The Expressive Power of Programming Languages for Architecture

[Extended Abstract]

Maria João de Maya Gomes Cunha e Sammer

Master of Science Degree in Architecture

May 2019

1. INTRODUCTION

From the birth of the discipline itself, architecture has used different tools and methodologies to be represented, explained, and sold throughout History.

Drawings and sketches are primary examples that have always been the base of the expression of intentions of architects. Nowadays, the architectural process still leans on these drawings and sketches to create the initial concept, to structure constraints, to study dynamics, and to build up ideas, that are later consolidated in technical drawings and models (figure 1.).

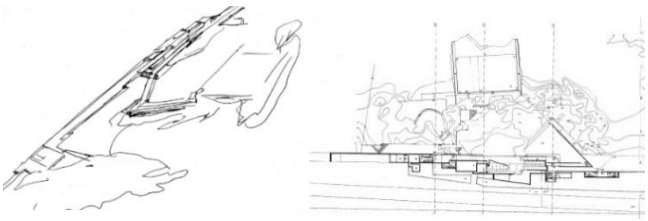

Fig. 1. Hand-drawing (left) and technical drawing (right) of Swimming Pools in Leça da Palmeira by Álvaro Siza Vieira, Leça da Palmeira, Portugal, 1967.

Architecture has always been influenced by new technologies in the fields of engineering, since it needs to deal with structure, lighting, acoustics, among others. More recently, another paradigm started influencing architecture with the ability to connect every contribution mentioned above: computation.

Computers are already an integral part of the common practice of architecture nowadays, as when mentioning technical drawings, one usually is referring to documents digitally produced by Computer-Aided Design (CAD) and Building Information Modeling (BIM) tools (figure 2).

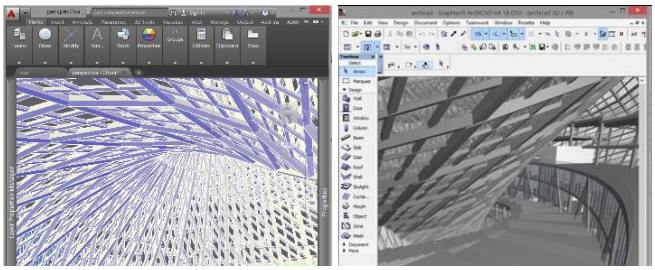

Fig. 2. CAD tool (left) and BIM tool (right) work environments.

CAD and BIM documentation is essential not only for further design stages, such as the construction of the project with the integration of other specialties, but also for the generation of 3D models. These digital models can then be used either (1) for their geometry, to showcase the overall look of the project and to produce virtual images or (2) to be assessed through computational analysis tools, according to their performance regarding lighting, acoustics, or structure.

More recently, computation opened a wider door for other possibilities that emerge from the fields of programming, introducing a core concept: algorithmic design (AD). AD represents an alternative approach to the manual manipulation of geometry in a modeling tool. Instead of using the tool's commands to create and modify geometry by hand, the architect builds an algorithm, a program, that defines a model that is then generated in the chosen tool.

Therefore, AD relies on the use of programming languages (PLs) to define rules and constraints within an algorithm that, when executed, generates the digital model of an idea. These programming languages can either be visual (VPLs) or textual (TPLs).

In the current practice of architecture, there are already several studios around the world that resort to algorithmic approaches to develop their projects in whole or in part. Representative examples of that are the Hangzhou Tennis

Center by NBBJ (Hangzhou, China, 2013) and the Morpheus Hotel by Zaha Hadid Architects (figure 3). Nevertheless, there is a clear propensity to the use of VPLs over TPLs among architects.

Fig. 3. Morpheus Hotel by Zaha Hadid Architects, City of Dreams, Macau, 2018.

VPLs, such as Grasshopper, are more intuitive for beginners and, therefore, easier to learn. Furthermore, they are equipped with user-friendly features and mechanisms that allow a better understanding of what is being programmed and a more dynamic and direct interaction with the modeling tool in use. However, despite these advantages, there is a serious handicap that appears whenever the program reaches a certain level of complexity: it becomes hard to understand (figure 4) and, therefore, difficult to maintain and develop, and some of these features simply stop working.

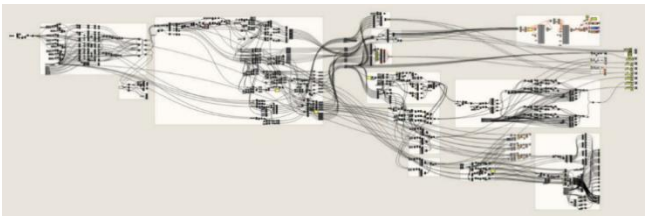

Fig. 4. Complex program in Grasshopper that generates the shell of the Hangzhou Tennis Center.

On the other hand, TPLs offer greater flexibility in creating and managing complex projects, surpassing the limitations of VPLs. The main obstacles to the use of TPLs are the demanding learning process and the lack of mechanisms that do facilitate programming for architects. However, if these difficulties can be overcome by simpler implementations and a more user-friendly learning process, the inherent advantages of mastering a TPL are believed to be rewarding regarding the ease with which complex projects can be developed, explored, and maintained.

Nevertheless, an in-depth understanding of the overall tendency for the use of the visual approach over the textual one is still lacking.

With the development of this thesis, we integrate a methodology that allows evaluating the relative advantages and disadvantages of VPLs and TPLs in their current application for architecture. Furthermore, the conclusions taken from the evaluation of this investigation allow the proposal of guidelines regarding two main aspects: (1) a methodology that relates the scale of complexity of a project and the approach that should be adopted to meet the correspondent objectives, and (2) the suggestion of future developments in TPLs that both dilute the barrier that exists between architects and textual programming approaches and develop the appeal and intuitiveness of TPLs, so that they can compete and surpass VPLs from earlier stages of the development of a project.

2. CONTEXTUALIZATION

Representation methods in Architecture

Since the birth of mankind as a collective species, communication has been the key to its survival and development. Alongside with the first sounds and gestures, drawings were made inside caves to express reality, to communicate intentions, or even to tell stories.

We came a long way since the Ancient era and, up to the present day, the methods and techniques we use to express ourselves have been developed greatly. Nevertheless, it becomes clear that, throughout all these centuries of evolution, the expression of architectural thoughts and ideas has always been made through visual methods: drawings, plans, sections, schemes, physical models, among others.

After the first drawings of the Ancient Era, the Greco-Roman period further developed city plans, besides introducing the idea of systematization with Vitruvius, the first time the textual approach was used as a representation method for architecture. After the Medieval period, where orthographic projections like plans and sections were mainly used, the Classicism became more invested in the systematization of classical patterns, while still developing techniques of drawing in perspective. These techniques later proved to be essential for the identity of the Baroque period, where perspective was applied to construction to create optical illusions. More recently, in the 19th century, a paradigm shift introduced a form-finding approach, a forward process to

find an optimal geometric solution that is statically in equilibrium within a design loading (Adriaenssens, Veenendaal, & Williams, 2014). Therefore, the design exploration with physical models was crucial, with Antoni Gaudi and Frei Otto as main pioneers.

The constant theorization of architecture introduced terms like Moretti's *Architettura Parametrica* way before being applied in practice, a concept that eventually evolved into more practical terms of the discipline with the development of new tools within a new paradigm: computing.

During the 20th century, the progressive work of several individuals made possible the digitalization of the drawing board, from the accomplishments of Ivan Sutherland, such as Sketchpad, to the most recent developers of the sophisticated digital tools used for the generation of representations of architecture.

Therefore, representation methods not only conquered a new digital dimension, that enables architects to produce accurate documentation with greater efficiency, but they also accomplished a new way of thinking with the introduction and development of algorithmic design.

Algorithmic Design

From the early 2000s that AD has influenced the way some architects work, which inclusively led to the conclusion that architecture was dealing with a paradigm shift (Terzidis, 2004) regarding a new way of thinking through algorithmic and parametric approaches.

However, the main controversy revolved around the ethics of the design process: the value of the fluidity of a hand-drawn idea entirely conceived in the architect's mind against the uncreative and insensitive automatism of designs generated by computers. Irrespective of all the opportunities guaranteed by the algorithmic process and thinking, the creative solutions generated through computational methods are still the object of great criticism (Castellano, 2011). Nevertheless, a strong belief seeks to counteract this reasoning by believing that computation is not a substitute for human creativity but rather the means to explore, experiment, and invest in an alternative realm (Terzidis, 2004). Therefore, the architect ultimately still leans on his critical sense, sensibility, and creativity to build his idea and to make decisions.

The power of applying computer science methodologies to architecture is in conquering not only new tools, that allow a more efficient and rigorous workflow, but also extending the creative reasoning with algorithmic thinking. In other words, architects become able to potentiate their imagination based on what is now possible to accomplish in terms of computation and assessment of geometry.

In addition to the controversy surrounding the role of programming in architecture, other AD-related questions focus more on the implementation of this methodology for the discipline: if architects should learn to program, what programming language should they use?

3. PROGRAMMING LANGUAGES

Programming languages (PLs) are used to translate the intentions of the user into instructions that the computer can understand and process, being a formal medium to express ideas and not solely a means to get a machine to perform operations (Abelson & Sussman, 1996).

Therefore, the algorithmic approach lays on the use of these PLs to define rules and constraints within an algorithm that, when executed, generates a digital model of an idea. PLs for architecture can either be visual or textual.

Visual Programming Languages

With visual programming languages (VPLs) the user interacts with the programming environment to create an algorithm through drag-and-drop and connection of boxes that represent programming abstractions, such as geometry, parameters, or functions (Ferreira & Leitão, 2015) (figure 5).

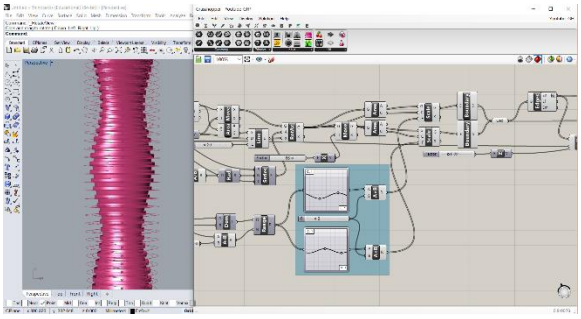

Fig. 5. Programming environment of a VPL.

VPLs provide a set of visual features and mechanisms that make programming more intuitive and appealing for less experienced users. Some of those features and mechanisms are: (1) number sliders and Boolean toggles, to quickly change a numeric or Boolean value, (2) immediate feedback, to visualize in the modeling tool and in real time the changes being made to the program, (3) traceability, to highlight in the modeling tool the geometry that is being generated by a selected part of the program, (4) visual input mechanisms, to import geometry directly from the modeling tool as an input to the program, among others. There are two main VPLs that are currently used by architects: Grasshopper and Dynamo.

Grasshopper is deeply integrated within a CAD tool, Rhino, while Dynamo was created to be integrated in Revit, a BIM tool. Both Grasshopper and Rhino require no deep knowledge of programming or scripting, still allowing designers to build from simple geometry to the awe-inspiring (Davidson, 2018). Nonetheless, they both provide specific abstractions that allow extending the features of the available factory-set abstractions (Ferreira & Leitão, 2015), by introducing written code, using TPLs such as Python. Dynamo, besides all the modeling features, further includes abstractions that import the families of objects available in Revit.

Textual Programming Languages

In textual programming languages (TPLs), the programming abstractions represented visually in VPLs as boxes, are defined by written code (figure 6). Programs in TPLs are, therefore, linear sequences of characters (Leitão, Santos, & Lopes, 2012) structuring a script of written instructions.

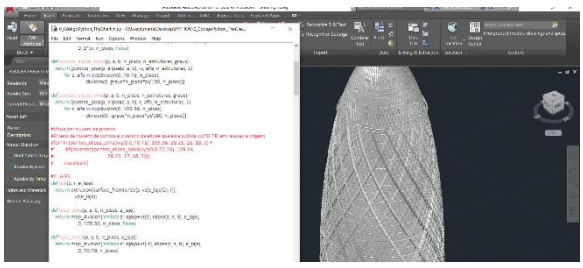

Fig. 6. Programming environment of a TPL.

TPLs firstly found their application in architecture through modeling tools: languages such as RhinoPython were created to extend the functionalities of Rhino, allowing the automation of repetitive processes. However, this first attempt did not succeed among architects, since neither the languages provided nor the programming environments they used were sufficiently intuitive for architects.

More recently, textual programming environments such as Rosetta (Lopes & Leitão, 2011) and Khepri were developed to surpass this limitation.

Rosetta and Khepri

Rosetta is a programming tool that allows the use of different PLs (front-ends) to create programs that interoperate with different back-ends, such as CAD tools (Lopes & Leitão, 2011) and BIM tools (Feist et al., 2016). This means that, by using one of the TPLs supported by Rosetta, such as Python, one can create a program that can be generated and visualized in one or more of the integrated modeling tools.

Khepri is also a textual programming tool descendent of Rosetta, also allowing the generation of geometric data in both CAD and BIM tools. Furthermore, the tool is also expanding for the analysis realm, as it is being developed with new implementations for different analysis tools.

Khepri is an on-going project that is being developed to implement new methodologies to bring the tool closer to architects. The main difference between Khepri and its ancestor is in its front-ends, since it currently solely accepts programs written in Julia, a high-level TPL that provides a very intuitive syntax due to its resemblance to mathematics.

Programming Languages in Architecture

Although there is no compelling evidence regarding the relative advantages of VPLs and TPLs (Menzies, 2002), nowadays, there is an overwhelming tendency for the use of VPLs over the use of TPLs among architects. This reality can be explained by the visual nature of the discipline, that makes architecture gravitate towards more visual ways of programming (Austin & Qattan, 2016).

This propensity can also be associated with the fact that the first TPLs that were made available for AD were not adapted to less experienced users, even though modern TPLs already provide a variety of linguistic features that are designed to surpass these limitations (Leitão, Santos & Lopes, 2012).

Regardless of the programming paradigm, PLs associated with AD undoubtedly bring advantages to the design process and have been incorporated by sounding references in contemporaneous architecture. Architectural studios such as NBBJ, Zaha Hadid Architects, Gehry and Associates, Foster + Partners, and UNStudio have been challenging themselves to build and think increasingly complex projects with bold geometries and optimized performances.

Characteristics of Programming Languages

There are five main concepts that characterize PLs and that relate to their appeal in terms of performance and ease of use in architecture: (1) computational power, (2) performance, (3) expressive power, (4) user interface features, and (5) limit of complexity.

The computational power of a programming language is the measurement of the complexity of the problems that can be described using it (Leitão & Proença, 2014). It has already been proven that most PLs are Turing-complete and, thus, have equivalent computational power, meaning that they all should be able to solve all the problems that they are instructed to solve. Therefore, computational power is not a differentiator, as opposed to other characteristics, such as performance and expressive power.

Performance defines the efficiency with which PLs compute an instruction. The faster the language can process the information given, the better its performance is. Thus, we can also relate the performance level of a PL to its appeal, since it is preferable to choose a faster language to compute a given instruction.

The expressive power of a PL reflects the ability and ease of expressing ideas using that programming language, i.e., it measures the breadth of ideas that can be described in that language (Leitão & Proença, 2014). Hence, it is dependent on the abstraction mechanisms the language provides to define those ideas, also relating to the human effort needed to describe them (Leitão & Proença, 2014).

Another aspect that differentiates PLs is their user interface, i.e., the digital environment used to communicate with the computer. This feature is different in VPLs and TPLs, since VPLs benefit from a graphical user interface – a canvas where the architect freely distributes the icons, while TPLs offer a textual user interface, which can be less appealing and more constrained to a specific structure and syntax.

All these previously mentioned characteristics are related to the fifth criterium mentioned, the limit of complexity, that determines the level of complexity supported by a language while maintaining the provided features fully working.

With the development of this investigation, we aim to test the expressive power and the limit of complexity of current VPLs and TPLs for architecture. This evaluation will be demonstrated along with the increasing complexity of the exercises being solved using both types of languages, in order to better identify the moments where the performance and efficiency of each paradigm start to become compromised.

4. EVALUATION

The methodology adopted firstly focuses on a comparative evaluation of the main characteristics of VPLs and TPLs in their current application for architecture, by developing a study both in theory and in practice.

Theoretical comparison

Regarding theory, VPLs and TPLs are compared according to what is expected of them to offer in terms of features, mechanisms, and performance. To accomplish that, we enumerate the most appealing and useful features and mechanisms, such as abstraction mechanisms, immediate feedback, and traceability, to then describe the extent of their implementation in current PLs for architecture.

This first comparison confirms that the abstractions, features, and mechanisms provided by visual approaches are significantly more appealing for architects.

Regarding TPLs in particular, the scalability of programs becomes one of the strongest advantages, being associated with a high-performance level, a consequence of the intrinsic flexibility of the textual approach. Therefore, the user can build programs that integrate different levels of complexity of the same project, decide which parts are to be generated, and even generate the same information in different modeling and analysis tools.

Despite the visual features provided by TPLs such as Processing (Reas & Fry, 2007), Luna Moth (Alfaiate & Leitão, 2017), and Rosetta (Lopes & Leitão, 2011) their application ended up being restricted to a scientific and academic context.

Still, the awareness of this gap between theory and practice has debunked more focused research and development of more sophisticated and robust tools, such as Khepri. This programming environment also aims to incorporate the possibility of using visual mechanisms as a way that brings textual programming closer to architects (Sammer, Leitão, & Caetano 2019; Sammer & Leitão, 2019).

Practical comparison

The practical comparison assesses the characteristics presented in theory by testing their implementation and application while programming hypothetical case studies.

These case studies are, thus, solved in both a VPL, Grasshopper, and a TPL, Julia within Khepri, with a format that illustrates the correspondent implementations as follows:

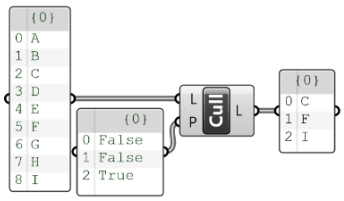

Fig. 7. *Cull Pattern* component in Grasshopper.

```
cull(template, as) =

  [a for (a,t) in zip(as, cycle(template)) if t]
```

In this example, Grasshopper's predefined component *Cull Pattern* (figure 6) was evaluated in terms of its functionality and application to, then, be implemented within Khepri.

In order to elucidate the viability of some characteristics and visual mechanisms provided, different challenges with increasing complexity were used. The analysis of the experiment shows that immediate feedback and traceability are quickly hampered by a greater complexity of the program and can even stop working for more extreme cases.

Another dimension of complexity regards the visual appearance of a program. In VPLs, a program that is visually complex generally compromises its readability. In an attempt to circumvent this problem, VPLs offer mechanisms to organize the program that are still not enough to make programs of greater complexity more readable. In sum, a visually complex program in a VPL is difficult to understand, modify, and maintain.

Contrastingly, complex textual programs do not mean illegible programs, once it is not a larger amount of code that makes the program harder to understand. This is associated with (1) the sophisticated abstraction mechanisms, that enable to build abstract calculation patterns so that they can be treated as simple operations, (2) the uniformity of the formal structure of the language, and (3) all the searching mechanisms available to navigate and organize the program.

The visual complexity of programs can also be overcome by the use of more sophisticated abstractions. Grasshopper already provides components that compute demanding operations or manage large data sets. However, these abstractions are not modifiable within their own implementation, meaning that it is not possible to make alterations to better answer to specific requests. The solution usually comprises the combination of several abstractions, which can increase the visual and computational complexity of the program.

Using the textual approach, these limitations are surpassed by the intrinsic flexibility of the paradigm. Even though it is possible to predefine abstractions within a programming tool, there is an endless combination of instructions and operations that, combined with mechanisms such as recursion or higher-order functions, structure a freer approach to programming new abstractions.

We can then conclude that, even though TPLs require a greater abstraction power from their users, their continuous use will translate in a gradual mastery of the approach.

Regarding visual mechanisms implemented within the textual approach, we conclude that they may require a greater time of computation when compared with VPLs. Unlike Grasshopper, the geometry programmed in Khepri is generated in the modeling tool every time the program is executed, a time-consuming task. Therefore, even though visual mechanisms in TPLs offer clear advantages regarding the interactivity with the modeling tool, the relevance of their

usage and the compromise of performance required should always be considered by the architect.

Another point assessed is related to the ability to use the same program in the context of different modeling and analysis tools, i.e. portability. VPLs are developed and integrated within either a CAD or a BIM tool, even though they provide plug-ins to extend their portability to other tools. However, some of these plug-ins can be difficult to use for less experienced users, regardless the all the documentation usually available.

On the other hand, environments like Khepri were already created and developed to integrate portability. Even though users must create a program that predicts the use of tools with more specific data requests, such as BIM tools, the alterations to the program are minimal, as they solely need to inform Khepri which back-end to connect with.

5. GUIDELINES

After studying the theoretical features of both types of languages and assessing their practical application, we are able to draw guidelines regarding their use and implementation by both (1) establishing a methodology that reflects on the use of either VPLs or TPLs according to the intentions of the architect in applying AD and (2) proposing guidance in the implementation of further features and mechanisms in TPLs that bring them closer to their users from early stages of the design process.

The choice of approach

There are different types of architects, with different beliefs and work methods, as there are different types of projects, varying in context, program, scale, and aesthetics, among others. Each stage of the design process requires different approaches to design. The first creative steps usually include a brainstorm materialized by sketches and drawings. The question that now arises considers the following stages after the conception of an idea: how to materialize and formalize a sketched concept? The proposed methodology identifies which approach should be adopted according to the type of project or documentation that the architect wants to generate.

These guidelines are illustrated by a 3D orthogonal relationship between the approach (manual or AD) and the tools in use (CAD and BIM tools and analysis tools) that establish a relation regarding the complexity of the programs created (figure 8).

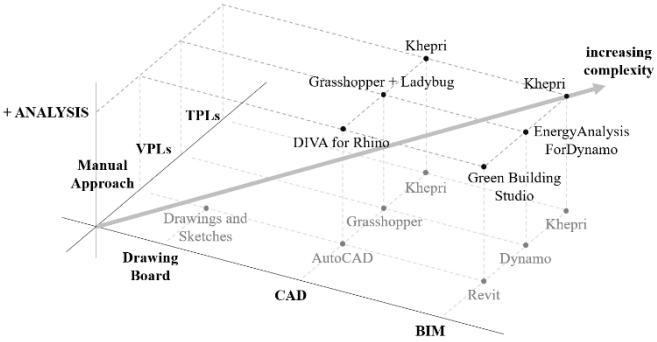

Fig. 8. Relation of the complexity of a project to the approach chosen and the tools used.

The conventional manual approach is worth using when the project solely includes simple geometry and requires a low level of exploration of the design space. After the first drawings and sketches, architects consolidate their ideas in CAD tools, that only allow modeling 2D and 3D geometry, and in BIM tools, that already allow the generation of more complex models to which is associated more sensitive information, such as materials and properties.

However, whenever the project reaches a certain level of complexity, requires a higher level of flexibility and experimentation, or compels to benefit from optimization and analysis, algorithmic approaches should be considered.

The paradigms approached are also related to different scales of complexity. When the requirements to meet solely include a more experimental manipulation of simplified geometry, in order to benefit from visual interaction mechanisms, VPLs offer true benefits. These benefits are usually associated with the use of one single type of modeling tools. In the case of Grasshopper, the geometry is primarily generated in a CAD tool, even though it is possible to extend this generation to more complex BIM models through the use of plug-ins.

VPLs are, therefore, more advantageous to easily and intuitively create simpler and experimental algorithms that solely have to fulfill that purpose – without the need of being developed to generate a full definition of a complex project.

When this complexity increases, VPLs loose some of their advantages, and their programs become hard to understand, change, and maintain. On the other hand, TPLs have no

problem in scaling to large and complex programs, without compromising neither their legibility nor their performance.

Moreover, the textual approach within Khepri further allows the portability of the programs created. This portability allows to also generate simpler geometry in CAD tools and more informative models in BIM tools. Therefore, the complexity of the programs is not only dependent on the programming paradigm in use, but also regards the tools to which the language is connected to.

The third dimension illustrated in the chart (figure 8) relates to analysis tools. While with VPLs this analysis is possible through the use of plug-ins, that might not be so intuitive to use, in programming tools like Khepri this connection is intrinsic due to the portability of the programs. Therefore, the architect solely needs to inform Khepri about the analysis tool to connect with. Analysis tools can, thus, be associated with either the manual or the algorithmic approach and integrated within CAD or BIM tools, always representing an increased complexity of both the work required and the documentation obtained.

The strategy for future implementations

Given the clear long-term advantages of TPLs in the development of increasingly more complex projects, as well as in all variants of analysis, optimization, and visualization of the obtained results, TPLs become an answer to the barriers encountered when using VPLs for complex programs. However, the demanding learning process and the limited availability of interaction and visualization mechanisms, make TPLs still unappealing for architects.

For this reason, this research also proposes guidelines for future developments of Khepri. Some of the guidelines include the implementation of (1) clearer error messages and debugging mechanisms – to aid architects understanding where did an error occur and how can it be fixed, (2) control flow visualization mechanisms – to facilitate the understanding of how the program created is computed, (3) documentation mechanisms and interfaces – to enable a freer and easier learning process by describing the available abstractions and mechanisms, among others.

6. CONCLUSION

The perspectives about what architecture means, what it does, and what it should do vary from region to region of the world, from culture to culture within a region, from person to person from a certain culture. However, there is a common reality to all architects that relates to the representation methods used to illustrate, explain, and sell their intentions for a project.

The evolution of the tools and methodologies used by architects to express their concepts and ideas has always accompanied the progression of humanity's needs in different fields. More recently, the discipline has acknowledged the benefits of computer science and programming to accomplish new esthetics associated with better performances and a more sustainable impact.

AD as a Representation Method for Architecture

With the development of this thesis, we put into perspective algorithmic design (AD) as the most recent representation method for architecture. Besides enabling a more dynamic and efficient workflow, AD applied to architecture already proved to bring further advantages.

Programs created within the AD approach represent an algorithmic description of a design that, then, is generated in computational tools for architecture. This approach enables architects to create parametric models of projects through the establishment of rules and constraints, which, in turn, allow the exploration of design variations more effortlessly and efficiently.

Therefore, AD provides a set of advantages of great relevance for architecture, namely (1) the easy and efficient exploration of the design space of a project; (2) the reduction of the cost of change through the parametrization of the models; (3) the control and minimization of errors while developing and modifying the models; (4) portability, that allows a single program to generate models in various modeling and analysis tools; (5) the direct connection with analysis tools, that prevents the loss of information, misinterpretation, and inaccurate results; (6) a more reliable collaboration process; (7) the ability to assess models with

optimization algorithms in order to obtain optimal results within the design space of a project; among others.

In sum, AD is a good methodology for helping the architect in making more informed and accurate decisions within the design spectrum of his intentions, by incorporating the advantages of the digitalization of the drawing board and extending them to a new realm of optimization, accuracy, and efficiency.

AD as an Architectural Reasoning

Nevertheless, AD has the potential to be more than just a support to the digital representation of a project, as it can further influence other aspects of the design process, namely the architectural reasoning, with the introduction of algorithmic thinking. Therefore, by acknowledging the potentialities associated with the algorithmic generation and assessment of geometry in modeling and analysis tools, architects are able to challenge their design intentions from the beginning of the design conception in order to accomplish new aesthetics and performative results.

Visual or Textual Programming Languages?

The main conclusion regarding this subject is that the answer to this question is relative and dependent upon different factors, such as the type of documentation to be produced, the type of exploration process adopted by the architect, the availability of the architect in working with AD, and the long-term utility of the AD models, among others.

In theory, visual programming languages (VPLs) and textual programming languages (TPLs) have the same computational power, which means that they are both capable of resolving all solvable computational problems. However, with the development of this investigation, we were able to conclude that this theoretical data has a different expression in practice. Other criteria such as performance, expressive power, user-interface features, and limit of complexity make the languages heterogeneous in the relative advantages and disadvantages they offer.

The Barriers to Surpass

As already stated, TPLs are less appealing and intuitive to use. Programs textually structured are more abstract and difficult to understand in the beginning. However, this limitation can easily be overcome, not only by adapting the implementation of TPLs for architects, but also by insisting on their integration within the architectural curriculum.

Furthermore, and besides practical limitations regarding the implementation of TPLs for architecture, there are other psychological obstacles in what concerns the tendency to use VPLs over TPLs.

Architects that are familiar with the visual paradigm within the AD approach tend to accommodate to what it provides, which sometimes means that they also accommodate to the limitations of the languages they are used to work with. This is corroborated by Paul Graham (2003): programmers tend to get attached to their favorite languages, i.e., the languages that are more familiar and comfortable to them.

We conclude that this phenomenon also applies to architects using AD and, once again, this limitation can only be surpassed in the core of the integration of AD for architecture: architects should first learn to master algorithmic thinking in order to be comfortable in creating textual AD programs of their designs. It is only through the insistence that textual approaches will be able to become generalized in the world of architecture.

Furthermore, the fact that TPLs are more demanding to learn and use is related to the knowledge of programming required that, in VPLs, is not necessarily mandatory. Therefore, an architect who learns the fundamentals of programming when working with a TPL, quickly and easily transitions and learns to work with a VPL. The opposite, however, is no longer the case, since the lack of in-depth knowledge of the principles of programming makes it difficult for an exclusive VPL user to write a program using a TPL.

Future Work

We suggest four different lines of investigation to give continuity to the work developed in this thesis: (1) the continuation of the comparison of other visual components of Grasshopper and the equivalent textual abstractions in Khepri, predefined or not; (2) the implementation of more predefined abstractions, features, and mechanisms that enable textual programming tools like Khepri to become more appealing and intuitive for architects; (3) the extension

of this investigation to the BIM and analysis realms by comparing not only VPLs and TPLs, but also VPLs within the different paradigms, namely Grasshopper using BIM and analysis plug-ins and Dynamo for Revit; and (4) the integration of the algorithmic methodology within the conceptual phase of the development of an architectural project, by assessing and comparing the two methodologies: one team of architects using the conventional approach, manipulating manually the various computational tools available for architecture, such as CAD, BIM, and analysis tools, and another team working with an algorithmic approach from the beginning of the conceptual phase, with the integration of all the available tools in a single program. This last line of investigation would be of particular relevance to provide useful clarification regarding (1) the compared efficiency of both work methods, (2) the physical effort and time needed in the development of the proposals, (3) the efficiency of the collaboration process within the team, (4) the time needed to generate digital models and other supports to the communication of the project, (5) the variety of documentation produced within the same time-frame, (6) the quality and accuracy of the results obtained, (7) the performance of the final idea, (8) the interest and relevance of the result obtained according to the objectives to fulfill with the project proposal, among others.

REFERENCES

Abelson, H., Sussman, G. J., & Sussman, J. (1996). The Structure and Interpretation of Computer Programs. Cambridge: MIT Press.

Adriaenssens, S. (Ed.), Block, P. (Ed.), Veenendaal, D. (Ed.), & Williams, C. (Ed.) (2014). Shell Structures for Architecture. London: Routledge.

Alfaiate, P. & Leitão, A. (2017). Luna Moth: A Web-based Programming Environment for Generative Design. Sharing Computational Knowledge! Proceedings of the 35th eCAADe Conference, 2, Rome: Sapienza University of Rome, pp. 511-518.

Austin, M. & Qattan, W. (2016). 'I'm a Visual Thinker.' Rethinking algorithmic education for architectural design. Living Systems and Micro-Utopias: Towards Continuous Designing, Proceedings of the 21st CAADRIA Conference, Melbourne, Australia, pp. 829–838.

Castellano, D. (2011). Humanizing Parametricism. Parametricism (SPC), ACADIA Regional 2011 Conference Proceedings, Lincoln, Nebraska, USA, pp. 275-279.

Davidson, S. (2018). Grasshopper: Algorithmic Modeling for Rhino. [Online]. Retrieved from: http://www.grasshopper3d.com/ [Accessed: Mai. 14, 2018].

Feist, S., Barreto, G., Ferreira, B., & Leitão, A. (2016). Portable Generative Design for Building Information Modelling. Living Systems and Micro-Utopias: Towards Continuous Designing, Proceedings of the 21st CAADRIA Conference, Melbourne, Australia, pp. 147–156.

Ferreira, B., & Leitão, A. (2015). Generative Design for Building Information Modeling. Real Time, Proceedings of the 33rd eCAADe Conference, 1, Vienna: Vienna University of Technology, pp. 635–644.

Graham, P. (2003, Apr.). Beating the Averages. [Online]. Retrieved from: http://paulgraham.com/avg.html [Accessed: Mar. 22, 2019]

Leitão, A. & Proença, S. (2014). On the Expressive Power of Programming Languages for Generative Design: The Case of Higher-Order Functions. Fusion, Proceedings of the 32nd eCAADe Conference, Newcastle upon Tyne: Faculty of Engineering and Environment, pp. 257-266.

Leitão, A., Santos, L., & Lopes, J. (2012). Programming Languages for Generative Design: A Comparative Study. International Journal of Architectural Computing, 10(1), pp. 139–162.

Lopes, J., & Leitão, A. (2011). Portable Generative Design for CAD Applications. ACADIA 2011: Integration Through Computation, Proceedings of the 31st ACADIA Conference, Calgary/Banff, Canada, pp. 196–203.

Menzies, T. (2002). Evaluation Issues for Visual Programming Languages. Chang, S. (Ed.), Handbook of Software Engineering and Knowledge Engineering, 2, Emerging Technologies, London: World Scientific Publishing Co. Pte. Ltd, pp. 93-101.

Reas, C. and Fry, B. (2007). Processing: a programming handbook for visual designers and artists, The MIT Press, Cambridge, Massachusetts & London, England.

Sammer, M., Leitão, A., Caetano, I. (2019). From Visual Input to Visual Output in Textual Programming. Intelligent & Informed: Proceedings of the 24th CAADRIA Conference, Wellington, New Zealand, pp. 645-654.

Sammer, M., & Leitão, A. (2019). Visual Input Mechanisms in Textual Programming for Architecture. Architecture in the Age of the 4th Revolution: Proceedings of the 37th International Conference on Education and research in Computer Aided Architectural Design in Europe (eCAADe), Porto, Portugal, to appear.

Terzidis, K. (2004). Algorithmic Design: A Paradigm Shift in Architecture? Architecture in the Network Society, Proceedings of the 22nd eCAADe Conference, Copenhagen, Denmark, pp. 201-207.