Generative Design for Building Information Modeling

[Extended Abstract]

Guilherme Barreto
Instituto Superior Técnico
gbarreto1992@gmail.com

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 another BIM application, ArchiCAD, having into consideration portability between the already supported back-ends. Finally, we evaluate our solution in terms of its adequacy, portability, performance, and support for ArchiCAD-specific operations.

Index Terms—Programming; Generative Design; Computer-Aided Design; Building Information Modeling; Portability; ArchiCAD.

1 INTRODUCTION

Computer-Aided Design (CAD) applications consist mainly of drafting and modeling systems, used for blueprints, elevations, and sections of buildings. However, most modern CAD applications also support the design of 3D models. When compared to the traditional pen and paper approach, these applications provide a more efficient working process in architectural design and better documentation [1]. The shift to CAD also stimulated the desire for more complex structures, due to facilitating their conception [2].

More recently, Building Information Modeling (BIM) applications have been replacing CAD tools. These, as defined by the United States BIM Standard [3], are a digital representation of a building, rich with information that can be used throughout a building’s life-cycle.

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 [4], [5].

Although CAD and BIM applications revolutionized the architectural design process, there is still improvements to be made and Generative Design (GD) is one of them. GD is an algorithmic-based approach to design that allows the generation of forms or shapes through algorithms [6]. Using GD, instead of designing a building, one designs the system that designs the building [7]. The advantages that it brings are: the automation of repetitive and monotonous tasks; the improvement of the process to create complex geometry; and faster exploration of alternatives to the original design [8].

GD stimulated the development of tools that allow its exploration for CAD applications. Some examples of such tools are Grasshopper, Visual LISP, and Rhino-Script. Due to the recent adoption of BIM tools, some applications were also developed to support it, such as Lyrebird, Rhino-Grasshopper-ArchiCAD and Dynamo.

The majority of these tools rely on visual metaphors, that offer a beginner-friendly experience. Although very appealing for developing small programs, visual programming languages can have scalability issues, making them difficult to understand, use, and modify [9]. Furthermore, some 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) [10], [11], and supports several programming languages, such as Racket and Python, that can be used by both beginners and experienced users.

2 RELATED WORK

In this section we analyze several tools that already explore GD for the BIM paradigm, and discuss how they influenced our solution.
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 [12]. 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 harder to understand and change [9]. Figure 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#, or Python scripts by using a specific code block, which can have multiple inputs and outputs [13]. Despite the ability to create small textual scripts, this feature does not eliminate the scalability problems, because these 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, that the authors “hope has enough inputs and options to handle everything without being overwhelming” [14]. 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 Rhino-Grasshopper-ArchiCAD
More recently, on September 2015, Graphisoft announced the public beta version of its Rhino-Grasshopper-ArchiCAD (R-G-A) connection [15]. 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.

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 [16].

By using nodes and wires, users can create programs in Dynamo. 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 [16].

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 ArchiCAD API
Graphisoft – the company behind ArchiCAD – provides an Application Programming Interface (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 programming, 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, to create a simple slab the user must take into consideration the following: manage a session to create the element; load default values for the element; change the desired attributes; and manage memory allocation for the element and its geometric properties. Later in the document, we show how our solution can substantially simplify the creation process of a slab, or any other element.
2.4 Geometric Description Language

Another tool that Graphisoft provides is Geometric Description Language (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 [17].

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.

2.5 Analysis

In this section we present an analysis of how the previous tools influenced the proposed solution. Table 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>
<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>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Lyrebird</td>
<td>✔</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>R-G-A</td>
<td>✔</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dynamo</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>

Table 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.

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).

To better understand what each of the analyzed tools is capable of doing, we show in Table 2 a more detailed look at the supported operations of each tool.

<table>
<thead>
<tr>
<th>Application</th>
<th>Textual Language</th>
<th>Geometric &amp; BIM Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasshopper</td>
<td>VB.NET, C#</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>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>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>

Table 2. A detailed analysis of the operations supported by the several tools.

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. Furthermore, we chose languages that are fit for beginners, such as Racket [18] and Python [19]. The solution also supports the use of an educational IDE, DrRacket [10], [11], that will ease their learning experience.

Finally, we do not want a simple translation of the API's functionality. Our operations hide the complexity of the API without losing their 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.

3 Architecture

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 2 shows a components diagram of this architecture.

![Diagram of solution architecture](image)

Figure 2. A components diagram depicting the overall architecture of the solution.

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.

### 3.1 Rosetta

Rosetta has a front-end/back-end architecture. The front-ends are programming languages fit for beginners, such as Racket, Python, and Processing. The back-ends are also diverse, as Rosetta supports an OpenGL back-end for fast visualization and several CAD tools.

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 [20].

The modeling operations provided by the abstraction layer were developed for each of the supported back-ends. 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.

The portability between back-ends is also facilitated by the geometrical abstractions provided by the abstraction layer. An example of those abstractions are the coordinate systems, such as Cartesian, polar, and cylindrical, which can be used by any modeling operations. 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.

Finally, Rosetta uses DrRacket as its IDE, which was originally developed to teach programming to beginners [10], [11], 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

Our abstraction layer is an extension of Rosetta’s abstraction layer, as it provides modeling operations that were developed specifically for the BIM ArchiCAD. 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, which makes them easier and quicker to use. For example, the slab operation has only one required parameter, the guide, which determines its shape. The remaining parameters have default values making them optional to use, as is the case for bottom-level, material, thickness, and others. These parameters can be used for further customization of the element.

Our slab operation substantially simplifies the previously described process of creating a slab using the ArchiCAD’s API. All the necessary details that were mandatory using the ArchiCAD’s 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. In our implementation, as we have seen before, it is only required a list of points that will outline the slab’s shape (guide). In Listing 1 we create the slab of the previous example.

```
(slab (list (x 0) (x 5) (xy 5 5) (y 5)))
```

Listing 1. Creating a slab using our solution, which only requires a list of points.

### 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.

As a final remark, to exchange information between the plug-in and the abstraction layer, we developed a communication channel based on sockets that uses Google Protocol Buffers [21]: a language-neutral, platform-neutral, and extensible mechanism for serializing structured data.

4 Generative Design for BIM

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; and the abstractions that allow portability between BIM applications.

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 used to program for CAD will have to adjust their programming practices in order to program for BIM.

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 2. On the other hand, had the GD program we can create an entire floor using only boxes, 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.

As a final remark, to exchange information between the plug-in and the abstraction layer, we developed a communication channel based on sockets that uses Google Protocol Buffers [21]: a language-neutral, platform-neutral, and extensible mechanism for serializing structured data.

Listing 2. A program that create a floor, composed of a slab and four walls, using only boxes.

```scheme
(define (slab p length width thickness)
  (let ((v (p-p p1 p)))
    (box p
      (* (sin (pol-phi v)) thickness)
      (* (cos (pol-phi v)) thickness)
      (* (sin (pol-rho v)) thickness)
      (* (cos (pol-rho v)) thickness)
      (+xyz (+pol p (pol-rho v) (pol-phi v))))
    (* (sin (pol-phi v)) thickness)
    (* (cos (pol-phi v)) thickness)
    (+xyz (+pol p (pol-rho v) (pol-phi v))))
  (* (sin (pol-phi v)) thickness)
  (* (cos (pol-phi v)) thickness)
  (+xyz (+pol p (pol-rho v) (pol-phi v))))

(define (wall p p1 thickness height)
  (let ((v (p-p p1 p)))
    (box p
      (+xyz (+pol p (pol-rho v) (pol-phi v)))
      (* (sin (pol-phi v)) thickness)
      (* (cos (pol-phi v)) thickness)
      height))))

(slab (u0) 10 10 1)
(wall (x 0) (x 10) 1 3)
(wall (x 10) (xy 10 10) 1 3)
(wall (xy 10 10) (y 10) 1 3)
(wall (x 0) (y 10) 1 3))
```

Listing 3. The same program as before, but taking advantage of abstractions to be more legible and easier to understand.

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 will be an easy transition for him to use the BIM operations, as seen in Listing 4.

```scheme
(define (slab p length width thickness)
  (let ((v (p-p p1 p)))
    (box p
      (* (sin (pol-phi v)) thickness)
      (* (cos (pol-phi v)) thickness)
      (+xyz (+pol p (pol-rho v) (pol-phi v))))
    (* (sin (pol-phi v)) thickness)
    (* (cos (pol-phi v)) thickness)
    (+xyz (+pol p (pol-rho v) (pol-phi v))))
  (* (sin (pol-phi v)) thickness)
  (* (cos (pol-phi v)) thickness)
  (+xyz (+pol p (pol-rho v) (pol-phi v))))

(define (wall p p1 thickness height)
  (let ((v (p-p p1 p)))
    (box p
      (+xyz (+pol p (pol-rho v) (pol-phi v)))
      (* (sin (pol-phi v)) thickness)
      (* (cos (pol-phi v)) thickness)
      height))))

(slab-rectangular (u0) 10 10 1)
(wall (x 0) (x 10) 1 3)
(wall (x 10) (xy 10 10) 1 3)
(wall (xy 10 10) (y 10) 1 3)
(wall (x 0) (y 10) 1 3))
```

Listing 4. The program modified to work with BIM operators. We needed to change the slab operation to one that produces a rectangular slab.

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.

An advantage that comes from being able to create elements using only geometric attributes is that it encourages an iterative development for the architect. This means that he will be able to build upon a program that begins with geometric information to one that has BIM information. 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.

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 of the associative rules that the BIM paradigm has is in the creation of 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 wall, meaning that a change in the wall is appropriately propagated to the door, such as translations, rotations and other transformations. 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 their creation.

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 and place a door 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.

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 and door knob. 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.

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, price and others. This not only eases the creation of certain elements, such as doors, but also the construction process, as the architect can choose a product that 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 3 we show three different types of doors. This was achieved by only changing a single parameter, as we can see in Listing 5. If we were trying to do the same change but using GD for CAD, we would need to write a considerable amount of code to change the door’s structure.

\[
\text{Listing 5. Changing a door’s type using our solution.}
\]

\[
\begin{align*}
\text{door} & \left( \text{wall} \left( \text{list} \left( \text{y} 30 \right) \left( \text{xy} 5 30 \right) \right) \right) \\
& \left( \text{x} 2.5 \right) \\
\text{door} & \left( \text{wall} \left( \text{list} \left( \text{y} 40 \right) \left( \text{xy} 5 40 \right) \right) \right) \\
& \left( \text{x} 2.5 \right) \\
& \text{"Arch Double Door"} \\
\text{door} & \left( \text{wall} \left( \text{list} \left( \text{y} 50 \right) \left( \text{xy} 5 50 \right) \right) \right) \\
& \left( \text{x} 2.5 \right) \\
& \text{"Revolving Door 18"}
\end{align*}
\]

4.2 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. 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 [22].

Despite the differences between the tools, we were able to create portable operations. 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. This material contains additional information that is important for the construction process, such as manufacturer, physical conductivity, density, heat capacity, and others.

Despite the focus on portability, our solution allows users to explore ArchiCAD’s specific features, like the element’s material. Later in the document, we show a case study that explores ArchiCAD’s materials and profiles to create a complex model.

5 Evaluation

In this section we evaluate our solution based on its adequacy, portability, performance and support of
ArchiCAD-specific operations, which are needed for more complex or specialized projects.

5.1 Adequacy
To test the adequacy of our solution in creating BIM models, we modeled the Absolute Towers, visible in Figure 4. They 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.

Figure 4. The Absolute Towers of Canada, developed by the MAD Architects. (source: 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. Furthermore, we were able to implement an operation that the APIs of both ArchiCAD and Revit did not support, namely, the intersection between walls and slabs. With this operation we were able to produce the correct structural walls, as seen in Figure 5.

Figure 5. 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.

Using our program we created the two Absolute Towers, which can be seen in Figure 6. This case study was also used to test portability between ArchiCAD and Revit [22]. 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.

5.2 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 7. 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.

Figure 7. A render of the building with a sinusoidal facade.

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 8.

Figure 8. The produced model in ArchiCAD (left) and in Revit (right).
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 9.

Figure 9. The Absolute Towers in ArchiCAD (left) and in Revit (right).

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 the next section, where we show how we can achieve a program that can both work with CAD and BIM applications.

5.3 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, which was facilitated by the intermediate abstractions already present in the CAD program; (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.

Having the new BIM program done, the next step is to have it 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 translate 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 10.

Figure 10. 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.

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.

5.4 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 are needed to be made 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 10) to compare the time each back-end took to create the same model. In the graph of Figure 11 we confirm the worse performance of BIM tools.

Figure 11. To test the performance of Rosetta’s back-ends we created the model seen in Figure 10 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.

5.5 Exploring ArchiCAD’s Functionality

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

The Market Hall of Rotterdam, visible in Figure 12, was designed by MVRDV. This building was previously modeled using Rosetta and a CAD application by an architecture student in her master’s thesis [7]. 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 13, 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 would dictated the geometry of the walls. To impose the super-ellipse to the wall we had to create profiles for each wall that would follow the required curve.

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 entrance and exit of the building, as is visible in Figure 14. 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.

The Market Hall of Rotterdam. (source: http://www.archdaily.com/)

Figure 12.

A render of the Market Hall model produced by our program in ArchiCAD.

Figure 13.

The detail of a rotating door of our model.

Figure 14.
6 Conclusion

Architects have moved from traditional tools, like pen and paper, to digital tools that improve their design process, such as CAD and BIM applications. However, there are architects that go one step further, and use 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.

Several tools were developed to explore GD with both the CAD and BIM paradigm. In our analysis of these tools we shown that many of them rely on visual languages, which despite being fit for beginners, have scalability issues. On the other hand, there are some tools that support textual programming languages, but that end up being either too complex for beginners or obsolete. Furthermore, none of these tools has portability, binding users to their system and application.

With our solution we support GD for BIM applications by expanding Rosetta with ArchiCAD, providing adequate and numerous modeling operations. Through Rosetta, we gained access to beginner-friendly programming languages that can scale to complex programs and an educational IDE. Our solution valued portability, allowing them 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. Finally, our solution allows users to relinquish portability in order to access ArchiCAD-specific operations that are needed for more complex or specialized projects.

6.1 Future Work

We plan to continue improving Rosetta’s features by implementing additional modeling operations that are available in ArchiCAD’s API but not available in Rosetta, such as an operation to create a skylight element.

Although our solution supports the creation of library parts, it currently requires the use of GD. 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.

An interesting line of work is to integrate Rosetta with ArchiCAD’s energy evaluation software, EcoDesigner STAR. By doing this, we can 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.

References


