

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Computer-Aided Design &amp; Building Information Modeling</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Generative Design</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Objectives</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>ArchiCAD &amp; Revit</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Summary</td>
<td>8</td>
</tr>
</tbody>
</table>
Architects have been changing their work tools throughout the years. Their initial instruments were the traditional pen and paper, which were error-prone and made changes in later design phases difficult and time-consuming. Furthermore, it was also difficult to keep every document of the project consistent and accurate throughout the development of the project. This eventually led to errors and omissions in paper documents, which caused unanticipated field costs, delays, and sometimes even lawsuits, resulting in financial expense [1].

With the advance of technology, several tools were developed to improve the process of architectural design. The first major shift was with Computer-Aided Design (CAD) applications that digitalized architecture and the second one was with Building Information Modeling (BIM) applications. In the next section we explain these two major paradigms, their differences and disadvantages.

1.1 Computer-Aided Design & Building Information Modeling

CAD applications consist mainly of drafting and modeling systems. They are used to design blueprints, elevations, and sections of buildings, although most of the modern applications also support the design of a 3D model. These tools provide a more efficient working process in architectural design and better documentation than the traditional pen and paper approach [1]. Furthermore, CAD tools stimulated the desire for more complex structures due to facilitating their conception [2].

Despite the advantages that CAD tools bring, they are still error-prone due to every building view being independent. This independence means that a change in one view must be manually propagated to all other views, giving room to user errors. Furthermore, conflicts in different building views, such as a piping view and an air-conducts view, are hard to identify and must be manually detected and corrected. These problems create a disconnection between the documents of a building project.

BIM applications were developed as an answer to these problems. The formal definition of BIM from the United States BIM Standard [3] is as follows: "Building Information Modeling is a digital representation of physical and functional characteristics of a facility. A BIM is a shared knowledge resource for information about a facility forming a reliable basis for decisions during its life-cycle; defined as existing from earliest conception to demolition." Furthermore, building models can be characterized by [1]:

- Building components, which are represented by objects that have data attributes and parametric rules.
• **Components that include data that describe how they behave**, which are needed for analysis and work processes, such as energy analysis and specification.

• **Coordinated, consistent and non-redundant data** such that all views of the model reflect changes made on components.

BIM applications have been gradually adopted in order to mitigate the fragmentation present in the design and construction processes [4]. This increase in adoption is being reinforced in several countries, by making mandatory that public sector projects provide BIM implementations, being United Kingdom a recent example [5].

The CAD and BIM paradigms are very different, and using the latter involves some significant shifts in design methodologies. For example, while CAD tools mostly deal with geometry, BIM tools are capable of producing an accurate virtual model of a building containing not only geometry, but also relevant data needed to support the construction, fabrication, and procurement activities needed to realize the building [6, 7]. These data attributes contain information regarding building components, such as what they are made of, how they behave and how much they cost [1].

A major advantage of BIM tools is the ability to support multiple views of the data contained within a drawing set, including 2D and 3D perspectives [1]. These views are not independent and are handled by the application, meaning that a change in one view will be propagated to the others. Not only this, but BIM projects are intended to be used throughout the lifecycle of a building, keeping all the project data consistent and accurate. This data includes information relevant not only to architecture but also to engineering and to construction.

Another important difference between the CAD and BIM paradigms is that building components have parametric rules that dictate their behavior in the model. These rules establish associations between objects, such as a door that can only exist hosted in a wall, which help ensure that the digital building components behave more like their real counterparts. These associations also facilitate the designer’s job by propagating changes in the design.

Although CAD is an evolution from the paper-based approach and BIM is another step in the evolution chain, there is still room for improvement. For example, the automation of repetitive and monotonous tasks, the creation of complex geometry, and faster exploration of alternatives to the original design [8] are all improvements that can still be made. All these improvements can be achieved with Generative Design (GD), which we address in the following section.
1.2 Generative Design

Generative Design is an algorithmic-based approach to design that allows the generation of forms or shapes through algorithms [9]. Using GD, instead of designing a building, one designs the system that designs the building [10].

Due to the algorithmic nature of GD it is possible to mechanize repetitive, time-consuming and tedious tasks. This automation relieves architects from error-prone work, allowing them to save time and effort during the design process. This advantage is most noticeable in projects that have a high degree of repetition, such as skyscrapers.

Another advantage that GD brings is the ability to simplify the creation of complex geometry. The designer can describe problems through an algorithm, turning the problem into a formal description. Furthermore, GD programs are typically parameterized, meaning that the designer can quickly generate and test with minimal effort a wide range of different solutions, thus supporting exploration and optimization in the design process [11].

The advantages of GD stimulated the development of tools that allow its exploration for CAD applications. Some examples of such tools are Grasshopper\(^1\), Visual LISP\(^2\), and Rhino-Script\(^3\).

Nevertheless, as explained in the previous section, BIM applications have several advantages over traditional CAD tools. As such, it is tempting to also explore GD with the BIM paradigm, and indeed, many BIM applications have APIs, which provide a programmatic form of interaction with the applications. However, the use of these APIs requires extensive knowledge of advanced programming languages, such as C# or C++, which few architects possess.

To overcome this problem, a few tools were developed to hide some of the complexity of the APIs. Dynamo and Rhino-Grasshopper-ArchiCAD are examples of such tools, as they provide simple programming languages, in some cases based on visual metaphors, offering a more beginner-friendly alternative. Although very appealing for developing small programs, these languages become a barrier for more complex programs, making them difficult to understand, use, and modify [12]. Furthermore, some of the tools can become obstacles themselves, due to performance issues when dealing with more complex and larger programs, that typically make the user interface unresponsive.

We propose a solution that allows the exploration of GD with BIM using a programming environment that is fit for beginners, but that can also be used for complex programs: Rosetta. This environment uses DrRacket, an educational Integrated Development Environment (IDE) [13,14], and supports several programming languages, such as Racket and Python, that can

\(^1\)http://www.grasshopper3d.com/ visited on March 2016
\(^2\)http://www.afralisp.net/autolisp/ visited on March 2016
\(^3\)http://wiki.mcneel.com/developer/rhinoscript visited on March 2016
be used by both beginners and experienced users. This means that the solution will be able to accompany the learning process of its users, and do not turn into an obstacle. Rosetta will be explained further in the thesis.

1.3 Objectives

This thesis explores the problem of programming architectural forms in the BIM context. As such, the thesis will contribute with the conception and implementation of appropriate algorithmic-based modeling operations for the BIM paradigm. The result will be an implementation of a GD tool capable of producing objects in a BIM application. We will evaluate our thesis for the specific case of the Graphisoft’s BIM application, ArchiCAD.

The modeling operations that the solution provides will take into consideration the characteristics of the BIM paradigm, such as the semantics introduced by having BIM objects, and the restrictions that exist between them. Furthermore, the operations provided by the tool will be developed in order to support portability to other BIM applications, allowing users to easily switch between them.

1.4 ArchiCAD & Revit

In this section we go over the BIM application that was selected to evaluate the thesis, ArchiCAD. Furthermore, due to the desire to have portability between BIM applications, we will also explain Revit, which is the BIM tool chosen for evaluating portability.

ArchiCAD has an operation for creating each building component that it supports, such as walls, columns, beams, stairs, and others. It also allows the creation of some geometric abstractions, such as lines, splines, arcs, circles and others.

The building components in ArchiCAD have properties that modify or give them additional information. Those can be: thickness, slant angles, stories, materials, and others.

The material is a more complex property as it contains additional information. It determines the appearance of the element, cost, thermal conductivity and others. Furthermore, if the material to be used is a composite of several materials, it will restrict the thickness of the building component to that of the material composite.

The story attribute is ArchiCAD’s representation of building floors. To create a story it is needed a reference story, a height relative to that story, and to state if the new story is to be inserted above or below the reference story. The story can also be named, although it is not required. This creation method will be important when we compare it to Revit’s building
floors.

By using stories, models can be organized by floors of the building. Each element has a home story that determines its z-coordinate. Some elements, such as walls and columns, can also be linked to a top story. This association controls the height of the element by calculating the difference between the home story and the top story heights.

Although ArchiCAD supports a great number of building components, it does not have all possible components. To overcome this limitation and also allow more designer freedom, ArchiCAD supports the creation of custom objects, known as Library Parts. These are created based on geometric forms, such as boxes, cylinders, cones and others. Despite their geometric nature, library parts support some BIM functionality, such as materials, stories, and others. By taking advantage of library parts we are able to create additional building components that might be necessary for designing a building, such as railings, elevators and other elements.

Nevertheless, library parts have disadvantages. They do not have the same level of semantics that the supported building components have. This will limit the operations that can be performed upon library parts. For example, we cannot insert a door, or window, in a library part that represents a wall.

Besides creating elements, ArchiCAD can create profiles for certain elements, namely, walls, beams, and columns. These profiles change the elements’ 2D section. By using profiles we can modify existing elements while keeping the semantics associated with each building component.

These are the most important concepts of ArchiCAD that are addressed in this document. Now we move on to the similarities and, more importantly, the differences that Revit has when compared to ArchiCAD.

Although Revit is based on the same BIM concepts of ArchiCAD, it has substantial differences that have an impact on how the tool works. This will, in turn, be reflected on the modeling operations that we created for ArchiCAD and the existing ones of Revit.

Similar to ArchiCAD, Revit supports the creation of both 2D elements and 3D elements. It has an object for each of the supported building components. However, Revit handles the elements’ properties quite differently. Each element has a family, and it is this family that determines the properties of an element.

In addition, each family has a specific material attributed to it that cannot be changed. To change the material of an element, one has to change the family of the element, or create an entire new family to support a given material.

In Revit we can create types, which are a specific element of a given family. They have
the same properties as their family, but with different values, such as bigger dimensions or a different material. These properties can have their values locked, meaning that the user cannot change it, or restricted by limits, allowing changes in a given range. This behavior, despite limiting the designer’s freedom, fits well within the premise of only allowing manufac-
turable elements, because distributors or constructors might only do specific sizes of building components and these are provided as families and family instances.

Revit allows the creation of families, meaning that the user can create a family, specify a material to it, model the geometry of the object, and control other properties. In ArchiCAD, the equivalent of creating a family can be either limited to changing the element’s properties, such as material and thickness, or apply a profile to an element. When the element’s properties or profiles are not enough, the user should create a custom library part.

The next difference is the representation of building floors. Although their concept is common to BIM tools, they have different implementations. In Revit they are called levels, opposed to ArchiCAD’s stories, and their creation method is also different. It is only required a name and an absolute height to create a level, whereas in ArchiCAD it is used a reference level, relative height and an insert location (above or below).

The differences between these two BIM applications, namely in element properties, profiles and stories/levels are a crucial point to portability, as something as simple as thickness or height are handled differently by the applications. In later sections of this document, we address the normalization of the differences between the applications. To end this section we present Table 1.1 that summarizes the differences of the compared tools, ArchiCAD and Revit.

1.5 Summary

In this section we introduced several topics relevant for the development of this thesis. We started by providing some background regarding the CAD paradigm that improved the architectural design process. Next, we presented the BIM paradigm that brings advantages over the CAD paradigm, such as objects that represent building components and data that is coordinated, consistent, and non-redundant.

After presenting the common tools used by architects we moved on to Generative Design, which is an algorithmic-based approach to design. It allows the mechanization of repetitive, time-consuming and tedious tasks, simplifies the creation of complex geometry, and faster and easier exploration of alternatives.

The objective of this thesis is to explore GD with the BIM paradigm. In order to evaluate the
Table 1.1: A table that compares ArchiCAD’s and Revit’s features.

<table>
<thead>
<tr>
<th>Elements</th>
<th>ArchiCAD</th>
<th>Revit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supports 2D elements, and 3D elements for each building components.</td>
<td>Supports 2D elements, and 3D elements for each building components.</td>
<td>Similar to ArchiCAD.</td>
</tr>
<tr>
<td>Element Properties</td>
<td>Each element has properties. These can be materials, thickness, width, height, cost, angles, and others. Properties are usually independent, although some materials can restrict thickness.</td>
<td>Each building component belongs to a family. Properties are bound to the families. Most of the families restrict or limit the values of their properties.</td>
</tr>
<tr>
<td>Custom Elements</td>
<td>Elements can be modified by their properties, and, in the case of walls, columns and beams by their profile as well. For further customization, it is necessary to create a library part, that allows more geometric freedom but restrict the semantics of the element.</td>
<td>Elements are defined by their family, and Revit allows the creation of new families. By creating them the user can specify the geometry of the element along with its properties.</td>
</tr>
<tr>
<td>Building Floors</td>
<td>Each element needs to be in a building floor. Some elements can be linked to an upper floor meaning that their height is determined by that upper floor. ArchiCAD’s implementation of building floors are stories. To create a story it is needed a reference story, a relative height to that same story and if the new story is to be inserted below or above.</td>
<td>Revit uses the same concept of building floors, with the exception of their name, and creation method. They are called levels and to create one it is only required a name and an absolute height.</td>
</tr>
</tbody>
</table>

thesis we chose ArchiCAD as the BIM application. Additionally, we want to explore portability between BIM tools and, as Rosetta already supports Revit, we will test portability with it. To this end we explained the primary concepts of ArchiCAD, followed by a comparison with Revit.

The remainder of the document is organized in the following sections:

- Related Work: an analysis of several tools that explore GD with the BIM paradigm;
- Architecture: a detailed look at the architecture of the solution, along with an explanation of Rosetta;
- Generative Design for BIM: an overview of the impacts GD has on BIM, along with challenges that we faced during the development of the solution;
- Evaluation: of the solution, based on its adequacy, portability, performance and support of ArchiCAD-specific operations;
- Conclusion: of the document, along with future work.
## Related Work

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Grasshopper</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Dynamo</td>
<td>15</td>
</tr>
<tr>
<td>2.3 RevitPythonShell</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Generative Components</td>
<td>16</td>
</tr>
<tr>
<td>2.5 ArchiCAD API</td>
<td>17</td>
</tr>
<tr>
<td>2.6 Geometric Description Language</td>
<td>20</td>
</tr>
<tr>
<td>2.7 Analysis</td>
<td>21</td>
</tr>
<tr>
<td>2.8 Summary</td>
<td>22</td>
</tr>
</tbody>
</table>
The advantages of GD stimulated the development of several tools, that we analyze in this section. The first one is Grasshopper, a widely used GD tool for the CAD application Rhinoceros 3D, and some of its extensions for BIM applications. Then, we present Dynamo, a GD tool for the BIM tool Revit, followed by RevitPythonShell, that allows the use of Revit’s API through IronPython. Next, we present Bentley’s parametric modeling system, Generative Components. Finally, we look at two tools provided by Graphisoft, the company behind the BIM tool ArchiCAD.

2.1 Grasshopper

Grasshopper is a plug-in for Rhinoceros 3D (a CAD application), which provides a visual programming language based on functional blocks that can be connected into a sequence of actions [15]. By using this simple and interactive way to program, Grasshopper becomes an enticing application for beginners.

Despite this, Grasshopper also suffers from a serious disadvantage: lack of scalability. As programs grow in complexity they become hard to understand and difficult to change [12]. Figure 2.1 shows an example of how the readability and maintainability of the program can suffer when the problem at hand is complex.

To mitigate this disadvantage, Grasshopper allows the creation of small Visual Basic (VB), C#, and Python scripts by using a specific code block. These blocks have multiple inputs and outputs that will allow the connection to other blocks in the Grasshopper program [16]. Despite the ability to create small textual scripts, this feature does not eliminate the scalability problems, because these blocks are still bound to Grasshopper’s visual language and cannot be used as large-scale programs.

2.1.1 Lyrebird

Grasshopper alone does not support BIM tools, therefore it cannot explore that paradigm. However, it can be extended with plug-ins, such as Lyrebird, that connects Grasshopper to Revit (a widely used BIM application). Lyrebird adds a single component to Grasshopper, visible in Figure 2.2, that the authors “hope has enough inputs and options to handle everything without being overwhelming” [17]. It is this component that receives as input all the information regarding the objects to be created on Revit, such as origin points, curves, and orientation, etc. Furthermore, the component also specifies the Revit Family that is to be used, for instance, wall, slab, and handrail, among others.
2.1.2 Hummingbird

An alternative to Lyrebird is Hummingbird, another plug-in that also allows the creation of native Revit objects based on data that is generated from Grasshopper. Unlike Lyrebird, this plug-in provides several components for use in Grasshopper, such as the wall, floor, and column components. Those are conveniently parameterized and their input can be information from other Grasshopper components, such as points and curves. Another difference between Lyrebird and Hummingbird is that the latter supports the creation of Revit’s Mass Families, which are objects composed of free forms that are normally used for volumetric studies [18,19].

2.1.3 Rhino-Grasshopper-ArchiCAD

More recently, on September 2015, Graphisoft announced the public beta version of its R-G-A connection [20]. To achieve this connection, Grasshopper added several components that represent the elements of ArchiCAD. These components receive the necessary geometrical information that is needed to create the corresponding ArchiCAD objects. For example, to create a column, the component receives the column’s begin and end points as inputs. Figure 2.3 shows an example where multiple columns are created.
Nevertheless, this solution suffers from the same problem described before: lack of scalability. Therefore, there will be repercussions on the readability and maintainability of programs.

2.2 Dynamo

Dynamo is an open source GD application that has two working modes: it can run in a stand-alone "Sandbox" mode, or run as a plug-in for other applications such as Revit or Maya (a computer animation and modeling software). Similarly to Grasshopper, it provides a visual programming language, designed to be accessible not only to programmers but also to non-programmers [21].

By using nodes and wires, users can create programs in Dynamo, as seen in Figure 2.4. The nodes are objects that perform an operation, such as storing a number or creating geometry. Through wires, it is possible to create relationships between the nodes, meaning that they link the output of a node to the input of another node, thus creating the data flow of the program [21].

Dynamo also supports the creation of small scripts using Python. These scripts are represented by code blocks in the visual diagram, that can have both inputs and an output. They are still connected by wires as other normal blocks.

Although the addition of small textual scripts gives more flexibility to the tool, those only
work as blocks in the visual diagram. Therefore, Dynamo suffers from the previously mentioned drawbacks of Grasshopper.

2.3 RevitPythonShell

RevitPythonShell (RPS) was developed by Daren Thomas [22], and was designed as an alternative to access Revit’s API. This tool allows the use of IronPython, an open-source implementation of Python, which is a more beginner-friendly language than C#, the official language of Revit’s API.

Although the tool might use a popular and beginner-friendly language, it is just a translation of the C# variant. It does not strive to hide any of the complexity present in the API. Users still use all the API’s functions as they were implemented, without any additional simplifications or abstractions. Furthermore, users still have to manage transactions in order to access the API’s functionality.

In sum, the application is no more than an IronPython interpreter of the Revit API, that allows its exploration using a beginner-friendly language, but that does not go beyond that.

2.4 Generative Components

Bentley’s parametric modeling system is Generative Components (GC), which enables the definition of complex parametric geometry assemblies. GC is a propagation-based system, composed of an acyclic directed graph and two algorithms, one for ordering the graph and
one for propagating values through the graph [23]. In Figure 2.5 there is a simple graph that manipulates the position of points.

Each node of the graph can contain variables and constraints between variables. The variables can be either dependent or independent. The independent ones, as the name suggests, do not depend on other variables, while the dependent variables are defined at the expense of other variables. As an example, a variable $a$ can be assigned the value 1, making it an independent variable. However, if $a$ is defined as the addition of two other variables, $b$ and $c$, its value will change whenever $b$ or $c$ changes [23].

GC provides four views: a 3D interactive view, a symbolic graph view, an object view, and a programmatic view [23]. These views change the way users interact with the system. The most notable one is the programmatic view that allows elements to be directly programmed using the system’s API.

### 2.5 ArchiCAD API

Graphisoft – the company behind ArchiCAD – provides an API, which is mainly used by professional programmers to develop plug-ins for ArchiCAD. The API is based on C++ and gives developers control over some of ArchiCAD’s functionality, such as element creation and manipulation. Although this API is the primary way of accessing ArchiCAD through pro-
gramming, it is not fit for beginners, since it requires the usage of a language that is not beginner-friendly. In addition, it requires knowledge of concepts that might be foreign to beginners, such as memory management, transactions, and polymorphism. As an example of how complex the API can be, we show a function to create a slab in Listing 2.1. In order to understand the function, one needs to understand several concepts:

- The opening of the undoable session (lines 4-7).
- The use of default values for easier creation of the element (lines 11-13).
- The attributes of the desired element, in this case a slab (lines 14-20).
- The required memory allocation for:
  
  **Element**, which has general information, such as its level and material (line 8).

  **Memo**, which has geometric information (line 9).

  Geometric information, such as coordinates, sub-polygons and arcs (lines 21-33).
- The function to create an element (line 44).
- The closing of the undoable session (line 48).

```c
void createSimpleSlab(){
  API_Element element;
  API_ElementMemo memo;
```
Listing 2.1: A function that creates a slab using the ArchiCAD C++ API.

Despite the complexity of this function, its result is just a simple rectangular slab. In Section 3 we show how our proposed solution can substantially simplify the creation process of a slab, or any other element.

As a final remark, the ArchiCAD API is not as easily accessible as the APIs of other BIM tools, because the user must sign a contract with Graphisoft to access the API. This might be
considered a barrier to potential users, making them less likely to try the API.

2.6 Geometric Description Language

Another tool that Graphisoft provides is GDL, a scripting language for object creation, that grew out of Beginner’s All-purpose Symbolic Instruction Code (BASIC). It provides commands such as BLOCK, EXTRUDE and TUBE, among others, that allow the creation of 2D and 3D objects [24]. These are ArchiCAD’s Library Parts that we have seen in Section 1.4. An example of a GDL script is shown in Figure 2.6.

![Figure 2.6](image.png)

**Figure 2.6:** A GDL script that shows how the 3D model of a chair can be created.

GDL is mostly used by manufacturers to create objects that represent their products and have the correct construction information. It is also useful to introduce objects that do not have a specific operation in ArchiCAD, such as railings, elevators, furniture, and others.

While GDL makes it possible to manipulate geometry to create objects, it does not support the semantics that the BIM paradigm offers, meaning that the created objects are not recognized as what they represent in BIM applications. For example, we can create a wall using a block operation, but ArchiCAD only recognizes it as a generic object, and not an actual wall. This will impact operations that are specific to wall elements, such as inserting a door.
or window.

We can conclude that although GDL provides a way to create objects that are not originally supported by ArchiCAD, it does not explore all the advantages that the BIM paradigm has to offer. Moreover, although understandable by a beginner, the language is now obsolete, relying on outdated control structures, including the infamous GOTO. As a result, it also suffers from scalability problems and is inadequate for large projects.

### 2.7 Analysis

In this section we present an analysis of how the previous tools influenced the proposed solution. Table 2.1 shows a comparison of the tools regarding language support (visual and/or textual), if they have geometric operations, and their level of support for BIM operations.

**Table 2.1:** A comparison between the analyzed tools, showing what kind of language support they have, if they support geometric operations, and the level of supported BIM operations, which is related to the number of check marks.

<table>
<thead>
<tr>
<th>Application</th>
<th>Visual</th>
<th>Textual</th>
<th>Geometric Operations</th>
<th>BIM Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasshopper</td>
<td>✓</td>
<td>✓ (small scripts)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lyrebird</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hummingbird</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>R-G-A</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamo</td>
<td>✓</td>
<td>✓ (small scripts)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RPS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ArchiCAD API</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GDL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The majority of the presented tools are based on visual programming languages that, as mentioned before, suffer from scalability problems that will affect the readability and maintainability of the programs. On the other hand, the tools that opt to use textual programming languages, as seen before, are not adequate to beginners due to their languages being complex (C++, C#) or obsolete (GDL). The only exception is RPS, which supports IronPython, but that is only a translation of Revit’s API, making no effort to simplify its complexity.

To better understand what each of the analyzed tools is capable of doing, we show in Table 2.2 a more detailed look at the supported operations of each tool.

With our solution we want to solve the problems of the currently available tools. To avoid the problems of scalability, we allow users to choose from a set of textual programming languages that are simple enough to learn but expressive enough to scale to programs of considerable size. However, we do not want to use complex programming languages. Hence, we support programming languages that are fit for beginners, like Racket [25] and Python [26]. Furthermore, the solution allows the use of an educational IDE, DrRacket [13, 14], that will...
Table 2.2: A detailed analysis of the operations supported by the several tools.

<table>
<thead>
<tr>
<th>Application</th>
<th>Textual Language</th>
<th>Geometric &amp; BIM Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasshopper</td>
<td>Small scripts: VB.net, C#, Python</td>
<td>Supports geometric operations provided by Rhino (CAD application).</td>
</tr>
<tr>
<td>Lyrebird</td>
<td>No</td>
<td>Supports the creation of some BIM objects in Revit.</td>
</tr>
<tr>
<td>Hummingbird</td>
<td>No</td>
<td>Supports the creation of some BIM objects and masses in Revit.</td>
</tr>
<tr>
<td>R-G-A</td>
<td>No</td>
<td>Under development, but already supports instantiation and manipulation of BIM objects.</td>
</tr>
<tr>
<td>Dynamo</td>
<td>Small scripts: Python</td>
<td>Supports the creation of BIM objects and masses in Revit.</td>
</tr>
<tr>
<td>RFS</td>
<td>IronPython</td>
<td>Supports all the operations provided by the Revit API, which includes geometric and BIM objects.</td>
</tr>
<tr>
<td>GC</td>
<td>C#</td>
<td>Supports all operations of Bentley’s softwares, namely Microstation and AecoSIM Building Designer.</td>
</tr>
<tr>
<td>ArchiCAD API</td>
<td>C++</td>
<td>Supports all geometric and BIM operations provided by ArchiCAD.</td>
</tr>
<tr>
<td>GDL</td>
<td>GDL (based on BASIC)</td>
<td>Supports Library Parts of ArchiCAD, which are created through geometric operations.</td>
</tr>
</tbody>
</table>

Finally, we do not want a simple translation of the API’s functionality. Our operations ease their learning experience. Our operations hide the complexity of the API without losing its expressive power. This allows users to avoid dealing with the complex concepts that the API requires and focus on the design they want to implement.

2.8 Summary

In this section we have seen several tools that explore GD with the BIM paradigm. On the one hand we have the ones that are based on visual metaphors, which are fit for beginners but have scalability problems. On the other hand we have the others that use textual languages, but that end up being too complex for beginners, due to the chosen language or by requiring the knowledge of advanced programming concepts.

In the next section we will go over the overall architecture of our solution. We will also explain Rosetta and how we expanded it with a new back-end and a new set of modeling operations for that back-end.
3 Architecture

Contents

3.1 Rosetta ............................................................... 25
3.2 Abstraction Layer ............................................... 27
3.3 ArchiCAD's Plug-in ............................................. 29
3.4 Communication Channel ....................................... 29
3.5 Interaction .......................................................... 29
3.6 Summary ............................................................ 30
In this section we go over the overall architecture of the solution: an application that allows designers to develop programs that can be portably executed in the context of different BIM tools and that can generate equivalent building models in each of those tools. The application is composed by two components: an abstraction layer and a plug-in for a BIM application. It is through the abstraction layer that the user has access to modeling operations that produce the desired effects in the BIM application. To this end, we developed a communication channel through which the abstraction layer sends the necessary information to the plug-in of the desired BIM.

The abstraction layer was designed to support multiple BIM applications and when possible, eliminate differences between the supported applications. This allows users to switch between them, without having to re-learn all the primitive operations from scratch. Moreover, this is useful to support portability between BIM applications. Figure 3.1 shows a components diagram of this architecture.

![Figure 3.1: A diagram depicting the overall architecture of the solution.](image)

In order to implement the solution we used Rosetta, a programming environment for GD, which will be explained in the following section. After that, we go over the two components that compose the solution, i.e., the abstraction layer and the plug-in. Finally, we describe how the communication channel was implemented, followed by an example of interaction between the components.

### 3.1 Rosetta

Rosetta has a front-end/back-end architecture, where the front-ends are programming languages and the back-ends are used to produce the modeling results. Every front-end has access to an abstraction layer that provides the modeling operations. This creates a loose coupling between the front-ends and back-ends, allowing users to switch programming languages and modeling applications [27].
Rosetta’s front-ends are programming languages fit for beginners, such as Racket, Python, and Processing. Its back-ends are also diverse, as it supports an OpenGL back-end for fast visualization and several CAD tools.

Rosetta’s abstraction layer provides geometrical abstractions, that can be used by other modeling operations. An example of those abstractions are the coordinate systems. It is possible to work with several systems, such as Cartesian, polar, cylindrical, and spherical. These abstractions set a common ground for other modeling operations to be built upon, such as walls, slabs, and columns that can use coordinates to describe their position and form.

The modeling operations provided by the abstraction layer need to be developed for each of the supported back-ends of Rosetta. Some of those operations are portable, meaning that they can be used regardless of the chosen back-end, due to the fact that they require the same parameters for different back-end applications. For example, the box operation – a common operation in CAD tools – requires a corner point and three dimensions (width, length and height) to create a box in any of the supported back-ends. Each implementation of that operation treats the information from the parameters in order to work with its corresponding back-end application.

Another feature of Rosetta is the ability to either use back-ends statically or dynamically. Each mode has its advantages and disadvantages. When the user chooses the static mode, it means that a single back-end is selected and the program will only interact with that back-end in its execution. It brings the advantage of checking before execution if the operations used are supported by the chosen back-end. This creates a direct link between the operations provided by Rosetta and the chosen back-end, as there is no dynamic choice of back-end. An example of this interaction can be seen in Listing 3.1.

```
1 (require rosetta/archicad)
2 (slab (list (xy 0 0)(xy 5 0)(xy 5 5)(xy 0 5))
```

**Listing 3.1:** Creating a slab using static back-ends in Rosetta.

When working with dynamic back-ends the user forgoes the verification of operations to be able to work with multiple back-ends in a single execution. The user chooses a back-end using the operation backend. This informs Rosetta which implementation of the operation is to be used. By using dynamic back-ends we can create the same model in different back-ends in a single execution of the program. In Listing 3.2 we see an example that takes advantage of dynamic back-ends.

Another advantage that comes from using dynamic back-ends is the ability to exchange
information between them. In other words, we can retrieve information from a back-end and use that information as input in other back-end. For example, we can manually create several lines, curves, and splines in a CAD tool, and, using Rosetta, extract their information and use it to produce objects in a BIM application.

```
1 (require rosetta)
2 (backend autocad)
3 (slab (list (xy 0 0)(xy 5 0)(xy 5 5)(xy 0 5)))
4 (backend rhino)
5 (slab (list (xy 0 0)(xy 5 0)(xy 5 5)(xy 0 5)))
```

Listing 3.2: Creating a slab using dynamic back-ends in Rosetta.

Finally, Rosetta uses DrRacket as its IDE, which was originally developed to teach programming to beginners [13, 14], with features like: syntax highlight, Read-Eval-Print Loop (REPL), debugger and error reporting. This makes Rosetta a beginner-friendly environment adequate for architects to learn how to program.

All these advantages made Rosetta a good application to extend. Furthermore, alongside our development, another BIM tool, Revit, was being added to Rosetta, which allowed us to test portability between it and ArchiCAD.

### 3.2 Abstraction Layer

The abstraction layer is an extension of Rosetta’s abstraction layer, as it provides modeling operations that were developed specifically for the BIM ArchiCAD. These operations take advantage of some functionality from Rosetta, namely, coordinates, coordinate systems, and other geometric abstractions. By using these abstractions, our operations were built on a common ground used by Rosetta’s back-ends, facilitating the possible portability with operations of other back-ends. Furthermore, previous users of Rosetta will be familiar with arguments of the provided modeling operations. For instance, a slab can be created from a group of points, meaning that if users already know how to create points in Rosetta, they will be able to create a slab.

These operations were implemented in Racket, the language in which Rosetta is currently implemented. However, through Rosetta’s functionality, we are able to export our modeling operations to the other supported languages, such as Python and Processing.

The modeling operations were designed to embrace the capabilities of the BIM paradigm, meaning that they support the semantics of BIM objects. In other words, the elements are defined by their building component rather than by their geometric form. For example, a wall
in a BIM tool is an actual element described as a building component, while in CAD tools it is simply a geometric object, such as a box.

Furthermore, these operations were created to be easily ported but at the same time allow the use of tool-specific functionality. This was achieved by assigning default values to secondary or tool-specific parameters of the elements. By doing so, the operations have a small set of required parameters making them easier and quicker to use. As an example, the slab operation is shown in **Listing 3.3**.

```plaintext
1 (slab guide
2   [thickness 0.3]
3   [bottom-level (current-level)]
4   [bottom 0]
5   [type-of-material (default-slab-type-of-material)]
6   [material (cond [(eq? type-of-material "Basic") "GENERIC - INTERNAL CLADDING"]
7     [(eq? type-of-material "Composite") "Generic Slab/Roof"])]
8   [parcs (list)]
9   [layer (default-layer)]
10  [reference (default-slab-reference)])
```

**Listing 3.3:** The slab operation of our solution. The guide is the only required parameter, while the remaining ones have default values.

The slab operation has only one required parameter, the guide, which determines its shape. This guide, along with the bottom-level parameter, are the only ones that have a direct translation from ArchiCAD to Revit (and vice-versa). The remaining parameters are either specific to ArchiCAD or require additional treatment to be portable between the BIM applications.

Our slab operation substantially simplifies the previously described process of creating a slab using the ArchiCAD’s API (see **Listing 2.1**). In our implementation it only requires a list of points that will outline the slab’s shape (guide). The operation has more optional parameters, such as thickness, material, and bottom offset, among others. In **Listing 3.4** we create the same slab of the previous example, and it is possible to see how it simplifies the creation of elements.

```plaintext
1 (slab (list (xy 0 0)(xy 5 0)(xy 5 5)(xy 0 5)))
```

**Listing 3.4:** Creating a slab using our solution.

All the necessary details that were mandatory using the ArchiCAD API are now hidden by our implementation, and are done by the back-end plug-in, such as opening and closing undoable sessions, memory allocation, and using the adequate functions to create the desired element. This back-end plug-in is further explained in the next section.
3.3 ArchiCAD’s Plug-in

The back-end that our solution adds to Rosetta is the BIM application ArchiCAD. In order to access its functionalities it is necessary to create a plug-in using ArchiCAD’s C++ API. It is through this plug-in that the information from the modeling operations is processed and then used to produce the desired effect, such as creating an object or altering it. This means that for each modeling operation available on the abstraction layer it requires a corresponding function on the back-end that uses the API’s functionality.

Besides receiving information from the modeling operations, the plug-in must also send information to the abstraction layer. This information is, predominantly, about the identification of each object. This is done using a unique identifier that ArchiCAD assigns to each object. It is through this identifier that previously created objects can be used in other operations. For example, to create a door in a BIM application it is required a wall as a host. This restriction is reflected in the door modeling operation by requiring a wall identifier as a parameter.

3.4 Communication Channel

The abstraction layer and ArchiCAD’s plug-in communicate through sockets using Google Protocol Buffers [28]: a language-neutral, platform-neutral, and extensible mechanism for serializing structured data. We created a set of messages that allow the exchange of information between the other components. This tool was appropriate to our solution because it supports both languages used in the abstraction layer and the plug-in, Racket and C++ respectively.

3.5 Interaction

In this section we present an example that showcases the interaction between the various components of the solution: a sequence diagram, visible in Figure 3.2, that describes the creation of a wall followed by the insertion of a door in the wall.

The diagram starts by showing the steps triggered by the wall operation. It needs to create a message suited for the operation, serialize it and send it to the plug-in. After that, the plug-in is responsible for unpacking the message and process its information in order to use the API’s functionality. Next, the plug-in has to send the ID of the wall back to Rosetta. This ID, along with the creation information of the wall, will be wrapped into a data structure and returned to the user. Creating a door causes a similar process, although it is noticeable the
requirement of the wall ID throughout its creation process. In the end, the wall ID will also be wrapped to become a value in the user program.

3.6 Summary

In this section we described the overall architecture of our solution and we explained how we took advantage of Rosetta to build it.

By using Rosetta we gained access to several features: (1) the support of multiple programming languages, such as Racket, Python and Processing; (2) an educational IDE, DrRacket, that is fit for beginners; (3) and access to multiple back-ends that can be used and that already have portability between them.

Rosetta also has a set of geometric abstractions, such as coordinate systems, that are commonly used in the modeling operations for each back-end. By also using them, we facilitate the portability between our added BIM back-end, ArchiCAD, and the other back-ends.

The architecture of our solution is composed by three main components: an abstraction layer, a plug-in for a BIM application, and a communication channel.

The abstraction layer provides the modeling operations necessary to interact with the added BIM application, ArchiCAD. It communicates with the plug-in developed specifically for ArchiCAD to exchange the information regarding the creation or manipulation of elements.

In turn, the plug-in is prepared to receive and correctly process the information so it can use the ArchiCAD's API functionality. The plug-in is also responsible by sending information back to the abstraction layer regarding the created elements, so they can be identified and used by other modeling operations.

We also discussed how the communication channel was realized. It is based on sockets, and uses a serialization protocol named Google Protocol Buffers.

Finally, we presented an example that illustrates the interaction between all the components of our solution when a modeling operation is used. In the next section we address Generative Design for BIM, while exploring several challenges that we faced during the development of the solution.
Figure 3.2: A sequence diagram that shows the creation of a door in a wall.
# Generative Design for BIM

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 CAD &amp; BIM Generative Design Comparison</td>
<td>35</td>
</tr>
<tr>
<td>4.2 Custom Library Parts</td>
<td>42</td>
</tr>
<tr>
<td>4.3 Retrieving Object Information</td>
<td>43</td>
</tr>
<tr>
<td>4.4 Portability</td>
<td>45</td>
</tr>
<tr>
<td>4.5 ArchiCAD Specific Functionality</td>
<td>47</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>49</td>
</tr>
</tbody>
</table>
In the following sections we discuss the impact of GD when applied to the BIM paradigm, addressing the following topics:

- The differences between the CAD and BIM paradigm, and what efforts can be made to bridge the gap between them;
- The ability to create custom Library Parts;
- The ability to retrieve object information from an existing model;
- The abstractions that allow portability between BIM applications;
- The ability to use application-specific features that are needed for more complex or specialized projects.

### 4.1 CAD & BIM Generative Design Comparison

As stated before, BIM applications not only create objects with geometry but also with semantics and information relevant to each building component. For instance, whereas in a CAD application a box can be used to represent several building components, such as walls, slabs, columns, beams and others, in a BIM application each building component has a specific modeling operation. This means that designers will have to adjust their programming practices when transitioning from programming for CAD to BIM applications.

The transition from programming for CAD to programming for BIM can be made easier if the users already use good programming practices. These promote the creation of intermediate abstractions that increase the legibility of programs. For example, in a CAD-based GD program we can create an entire floor using only boxes, as illustrated in Listing 4.1. On the other hand, had the designer used intermediate abstractions that create the needed building elements, the GD program would be much more clear about what it was doing. This is visible in Listing 4.2.

```lisp
1 (box (u0) 10 10 1)
2 (box (x 0) (xyz 10 1 3))
3 (box (x 10) (xyz 9 10 3))
4 (box (y 10) (xyz 10 9 3))
5 (box (x 0) (xyz 10 1 3))
```

Listing 4.1: A program that creates a floor, composed of a slab and four walls, using only boxes.

```lisp
1 (define (slab beginPoint length width thickness)
```

35
To take advantage of these intermediate abstractions, our solution supports the creation of elements using only geometric attributes, leaving the other attributes with default values. This allows for an easier transition from CAD programs to BIM programs. For example, as the user already used abstractions for slabs and walls, it eases the transition to BIM operations, as seen in Listing 4.3.

These abstractions are important to transition from a CAD program to a BIM program, as well as to do the inverse transition. We can create primitives for CAD tools that correspond to each of the building components used in a BIM program. This means that the program still contains all the information and semantics of the elements, but the model can be generated in CAD tools using only the element’s geometric information.

As previously mentioned, our solution supports the creation of BIM elements using only geometric attributes, providing automatic defaults for the remaining, BIM-related, attributes. This encourages an iterative development for the architect, as he will be able to build upon a program that begins with only geometric information to one that also has BIM information, such as material specification. This iterative development fits the exploration process of the architects, as in the early phases of a project they are only concerned with geometric forms. Furthermore, the user can go one step further in their development and relinquish portability to use ArchiCAD specific functionality, which we address in Section 4.5.
Another difference between the BIM and CAD paradigm is the associative rules imposed on BIM elements. These rules can control elements’ attributes, such as their position and height, or even their creation.

An example that illustrates how those rules can control the elements’ attributes is the concept of levels that BIM applications support. These, as we have seen before, are an abstraction of building floors. Each element has to be in a level, as this determines its position in the z-axis. Furthermore, there are elements that can have a top level, which controls the height of the element. This is the case for elements like walls and columns, where it is relevant to have a link to upper floors in order to control their height. By using levels we are able to organize the model in resemblance to real-life buildings. Not only this, but they allow for easier propagation of changes, as any alteration on a level is reflected on their linked elements.

Despite the usefulness of levels, they are a concept foreign to the CAD paradigm, as each element has its own position on the z-axis and its own height. To normalize this difference, we can treat the bottom level as the position of an element on the z-axis, and the difference between the bottom level and top level as the element’s height, thus allowing us to support GD programs that use levels and that can run equivalently in both CAD and BIM tools.

Levels are not the only concept that imposes associative rules on elements. There are certain elements that require associations to other elements to be created, such as a door. In BIM, a door can only be created if it is hosted in a wall. This relation creates a link between the door and the wall, meaning that a change in the wall is appropriately propagated to the door. In a BIM program this association is reflected in the operation that creates a door, as it requires a wall as a parameter, enforcing the link between the elements and imposing a strict order to the creation of these elements.

As expected, the creation of a door in CAD is quite different, as it does not support the associative rules between elements. In a CAD tool it is possible to model a door and a wall in any desired order, meaning that it would be possible to first create the door and only then create the wall that will host it. A possible workflow could be the following sequence of operations:

1. Create the door by modeling every component of the door, e.g., casing, panels, door knob, and others;
2. Create the wall represented by a box;
3. Create the door opening on the wall by:
   (a) Creating a box corresponding to the door’s opening;
   (b) Subtract said box to the wall in order to create the opening.
However this is not the only difference in the creation of a door. CAD applications, in general, do not have a door element, meaning that users need to create the element, relying on the available geometric forms. Depending on the detail level of the door, they might need to model the adequate components such as casing, panels, door knob, and other components. Although this gives the architect a high degree of freedom, the task can be tiresome and time-consuming due to requiring the manual modeling of all of the door’s components. In Listing 4.4 and Listing 4.5 we can see the difference in how much code is needed to produce the same door (Figure 4.1) in CAD and in BIM tools.

```racket
1 #lang racket
2 ;Function to create casing of a door
3 (define (casing p L l h)
4  (define casing-length 0.06)
5  (define casing-width 0.12)
6  (sweep (line (+x p (- (/ L -2) (/ casing-length 2)))
7  (+xz p (- (/ L -2) (/ casing-length 2))) (+ h (/ casing-length 2)))
8  (+xz p (+ (/ L +2) (/ casing-length 2))) (+ h (/ casing-length 2)))
9  (+x p (+ (/ L +2) (/ casing-length 2))))
10  (surface-rectangle p casing-length casing-width)))
11
12 ;Function to create door body
13 (define (door-body p L l h)
14  (box (+xy p (/ L -2) (/ l -2)) L l h))
15
16 ;Function to create door glasses
17 (define (glass p L l1 h)
18  (define door-margin-sides 0.1)
19  (define door-margin-up 0.17)
20  (define door-margin-bottom 0.3)
21  (define door-middle-horizontal 0.2)
22  (define door-middle-vertical 0.07)
23  (define n-glasses-horizontal 2)
24  (define n-glasses-vertical 2)
25  (define glass-area-length (- L
26    (* door-margin-sides 2)
27    (* door-middle-vertical (- n-glasses-horizontal 1))))
28  (define glass-area-height (- h
29    door-margin-up
30    door-margin-bottom
31    (* door-middle-horizontal (- n-glasses-vertical 1))))
32  (define glass-length (/ glass-area-length n-glasses-horizontal))
33  (define glass-height (/ glass-area-height n-glasses-vertical))
34  (map-division (lambda (a b) (box (+xz p a b) glass-length l1 glass-height))
35    (+ (/ L -2) door-margin-sides)
36    (+ (/ L -2) door-margin-bottom
37    door-margin-sides
38    glass-area-length
39    (* door-middle-vertical n-glasses-horizontal))
40    n-glasses-horizontal
41  #f
42  door-margin-bottom
43  (+ door-margin-bottom
44
38
```
44 glass-area-height
45    (* door-middle-horizontal
46        n-glasses-vertical)
47    n-glasses-vertical
48    #f))
49
50 ; Function that uses the previous functions to create the complete door
51 (define (door p L l h)
52    (define glass-thickness 0.005)
53    (define door-margin 0.08)
54    (casing p L l h)
55    (subtraction (door-body p L l h)
56      (glass (+y p (- l)) L (* l 2) h)
57      (glass (+y p (/ glass-thickness -2)) L glass-thickness h))
58
59 ; Finally, we create a box that represents a wall and create an opening
60 ; for the door by subtracting a box with the dimensions of the door.
61 (subtraction (box (x -5)(xyz 5 1 4)) (box (x -1)(xyz 1 1 4)))
62 (door (x 0) 2 1 3)

Listing 4.4: The code needed to create the door in a CAD tool

1 #lang racket
2 ; The first argument of the door function is the wall host,
3 ; followed by the door position relative to the host,
4 ; and finally we manipulate the door's structure through its properties.
5 (door (wall (x 10) (x 10.9) #:height 2.1)
6    (x (/ 0.9 2))
7    #:properties '("gs_door_typ_m" 47
8        "gs_hor_gnum" 2
9        "gs_ver_gnum" 2))

Listing 4.5: The code to create the door in the BIM application ArchiCAD. We first create a wall and only then create the door. We take advantage of the properties of the door objects to manipulate the structure of the door.

Figure 4.1: The door produced by the code seen before.

In Listing 4.5 we took advantage of keyword parameters to customize the door's structure. However, they are not the version used in Racket, but the one adopted by Rosetta, based
on Python’s keyword parameters. We can use any argument that has a default value as a keyword parameter. This allows us to skip the natural order of the parameters by naming the parameter with the prefix #: and specifying its value.

The door element in BIM is part of a library of objects, which is normally composed by elements created by manufacturers based on the products they offer. These elements contain several attributes relevant to their construction, such as correct fabrication dimensions and price. This not only eases the creation of certain building elements, such as doors, but also the construction process, as the architect can choose a product that actually exists in the market.

Another advantage that comes with the library of objects is the ability to change types, which determines the overall structure of the chosen element. In the case of the door, this avoids the need to manually alter the doors’ components. As such, to change the type of a door in a BIM program it is only required to change a single parameter. As an example, in Figure 4.2 we change from the default door to another housing door, and then change it to a rotating door that is more commonly seen in market halls or hotels. This was achieved by only changing a single parameter, as we can see in Listing 4.6. If we were trying to do the same change but using GD for CAD, we would need to write a considerable amount of code, similar or even bigger than the one presented in Listing 4.4.

![Figure 4.2](image)

**Figure 4.2:** The door on the left was created using the default type of door. The other two doors were created by specifying the type of door but still using the same parameters as before.

| 1 | (door (wall (list (y 30) (xy 5 30)) (x 2.5)) |
| 2 | (door (wall (list (y 40) (xy 5 40)) (x 2.5)) "Arch Double Door, 2 Sidelights, Transom") |
| 3 | (door (wall (list (y 50) (xy 5 50)) (x 2.5)) "Revolving Door 18") |

**Listing 4.6:** Changing a door’s type using our solution.

The code seen in Listing 4.6 shows how the user can quickly change the type of door by adding an optional parameter. The first and second arguments of the door are mandatory,
as they are the wall host and the door’s position relative to the wall. The third argument is optional and allows the user to choose a completely different type of door.

As a final example of the differences between programming for BIM applications rather than CAD, we will look at the railing element. The process to create this element is similar to the one used for doors, meaning that in CAD it requires the modeling of all their components, while in BIM tools we can choose their type from a library of objects. A disadvantage of using these predefined objects is that it can limit the design possibilities and creative freedom of the architect. To mitigate this issue, BIM tools provide some attributes that can be changed to control and modify specific parts of a type of object. For example, we can alter the length of the railing and all the other components will adjust to the new dimension, as seen in Figure 4.3. This means that rather than having to manually modify the object’s geometry to achieve the desired change as is the case in a CAD program, in BIM programs we can alter a single attribute to achieve the same effect.

**Figure 4.3:** We changed the length of the railing modifying an attribute of the element. In order to adjust to the new dimension, the element automatically created additional posts.

Although the ability to change some of the elements’ attributes eases the alteration of the objects, it is also limited to the available attributes of a given object. As a result, in a BIM tool an architect cannot explore different shapes as freely as he can in a CAD tool. The library parts come with a limited set of attributes, which might not be enough for certain projects. For example, in ArchiCAD it is not possible to give an angle only to the top handrail of a railing, as the angle given will alter the entire railing, as seen in Figure 4.4. This is a problem within the library part itself and the limited number of attributes that it has.

To surpass this limitation, BIM tools allow the creation of custom objects. However, each BIM application has its representation of objects, e.g., families for Revit and library parts for ArchiCAD. In the next section we explain how we can create library parts using our solution.
Due to the limited set of attributes that the railing has, we are not able to apply an angle only to the top handrail. The only attribute available applies an angle to the overall railing.

### 4.2 Custom Library Parts

Library parts are the representation of objects in ArchiCAD, and GDL is used to create them. As mentioned in Section 2.6, GDL is a scripting language that is based on BASIC. Our solution allows users to create GDL code that is then processed by our plug-in and creates the corresponding library part in ArchiCAD. To that end, we have an operation to create library parts that receives GDL code as parameters. In Listing 4.7 we show the signature of said operation. After creating a library part, we can create instances of it by using the object creation operation. Listing 4.8 shows a possible instantiation of the created library part while using the added parameter to control the inclination of the top handrail.

**Listing 4.7**: The operation that allows the creation of custom library parts. It is responsible by sending the GDL code and additional information such as type to the ArchiCAD plug-in.

```plaintext
(library-part name 2D-section 3D-section [master-code ""] [parameter-code "]") [type "Object"] [parent-id "ModelElement"] [properties '()])
```

**Listing 4.8**: Instantiating our created railing. It uses the added parameter to control the inclination of the top handrail.

```plaintext
(object "railing-test" (x 0) #:properties (list "lSlo" pi/6))
```

As we mentioned before, library parts have a limited set of attributes that limit the exploration of forms. In Figure 4.4 we shown how we are not able to modify the angle of only the
top handrail. However, by creating a railing that has an attribute to modify the top handrail’s angle, we are able to solve this problem. In Figure 4.5 we can see variations of the top handrail’s angle.

![Figure 4.5](image)

**Figure 4.5:** By taking advantage of our library part we are able to change the top handrail’s angle by changing a single parameter.

GDL is useful when the library does not have an object that meets the architect’s design. It allows the architect to lower its design level to one where he can model the geometry of an object, providing an almost similar degree of freedom that CAD tools have. However, it is important to note that GDL is a rather low-level language and, thus, programming library parts might not be adequate for the inexperienced programmer.

### 4.3 Retrieving Object Information

Another capability of our solution is the ability to programmatically retrieve information from an existing CAD or BIM model. This ability can open several lines of development. For example, we can have a model that contains only geometric information, such as lines and arcs, and create BIM elements from that information, such as walls and slabs. Furthermore, due to Rosetta’s functionality we can develop a single program that retrieves information from a CAD tool and uses it in a BIM tool.

Another line of development is to create a program from a model that was designed by traditional means. To do this, we need to extract information from a model, and translate that information into program elements that recreate the model. After that, we can do a refactoring process to make the program more legible and parameterizable.

Our solution supports this feature by providing operations that retrieve information from
several elements of ArchiCAD, such as, lines, walls, slabs, and others. These operations come with not only geometric properties but also BIM-specific information, such as material, levels, element associations (such as the wall-door relation), that can be used to translate the original designed elements to code.

Using this technique we are able to retrieve information from complex projects, such as the one visible in Figure 4.6, and treat that information as we see fit. As an example, we extracted information from the previous mentioned project to make a selective change on the walls, changing the material of the walls that have a specific material. This was achieved by reading the information of the walls from the project and recreating them with the new material, with the result visible in Figure 4.7. It is important to note that the walls keep their other properties, and keep the other elements that were associated with them, namely windows. If we were to do this change manually, we would have to select each wall and apply the new material, as ArchiCAD does not have a mechanism to select walls with a given material.

**Figure 4.6:** A complex project that was created by traditional means.

**Figure 4.7:** The result of making a selective change to the walls of the project.
4.4 Portability

Rosetta already supports portability between CAD tools. Naturally, we also want portability between its BIM back-ends. At the moment, Rosetta supports two BIM applications: Revit and ArchiCAD, the back-end we introduced. Although both are BIM applications that rely on the same elements and rules, in some cases they handle those concepts quite differently. Therefore, we want our primitive operations to normalize some of the differences between these applications.

As previously stated, our solution uses some abstractions already present in Rosetta, such as coordinate systems. The implementation of the Revit’s primitive operations were also based on the same abstractions. The use of these abstractions eased the portability of several operations, but it was not enough due to the differences in how BIM information is handled in each application, as we have seen in Section 1.4. For example, a Revit family determines information regarding the element, such as material, thickness, and others, while in ArchiCAD there are no families. Furthermore, the family parameters can be locked, meaning that they have specific family values and the user cannot change them. Most of the times, if the user wants to change a particular aspect of a building element, such as material or thickness, it has to change either to a specific instance of the family or change the family itself. This is not the case for ArchiCAD, as each element has those properties independent of each other, and not grouped by a family [29].

Despite the differences between the tools, we were able to create portable operations, and the wall is an example of the available portability. In Listing 4.9 we have the wall operations of Revit and ArchiCAD. The first level of portability is achieved by providing default values for every parameter. This not only allows the user to quickly experiment with the operation and see immediate results, but also allows for portability between Revit and ArchiCAD, because the default values will be used.

The next step in portability is to extract some information from the Revit family that can be translated to the ArchiCAD’s operations. For example, a family in Revit has a certain thickness that can be retrieved to be used as an argument for ArchiCAD. However, there are certain parameters that cannot be retrieved from the family, as is the material’s case.

```
1 (walls guide
2  [bottom-level (current-level)]
3  [top-level (upper-level bottom-level)]
4  [family (default-wall-family)]
5
6 (walls guide
7  [thickness (default-wall-thickness)]
```
Besides handling the properties of elements differently, BIM applications have more differences, such as the representation of building floors. In the case of Revit they are called levels, and in the case of ArchiCAD they are called stories, but the differences do not end with the name. As we have seen in Section 1.4, Revit’s levels are created given a name and an absolute height, which is relative to the origin point of the model. This is not the case for ArchiCAD, which requires a reference story, a height relative to that story and to state if the new story is going to be inserted above or below the reference story. To overcome these differences, we created a set of normalized modeling operations.

The first operation is the one that creates a level, designated level. The operation receives a height that will determine the level to create. If the level already exists, the function does not create it, but instead returns the level’s identifier. It is this operation that hides the differences in the creation of a level. In the ArchiCAD plug-in, we make the calculations to obtain the necessary information to create a level given an absolute height. This means we must choose the story that is closer to the given height, insert a story above it with the difference of the heights, and make sure that the new story has the correct height so the stories above are not moved. In Revit these additional calculations are unnecessary as the API creates its levels using an absolute height. The end result is that the creation of levels becomes identical in both BIM applications.

We created another abstraction related to the levels, called current-level. Our abstraction layer has always a level that will be used to create the elements. That level can be obtained by using current-level. Every time current-level changes, all elements will be correctly created in the new level. Furthermore, by having a level that is always applied by default to all elements, the user does not need to specify a level, which improves the portability with CAD tools.
Finally, we have an operation called `upper-level` that, as the name suggests, creates a level above the current level. In **Listing 4.10** is an example of how the operations can be used to manipulate levels. We create all the levels first (line 5), and then iterate through them (line 7). We update the `current-level` in every iteration (line 8), and construct the elements we want, in this case slabs (line 9) and walls (line 10). Finally, we create a roof (line 12) that must be placed in the level above the last level of the slabs, and we achieve that by using the `upper-level` function to create the appropriate level. The final result is visible in **Figure 4.8**.

```racket
1 #lang racket
2 (require rosetta/archicad)
3 (define height 15)
4 (define wall-height 3)
5 (define levels-list (map level (range 0 height wall-height)))
6
7 (for ([lvl levels-list])
8   (parameterize ((current-level lvl))
9     (slab (list (xy 0 0) (xy 5 0) (xy 5 5) (xy 0 5)))
10    (walls (list (xy 0 0) (xy 5 0) (xy 5 5) (xy 0 5))))
11
12 (roof (list (xy 0 0) (xy 5 0) (xy 5 5) (xy 0 5))
13   (upper-level (last levels-list) wall-height))
```

**Listing 4.10**: An example of the usage of levels.

![Figure 4.8: A simple tower, which is the result of our program that manipulates levels.](image)

### 4.5 ArchiCAD Specific Functionality

BIM applications share a great deal of functionality, but there are some operations that are tool-specific. Although our solution strives to achieve portability between back-ends, we also
want to support tool-specific functionality. As such, we also provide modeling operations and parameters that work only when the chosen back-end is ArchiCAD.

In ArchiCAD, every element has a material attribute, that determines several properties of the element, such as manufacturer, physical conductivity, density, heat capacity, and others. ArchiCAD’s materials are divided into two categories, either building materials or composite materials. The building materials are created from one surface, for example, brick, glass, concrete, and many others. The composites are built from a group of building materials, for example, a composite of plaster and brick.

To provide ArchiCAD’s materials we added two parameters in every operation that creates an element: one to determine the material; and another to control what type of material (either building material or a composite). Although these parameters are not portable between other back-ends, they have a default value specific for each element, that eases the portability of the modeling operations that use materials. In **Listing 4.11** we create several walls to exemplify how the material parameter works. The produced walls can be seen in **Figure 4.9**.

| 1 | (wall (xy 0 20)(xy 5 20)) |
| 2 | (wall (xy 0 30)(xy 5 30) #:material "Plywood") |
| 3 | (wall (xy 0 40)(xy 5 40) #:material "Brick Double Plastered" #:type-of-material "Composite") |

**Listing 4.11**: The creation of several walls using different materials and type of materials.

![Figure 4.9](image_url): The walls created from the code above, which use different materials.

Another functionality specific to ArchiCAD is the creation of profiles that can be used in specific elements, namely walls, beams and columns. A profile is a 2D section that will modify the element that is applied to. Through profiles the user has a higher degree of control over the geometry of the wall, beam and column elements. He can explore forms that are more uncommon and apply them to those elements.
To provide this functionality we created an operation for profiles. A profile can be created, and then applied to walls, beams and columns. The modeling operations for those elements are ready to receive a profile as a parameter. The application of a profile to a wall can be seen in Listing 4.12 and the resulting wall in Figure 4.10.

```
1 (wall (xy 0 20)(xy 5 20) #:profile-name "Two Story Bent Wall")
```

**Listing 4.12:** A profile being applied to a wall, in its creation process.

![Figure 4.10: A profile being applied to a wall.](image)

To avoid portability conflicts in those modeling operations the elements have, by default, no profile. This allows the operations to be portable, and if the user wishes to relinquish portability, he can do it by simply using the non-portable parameters.

### 4.6 Summary

In this section we explored GD for the BIM paradigm while addressing challenges that were faced during this thesis.

We started by comparing the CAD and BIM paradigms. Follow by an explanation of how the changes between them can be mitigated by abstractions built on top of geometric operations, and by translating information from BIM parameters into geometric information. Next, we
explained some of the associations that BIM tools support, such as the link between walls and the doors they host. Finally, we showed how sometimes BIM applications can be more restrictive than CAD ones, for example in manipulating the geometry of elements.

As a solution to the limits in modeling geometry, we introduced the ability to create ArchiCAD’s Library Parts. By creating library parts, the user is lowering its design level to one where he can have control over the geometry of elements. However, this requires the user to know GDL, an ArchiCAD-specific language.

Next, we showed how retrieving object information from an application, either a CAD or BIM one, can open several lines of development. In one hand, we can simply retrieve information from a manually created model, translate it into a rudimentary program, and apply a refactoring process to make it more abstract and more parametric. On the other hand, we can make a connection between CAD and BIM tools by retrieving geometric information from a CAD application and use it to create BIM objects. To support this functionality our solution extends Rosetta with a set of operations that can retrieve information from ArchiCAD.

The next topic we addressed was portability. As Rosetta already supports portability between CAD tools, we also wanted to support it with BIM applications. To achieve portability between ArchiCAD and Revit we started by providing default values for every additional or different parameter in the modeling operations. The next level in portability is to extract information from the Revit family and translate it to ArchiCAD’s operations.

Finally, we elaborated on ArchiCAD-specific functionality, such as the material property, which can be applied to any element in ArchiCAD, giving it additional information important for construction and energy analysis. The other specific feature we addressed was profiles, which modify the 2D section of a small set of elements.

In the next section we evaluate our solution, by presenting several case studies that explore the solution’s adequacy and performance.
## Contents

| 5.1  | Comparison with R-G-A                         | 53 |
| 5.2  | Going Beyond R-G-A                            | 55 |
| 5.3  | Portability Between BIM Applications          | 58 |
| 5.4  | Performance of ArchiCAD and Revit             | 60 |
| 5.5  | Portability Between CAD and BIM Programs      | 61 |
| 5.6  | Performance of CAD and BIM Applications        | 64 |
| 5.7  | Exploring ArchiCAD’s Functionality            | 64 |
| 5.8  | Summary                                      | 70 |
In this section we evaluate our solution based on: its adequacy in producing BIM models by first comparing it with Rhino-Grasshopper-ArchiCAD (R-G-A), and then by showing an example that goes beyond what has been made with R-G-A; its portability with both BIM and CAD applications; its performance against other BIM and CAD tools; and its support of ArchiCAD-specific functionality.

5.1 Comparison with R-G-A

In this section we compare our solution with R-G-A by producing an example described in its manual. The example is a tower composed of rotated slabs connected with beams. The R-G-A program is parameterized, allowing the change of the slab’s shape, which in turn auto-adjusts the beams. In Figure 5.1 it is possible to see a variation of the tower.

![Figure 5.1: The tower produced by R-G-A.](image)

It is important to notice that the example of the R-G-A’s manual contains a bug related to the rotation of the tower. When the tower has no rotation, the grasshopper program fails to execute and no changes are made in the ArchiCAD model. This happens because ArchiCAD cannot handle beams that are completely vertical. This problem was verified in the version 18 of ArchiCAD, the one used in this thesis.

In our solution we decided to initially replicate the same model, i.e., use beams to connect the slabs. Although we were able to create the model, it had graphical glitches due ArchiCAD’s inability to produce vertical beams. Hence, we decided to switch to columns instead of beams, as ArchiCAD can easily create vertical columns. A comparison between the two towers can
be seen in Figure 5.2, illustrating the difference between using beams or columns. By using columns we were able to reproduce the same tower, but without the beam’s bug, visible in Figure 5.3.

![Figure 5.2: On the left: vertical beams. On the right: columns.](image)

![Figure 5.3: Our version of the example tower.](image)

With this example we have shown a small part of what our solution is capable of achieving. In the next case study, we present a model that is more complex than any example present in R-G-A’s manual.
5.2 Going Beyond R-G-A

The previous example, despite using some BIM functionality, it was limited to two building components: slabs and beams. To further test the adequacy of our solution in creating BIM models, we chose a more complex model, the Absolute Towers.

The Absolute Towers, visible in Figure 5.4, are ideal to explore the advantages of GD, as the building has a high degree of repetition and a controlled variation in its shape. This variation is present in its slabs, as these have a rotation that changes according to their floor level, which in turn affects several other elements, including walls, railings, and others.

![The Absolute Towers of Canada, developed by the MAD Architects.](https://www.daniels.utoronto.ca)

To recreate the Absolute Towers in ArchiCAD, we used the most adequate BIM elements so that they match each building component. Contrary to the previous example, the final model has a wide range of different BIM elements, namely slabs, walls, columns, railings, stairs, landings, and roofs. To better understand the generated model, we will describe how we created the major components of the building.

To model the slabs we used a super-elliptical form, and rotated their points to fit correctly with the tower’s shape. Each tower has different rotations that are applied to their slabs. In Figure 5.5, we can see the slabs of each tower with their different rotation angles.

Following the slabs, we modeled the structural walls, that are contained in a smaller super-ellipse, as visible in Figure 5.6. If we were implementing this using a CAD back-end, one tempting approach would be to model extra-long walls which we would then intersect with a
super-elliptical cylinder that defined the interior of the building. However, in a BIM tool there is no intersection operation between elements.

To solve this problem, we implemented a virtual intersection between walls and slabs. To do so, we compute the intersections between the slab’s frontier line and the walls’s center line, followed by the alteration of the wall so that it correctly intersects with the slab. It is important to mention that the only input required from the users is the slab and wall. In Figure 5.7 we see the structural walls of a tower before and after the intersections.
After having the overall structure of the building, we added more details, such as columns, stairs, landings and railings. In order to accommodate stairs, the slabs need to have openings so that we can model the correct structure of the building. In a CAD tool, these openings would be achieved by performing a subtraction to the slab, however, BIM tools do not have subtractions. They support the creation of openings in elements, such as slabs. Hence, we developed a modeling operation that creates openings in slabs. In Figure 5.8 we can see a render that shows some of the added objects, namely, the stairs, landings and railings.

Figure 5.7: The structural walls of an Absolute Tower, showing how the intersection with the inner-slab produces the final shape.

Figure 5.8: A render of a slice of the building, showing some of the added details, such as stairs, landings and railings.
Using our program we created the two Absolute Towers, with all the above mentioned BIM objects. A render of the model can be seen in Figure 5.9. Furthermore, because we parameterized our program, we are able to explore variations of the original concept with little effort. For example, we can alter the overall shape of the building by changing the slab’s shape. This automatically adjusts the position and shape of every other building component, contrary to the traditional approach, where it would be needed to manually adjust all building components. In Figure 5.10 we can see two towers that have different slab shapes.

![Figure 5.9: A render of the Absolute Towers created by the program developed for the case study.](image)

Finally, the Absolute Towers’ case study was also used to test portability between ArchiCAD and Revit [29]. In the next section we discuss how we achieved portability between the two BIM applications by presenting a new case study and further examining the Absolute Towers.

### 5.3 Portability Between BIM Applications

In order to evaluate the portability of our solution, we present two case studies that show how the same program can produce the same model in two different BIM applications.
The first case study to test portability is of a building with a facade that follows a sinusoidal surface, visible in Figure 5.11. It relies only on four building components, namely, slabs, beams, columns and roofs. Due to this, it was the ideal starting point to test portability.

The program relied on the default values that each element had, in the case of ArchiCAD for its properties, such as material, and in the case of Revit for its family. This allowed us to have one program capable of producing two models in two different BIM applications. The final models can be seen in Figure 5.12.

The next case study is the previously seen Absolute Towers, due to having a larger number of building components. To achieve a portable program between ArchiCAD and Revit, we had to give up of some operations that were not available in Revit. For example, in Revit the resulting model lost railings and stairs, with their respective landings. The models can be seen in Figure 5.13.

These two case studies show that is possible, to a certain degree, to have portable programs between different BIM applications. This allows us to: reuse programs to create models in different BIM applications, saving time in porting programs; choose the desired application based on the architect’s preference, or on the tool’s performance. This last point leads us to our next section, where we test the performance of each BIM application, in order to understand what gains we would have in using a certain BIM tool over the other.
5.4 Performance of ArchiCAD and Revit

Throughout the development of our solution we noticed performance disparities between ArchiCAD and Revit. This can be attributed to the implementation of the applications and to the languages of each API, C++ for ArchiCAD and C# for Revit. Despite this, we deemed important to show a time comparison between the two applications, as we provide an option for architects to switch between the two back-ends, allowing them to take advantage of the one with the best performance. To this end, we timed the creation of the two previous case studies, in both ArchiCAD and Revit. The results are shown in the graph of Figure 5.14.

As it is possible to verify, ArchiCAD has a better performance than Revit. However, Revit is capable of working in a non-blocking mode, allowing the user to interact with the tool while the model is being created. This is not the case with ArchiCAD, as it blocks any interaction.
while creating the model, due to the way the API is designed.

Despite the performance of ArchiCAD, the time it takes to create a model is still greater than any CAD tool. Hence, we found the need to have the same program work both with BIM and CAD tools. In the next section we describe the process of having a BIM program work, with minimal changes, in CAD applications.

5.5 Portability Between CAD and BIM Programs

Having a BIM program that works with CAD applications, without needing to modify neither the operations nor the structure of the program, has several advantages: the program keeps the BIM information, regardless of what application it produces the model in; and we can take advantage of the performance of CAD applications in early stages of a project, where exploration is important.

The case study we used to test the CAD-BIM portability of our solution is the previously seen building with a facade, due to having a limited set of building components. In fact, this building was first created for CAD tools. We had to port the program to work with BIM tools, by following these steps: (1) identifying the building components needed to create the building; (2) moving the dimensional properties of the elements to the families of each building components; and (3) organizing the building by levels, based on the height of each floor.

This porting process was heavily facilitated due to intermediate abstractions that the CAD program had, which clearly identified each part of the building. For example, in Listing 5.1 we can see the abstraction for the facade element. To port this operation to BIM we modified the right-cuboid operation to beam, and applied a family to it, as visible in Listing 5.2.
Figure 5.14: The time that each back-end took to create each of the case studies. The difference in performance is possibly attributed to the applications themselves, and to the languages that each API uses, C++ for ArchiCAD and C# for Revit.

Listing 5.1: The intermediate abstraction for the facade element of the original CAD program.
Let's consider the following code snippet:

```
(beam pa pb)))
```

This snippet represents the BIM version of the facade element. It was needed to specify the family of the element (line 4) and use the correct building component (lines 10 and 16).

The next step is to have the program work with any CAD or BIM application, without having to revert to the original operations. To achieve said portability, we created operations in Rosetta that would be capable of producing the same BIM building components in CAD applications. Those operations relied heavily on extracting information from the family of each element and translating it into the geometric attributes of the existing CAD operations, for example, the dimensions of beams into width and thickness of right-cuboids, and the level's height into the height of the element. By doing this we were able to have our program produce the same model in a wide-range of back-ends, visible in Figure 5.15.

![Figure 5.15: The same program, which contains BIM information, generated the model in different back-ends. Starting from the top left: OpenGL; Rhino; ArchiCAD; SketchUp; AutoCAD; and Revit.](image)

It is worth noting that these operations can be used in other CAD-specific programs in order to turn them into CAD and BIM programs. This is especially helpful because there is already a large collection of CAD programs, which can use these operations to work with BIM applications.

This level of portability between CAD and BIM applications allows us to have a workflow that fits the architects’ needs. We have the CAD’s better performance for exploration in the early phases of the project, and the BIM information needed for construction processes in the later phases of the project.

In the next section we take advantage of this level of portability to evaluate the performance of each tool.
5.6 Performance of CAD and BIM Applications

BIM applications are noticeably slower than CAD tools, due to having richer elements, that have both properties and rules. As such, the creation of models in BIM can have an impact on the productivity of architects, as any change on the model would imply having to wait for the model to be rebuilt. This has a major impact in the early phases of the project, when exploration is paramount, and it can also be important in the later phases of the project, when changes to the model are needed to meet the client’s requests.

To mitigate this problem, we have shown how Rosetta’s portability allows us to switch between its back-ends, either CAD or BIM, at any stage of the project. To understand how big the performance gap is between CAD and BIM tools, we used the previous example (Figure 5.15) to compare the time each back-end took to create the same model. In the graph of Figure 5.16 we confirm the worse performance of BIM tools.

![Time Comparison Graph](image)

**Figure 5.16:** To test the performance of Rosetta’s back-ends we created the model seen in Figure 5.15 and compared the time each one took. The results are shown in this graph, where it is visible that the BIM applications have the worst performance.

In the next section, we put aside the portability concerns to evaluate how our solution allows the exploration of ArchiCAD’s specific functionality.

5.7 Exploring ArchiCAD’s Functionality

Our solution, despite supporting portability between several back-ends, also supports ArchiCAD’s specific functionality. To evaluate this, we present two additional case studies
that go beyond the previous ones, by using operations that were developed specifically for ArchiCAD.

5.7.1 Shell of Louvre Abu Dhabi

The shell of Louvre Abu Dhabi, visible in Figure 5.17, was originally developed using generative modeling technology [30], taking advantage of the F# programming language and the CAD application Rhino. This fact alone was an incentive to try modeling the shell using our solution. Before going into our produced shell, we explain the information that we used to replicate the shell.

![The Abu Dhabi shell (source: http://openbuildings.com/).](image)

**Figure 5.17:** The Abu Dhabi shell (source: http://openbuildings.com/).

The shell itself is composed of several layers and a steel framework in between the layers, as it is visible in Figure 5.18. All of the layers are made of a single element that is repeated throughout the layer’s shape. The element is a star formed of connected beams, which can be seen in Figure 5.19. Additionally, each of those beams has a profile that was modeled to reflect the building component, also visible in Figure 5.19. Finally, each layer is rotated and scaled to produce the unique pattern of the shell.

To model our shell we started by developing the layers without using any profile. After having the layers created, we added profiles to beams, providing an additional method of visualization. Figure 5.20 shows our profile applied to one beam. Unfortunately, we could not create the full shell using the detailed profile due to a limitation of the BIM application.
as it could not handle that level of detail applied to the large number of beams of each layer. Although we do not have all the information regarding the transformations applied to each layer, we were able to create an approximation of the original shell. In Figure 5.21 we have an overview of the shell, and in Figure 5.22 we have a more detailed view that shows the complexity of the shell’s structure.

The next case study is the Rotterdam’s Market Hall, where we continue the exploration of different profiles to achieve a more complex shape.
5.7.2 Market Hall Rotterdam

The Market Hall of Rotterdam, visible in Figure 5.23, was designed by MVRDV. This building was previously modeled using Rosetta and a CAD application by an architecture student in her master's thesis [10]. The generated model was heavily based on surfaces and solids, both manipulated by transformations and lofting operations to achieve the unconventional shape of the building. For this experiment, instead of adapting the student’s program, we created one from scratch with BIM functionalities in mind. Additionally, due to the building’s unconventional shape, we were able to explore the limits of the ArchiCAD application and its API.

The model that our program generates can be seen in Figure 5.24, and is composed by
common objects, such as walls and slabs, but also some uncommon ones like curtain walls and shells. We started by creating two super-ellipsoid shapes that dictated the geometry of the walls. To impose the super-ellipse to the walls we had to create profiles for each wall that follow the required curve. In Figure 5.25 we can see all the walls that have a custom profile.

The curtain walls were used to build the front, and back facade of the building. The curtain walls are divided by levels, instead of being a single element that goes from the bottom floor to the top floor and that follows the building’s shape. This was required due to the inability of changing the curtain wall’s shape through the API. Furthermore, for similar reasons, we cannot use additional elements in the curtain wall, such as junctions or doors. Due to these unforeseen problems, we modeled an alternative version of the building using walls instead of curtain walls. This allowed us to see what the result would be with the revolving doors of the
Figure 5.25: All the walls of the building that have a custom profile.

entrance and exit of the building, as is visible in Figure 5.26. To model these doors we simply chose their type and inserted them into each wall, which requires just one line of code, while in CAD we needed to model every detail of the door. The original program required 34 lines of codes just for that door.

An advantage that stems from the properties of the elements, such as the type of the door, is that we can change them without having to modify the structure of the program. In the CAD version of the program, one would need to create a new door function and use higher-order functions to produce a different door type. In our version, we simply need to change the default door type, as visible in Listing 5.3.

Figure 5.26: The detail of a rotating door of our model.

```
1 (parameterize ((default-door-type "A Different Door Type")) (market-hall (u0) 120 6 11))
```

Listing 5.3: Changing the doors’ type by using the modeling operation’s type parameter.
Besides curtain walls we also used shells, another uncommon element. These are normally used to create roofs that have a more complex shape, which is exactly our case. They still have BIM information, like materials, which is ideal for our BIM model. By using shells we also gained the ability to use an operation that trims walls with shells. This allowed us to control the top level walls’ shape without requiring profiles, as is visible in Figure 5.27.

![Figure 5.27: The process of trimming the walls using a shell.](image)

### 5.8 Summary

In this section we went through several experiments that evaluated the capabilities of our solution. We started by showing that our solution is capable of not only doing what R-G-A can do, but also go beyond it by producing more complex models. Next, we evaluated the portability of our solution with other BIM applications, by presenting a case study that works both on ArchiCAD and Revit.

The next topic was the performance comparison of ArchiCAD and Revit, which shown the better performance of ArchiCAD. Although this difference is, in great part, due to the implementation of the BIM applications themselves and their respective APIs, it is still worth mentioning their performance differences. Despite the better performance of ArchiCAD, it is still not close to the performance of CAD tools. Thus, we shown how to create a program that works both with CAD and BIM tools, allowing us to take advantage of the performance of CAD tools in the early stages of a project, where exploration is paramount.

Finally, we evaluated how our solution can explore ArchiCAD-specific functionality, despite supporting portable operations. To this end, we described two case studies that took advantage of ArchiCAD-specific operations to produce complex models.

In the next section we conclude our thesis and propose possible future work.
Conclusion

Contents

6.1 Future Work ................................................................. 74
6.2 Contributions ................................................................. 75
Architects have moved from traditional tools, like pen and paper, to digital tools that improve their design process, such as Computer-Aided Design (CAD) and Building Information Modeling (BIM) applications. However, there are architects that go one step further, and use Generative Design (GD) to create their models.

GD is an algorithmic approach to design that opens a new development path for architects that want to program their models. By using GD, architects are able to mechanize repetitive, time-consuming, tedious and error-prone tasks, simplify the creation of complex geometry, and easily explore alternatives to the original design.

CAD tools were the first to be used with GD, with several tools available, such as, Grasshopper, Visual LISP, and RhinoScript. However, with the rise of BIM, developers recognized the need to advance to this new paradigm. This resulted in the creation of several applications that combine GD with BIM as we have seen in Section 2, such as Lyrebird, Dynamo, Generative Components and others.

In our analysis of these tools we shown that many rely on visual languages, which despite being fit for beginners, have scalability issues. Others, use textual programming languages that are either too complex for beginners or obsolete. Furthermore, none of these tools supports portability, binding users to their system and application.

With our solution, we want users to not only choose the application they want to use, but also the language they want to program. For these reasons Rosetta was a good environment to implement our solution. It already supports several programming languages that are fit for beginners, such as Racket, Python and Processing, and that can scale to complex programs. Furthermore, Rosetta supports an educational Integrated Development Environment (IDE), DrRacket, that has a Read-Eval-Print Loop (REPL), a debugger, syntax highlight and other features. Finally, it has portability between its supported back-ends, which include an OpenGL back-end, CAD applications, and a single BIM application, Revit.

The implementation of our solution is an extension of Rosetta, as we have seen in Section 3. Its architecture is composed of an abstraction layer with numerous modeling operations, a plug-in for the new back-end ArchiCAD, and a communication channel that allows the information exchange between the other two components.

With the development of our solution we explored the intricacies of GD for the BIM paradigm, which we examined in Section 4. We discussed: (1) how the differences between CAD and BIM affect programming; (2) how we can create custom objects; (3) how we can have a bi-directional exchange of information in order to not only create a model but also retrieve useful information from it, thus opening new lines of development, such as creating programs from existing models, use geometric information from a CAD model to produce a BIM model.
and make changes through programming on a model created by traditional means; (4) how portability can be achieved not only between BIM tools but also with CAD tools; and last but not least, (5) how we can still have specific ArchiCAD operations for more specialized models. This allows the user to choose between a portable but less detailed program or an unportable one that fully explores ArchiCAD’s capabilities.

Finally, in Section 5 we evaluated our solution in terms of: adequacy; portability; performance; and support of ArchiCAD’s specific functionality. We shown the adequacy of our solution in producing BIM models by designing several models that have diverse BIM elements. We tested the portability of our solution not only with BIM applications but also CAD tools. Finally, we demonstrated that despite supporting portability, our solution is capable of accessing ArchiCAD-specific functionality to create specialized and complex models.

This thesis explored GD for BIM applications by implementing a solution that expands Rosetta with ArchiCAD, providing adequate and numerous modeling operations for the added back-end. Our solution valued portability, so it is easy to switch working applications, allowing users to take advantage of the performance of CAD tools in the initial phases of a project and of the BIM paradigm in the later phases. Furthermore, due to the flexibility of our solution, users can relinquish portability in order to access ArchiCAD-specific operations that are needed for more complex or specialized projects.

6.1 Future Work

There are still improvements that can be made to the presented solution, and work that can be done related to Rosetta. First, we plan to implement additional modeling operations that are available in ArchiCAD’s API but not available in Rosetta. For example, the skylight element was one element that we did not make available due to being unnecessary to the case studies. Furthermore, there are patches that can fix some of the bugs we encountered when developing operations, such as the lack of functionality in manipulating curtain walls.

Next, and in the same line of development, we plan to make the transition of our plug-in to newer versions of ArchiCAD. To do this it is required to update the plug-in to work with the newer versions of the API. By doing this, we would allow users to select their preferred version of ArchiCAD.

Our solution supports the creation of library parts, although it currently requires the use of GDL. It would be very interesting to create a compiler that uses GDL as a compilation target in order to allow users to program in a higher-level language, easing the design of library parts.
Another interesting line of work is to integrate Rosetta with ArchiCAD’s energy evaluation software, EcoDesigner STAR. By creating this possible connection we could test the energy performance of a building and eventually optimize its design to achieve a better energy efficiency.

Finally, it would be important to extend the current work on the Revit and ArchiCAD back-ends with the addition of new back-ends to Rosetta, such as Bentley’s BIM application, MicroStation. This not only would expand Rosetta’s potential user base but also further test its portability with the new back-end.

### 6.2 Contributions

During the development of this thesis we had the opportunity to publish a scientific paper:


This article focused on the portability of Rosetta, and was written in conjunction with two colleagues, one that is responsible for the Revit implementation and an architect that helped us with the case study and with her knowledge of architecture.
Bibliography


