Algorithmic Representation Space

Renata Castelo-Branco, Inês Caetano, and António Leitão
INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal;
renata.castelo.branco@tecnico.ulisboa.pt
ines.caetano@tecnico.ulisboa.pt
antonio.menezes.leitao@tecnico.ulisboa.pt

Abstract

Architecture is an ancient profession, and the means used to produce architectural entities have constantly changed to respond to new design trends and representation needs. Although for centuries this meant gradual changes in the design practice, the increasing technological development witnessed since the 60s has propelled the appearance of increasingly powerful representation methods. One of them – Algorithmic Design (AD) – is based on algorithms, allowing for great design freedom. However, it relies on an abstract representation of design intentions, which hinders their immediate comprehension. To appeal to a broader architectural community, we propose the Algorithmic Representation Space (ARS), a new approach to the way architects represent algorithmic descriptions. The ARS intends to lower AD's comprehension barriers and simultaneously merge it with more traditional representation means, by encompassing not only the algorithm but also its outputs, along with mechanisms that aid the comprehension of the design space it represents.

1. Introduction

Architecture has always explored the latest technological advances, causing changes in the way architects represent and conceive design solutions. Over the past decades, these changes were mostly due to, first, the integration of new digital design tools, such as Computer-Aided Design (CAD) and Building Information Modelling (BIM), which allowed the automation of paper-based design processes [1], and then, the adoption of computational design approaches, such as Algorithmic Design (AD), causing a more accentuated paradigm shift within the architectural practice.

AD is a design approach based on algorithms that has been gaining prominence in both architectural practice and theory [2,3] due to its greater design freedom and ability to automate repetitive design tasks, while facilitating design changes and the search for improved solutions. Its multiple advantages have therefore motivated a new generation of architects to increasingly adopt the programming environments behind their typical modelling tools, going "beyond the mouse, transcending the factory-set limitations of current 3D software" [3; p. 203]. Unfortunately, its algorithmic nature makes this approach highly abstract, deviating from the visual nature of human thinking, which is more attracted to graphical and concrete representations than to alphanumerical ones.

To approximate AD to the means of representation architects typically use and thereby make the most of its added value for the practice, we need to lower the existing comprehension barriers, which hinder its widespread adoption in the field. To that end, this research proposes a new approach to the

representation of AD descriptions – the Algorithmic Representation Space (ARS) – that encompasses, in addition to the algorithm, its concrete outputs and the mechanisms that contribute to its understanding.

2. Algorithmic Representation Method

Despite the cutting-edge aura surrounding it, AD is a natural consequence of architects' desire to automate modelling tasks. In this approach, the architect develops algorithms whose execution creates the digital design model [4] instead of manually modelling it using a digital design tool. Compared to traditional digital modelling processes, AD is advantageous in terms of precision, flexibility, automation, and ease of change, allowing architects to explore wider design spaces easily and quickly. Two AD paradigms currently predominate, the main difference between them lying in the way algorithms are represented: architects develop their algorithms either textually, according to the rules of a programming language, or visually, by selecting and connecting graphical entities in the form of graphs [5]. In either case, the abstract nature of the medium hinders its comprehension.

2.1. Algorithmic Design Paradigms

Algorithms are everywhere and are a fundamental part of current technology. In fact, digital design tools have long supported AD, integrating programming environments of their own to allow users to automate design tasks and deal with more complex, unconventional design problems. Unfortunately, despite its advantages and potential to overcome traditional design possibilities, AD was slow to gain ground in the field, remaining, after almost sixty years, a niche approach. One of the main reasons is the fact that it requires architects to learn programming, which is an abstract task that is far from trivial. This is aggravated by the fact that, for decades, most tools have had their own programming language, which in most cases was limited and hard to use, as well as a programming environment providing little support for the development and comprehension of algorithmic descriptions. Examples include ArchiCAD's GDL (1983); AutoCAD's AutoLisp (1986) and Visual Lisp (2000); 3D Studio Max's MAXscript (1997); and Rhinoceros 3D's Rhino.Python (2011) and RhinoScript (2007).

To make AD more appealing to architects and approximate it to the visual nature of architectural design processes, visual-based AD environments have been released in the meantime. In these environments, text-based algorithmic descriptions are replaced by iconic elements that can be connected to each other in dataflow graphs [6]. Generative Components (2003) is a pioneering example that inspired more recent ones such as Grasshopper (2007) and Dynamo (2011). These tools offer a database of pre-defined operations (components) that users can access by simply dragging an icon onto the canvas and providing it with input parameters. For standard tasks covered by existing components, this speeds up the modelling task considerably. Furthermore, since programs are represented by graph structures – with nodes describing the functions, and the wires connecting them describing the data that gets transferred between them – it is easy to see which parts of the algorithm are dependent upon others, and thus, where the changes are propagated to. However, this is only true for small algorithms, which are a rare find in visual-AD descriptions [7]. Therefore, despite solving part of the existing problems – which explains the growing popularity of this paradigm in the community – others have emerged, such as its inability to deal with more complex and larger-scale AD solutions [5,8,9].

In sum, AD remains challenging for most architects and a far cry from the representation methods they typically use. Human comprehension relies on concrete instances to create mental models of complex concepts [10]. Contrastingly, AD, either visual or textual, operates at a highly abstract level. This grants it its flexibility but also hinders its comprehension.

2.2. Algorithmic Abstractness Vs Model Concreteness

Abstraction can be regarded as the process of removing detail from a representation and keeping only the relevant features [11]. Some authors believe abstraction improves productivity: it not only focuses on the "big idea" or problem to solve [12] but also triggers creative thinking due to its vagueness, ambiguity, and lack of clarity [13].

Abstraction in architecture can be traced back at least as far as classical antiquity. Architectural treatises, such as Vitruvius' "Ten Books on Architecture" [14], are prime examples of abstract representations because they intend to convey not specific design instances, but rather design norms that are applicable to many design scenarios. However, the human brain is naturally more attracted to graphical explanations than textual ones [15–17], a tendency that is further accentuated in a field with a highly visual culture such as architecture. For that reason, even the referred treatises were eventually illustrated after the birth of the printing press [18].

The algorithmic nature of AD motivates designers to represent their ideas in an abstract manner, focusing on the concept and its formal definition. This sort of representation provides great flexibility to the design process, as a single expression of an idea can encompass a wide range of instances that match that idea, i.e., a design space. Contrariwise, most representation methods, including CAD and BIM, compel designers to rapidly narrow down their intentions towards one concrete instance, on account of the labour required to maintain separate representations for each viable alternative.

In sum, abstraction gives AD flexibility and the ability to solve complex problems, but it also makes it harder to understand. Abstraction is especially relevant when dealing with mathematical concepts, such as recursion or parametric shapes; nature-inspired processes, such as randomness; and performance-based design principles, such as design optimisation. It is also critical when developing and fabricating unconventional design solutions, whose geometric complexity requires a design method with a higher level of flexibility and accuracy. Sadly, these are also the hardest concepts to grasp without concrete instances and visual aid.

Nevertheless, the described comprehension barrier, apparently imposed by the abstract-concrete dichotomy, is more obvious when the AD descriptions are independent entities with little to no connection to the outcomes they produce. Figure 1 represents the current conception of AD: there is a parametric algorithm, representing a design space, which can generate a series of design models when specific parameters are provided. We propose to overthrow this notion by including the outcomes of the algorithm in the design process itself, changing the traditional flow of design creation to accommodate more design workflows and comprehension approaches.

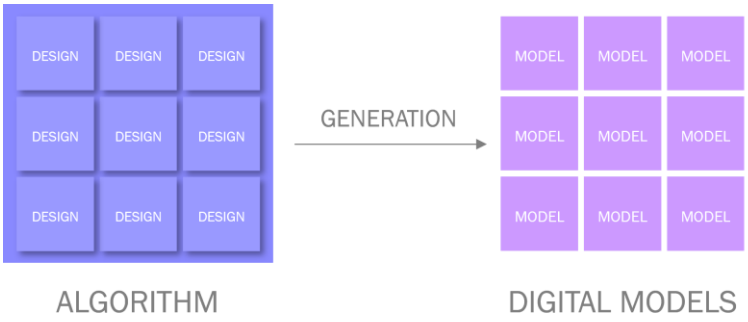

Figure 1: AD workflow - an algorithm, representing a design space, generates a digital model for each design instance.

3. Algorithmic Representation Space

AD descriptions have an abstract nature, which is part of the reason they prove so beneficial to the architectural design process. However, when it comes to comprehending an AD – i.e., creating a mental model of the design space it represents – this feature becomes a burden. Human cognition seems to rely heavily on the accumulation of concrete examples to form a more abstract picture [10]. For this reason, we advocate that, for a better comprehension of an AD, the algorithms themselves do not suffice.

This research proposes a new way to represent algorithmic descriptions that aids the development and understanding of AD projects. Under the name of Algorithmic Representation Space (ARS), this concept encompasses not only the algorithm but also its outcomes and the mechanisms that allow for the understanding of the design space it represents. AD descriptions stand to benefit significantly from the concreteness of the outputs they generate, i.e., the digital models. If we consider the models as part of the AD representation, we reduce its level of abstraction and increase its understandability, approximating it to the visual nature of human understanding. Nevertheless, we must also smooth its integration in more traditional design workflows, helping architects who still develop their models manually in digital design tools or are forced to use pre-existing models. Accordingly, the proposed ARS also enables the use of already existing digital models as starting points to arrive at an algorithmic description.

There are two core elements in the ARS (Figure 2), the **algorithm** and the **model**. The algorithm represents a design space in a parametric abstract way, which makes the multiple design alternatives it represents difficult to perceive. Contrastingly, each model represents an instance of a design space in a static but concrete way. Combining the former's flexibility with the latter's perceptibility is therefore critical for the success of algorithmic representation. For conceptual reasons, the presented illustration of the ARS treats the two elements as equal. Nevertheless, one must keep in mind that the algorithm can generate potentially infinite digital models, and the concept holds for all of them.

We consider two entry points into the ARS: *programming* and *modelling*. Each will allow architects to traverse the ARS; in the former case, from algorithm to model, by *running* the instructions in the algorithm to generate a model; and in the latter, from model to algorithm, by *extracting* an algorithmic description capable of generating the design instance and then *refactoring* that description to make it parametric as well. In either case, it is important the ARS contemplates the visualisation of these

algorithm-model relationships. Therefore, we propose including techniques such as *traceability* in any ARS. In the following section, we will use a case study, the Reggio Emilia Train Station by Santiago Calatrava, to illustrate the ARS and each of the proposed principles.

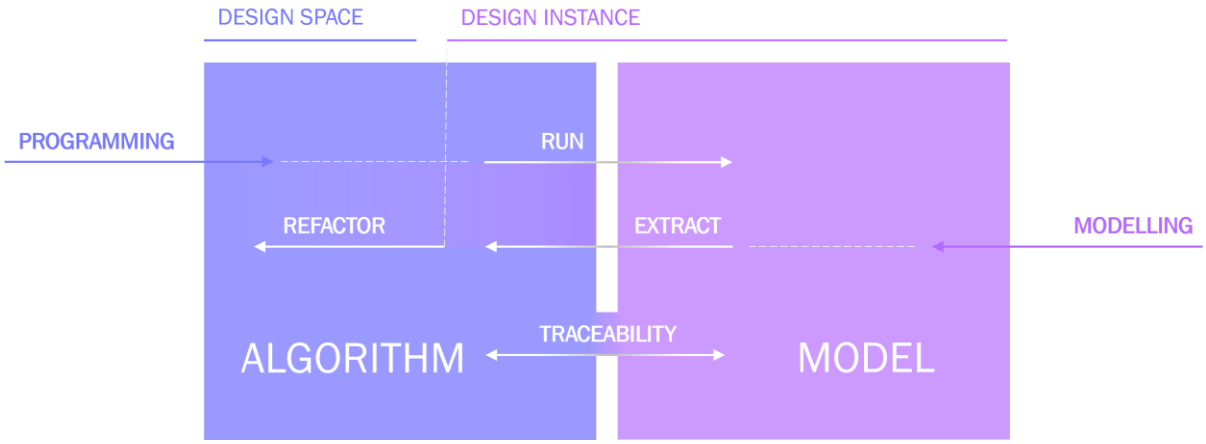

Figure 2: Building blocks of the ARS.

3.1. Programming

The typical AD process entails the creation of a parametric description that abstractly defines a design space according to the boundaries set by the architect (Figure 3). The parametricity of this description, or the size of the design space it represents, varies greatly with the design intent and the way it is implemented (e.g., degrees of freedom, rules, and constraints). By *instantiating* the parameters in the algorithm, the architect specifies instances of the design space, whose visualisation can be achieved by generating them in a digital design tool, such as a CAD, BIM, or game engine (Figure 3 - running the algorithm). Figure 4 presents several variations of the Reggio Emilia station achieved by running the corresponding AD description with varying input parameters, namely with a different number of beams, different beam sizes, and different amplitudes and phases of the sinusoidal movement.

Given the flexibility of this approach, the process of developing AD descriptions tends to be a very dynamic one, with the architect repeatedly generating instances of the design to assess the impact of the changes made at each stage. Consciously or not, architects already work in a bidirectional iterative way when using AD. However, this workflow can also greatly benefit from a more obvious showcasing of the existing relations between algorithm and model. Traceability mechanisms allow precisely for the visual disclosure of these relations (i.e., which instruction/component generated which geometry), and several AD tools support them already.

3.2. Creating Models

AD is not meant to replace other design approaches but, instead, to interoperate with them. This interoperability is important, to take advantage of the investment made into those well-established representation methods such as CAD and BIM, especially for projects where digital models already exist

or are still being produced. Therefore, the second entry point to the ARS is the conversion of an existing digital model of a design into an AD program. This might be necessary, for instance, when we wish to optimise it for new uses and/or to comply with new standards [19]. This process entails crossing the ARS in the opposite direction to that described in the previous section (Figure 5).

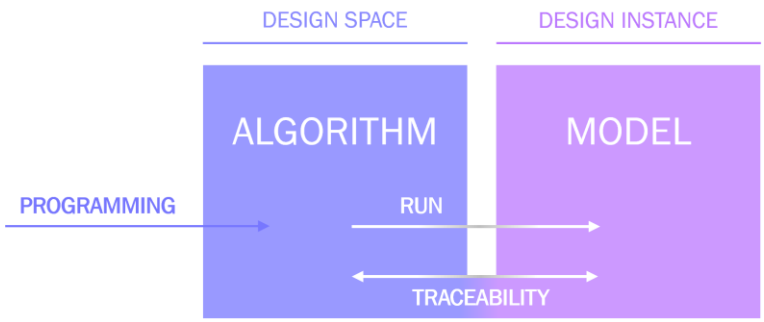

Figure 3: Entering the ARS by programming.

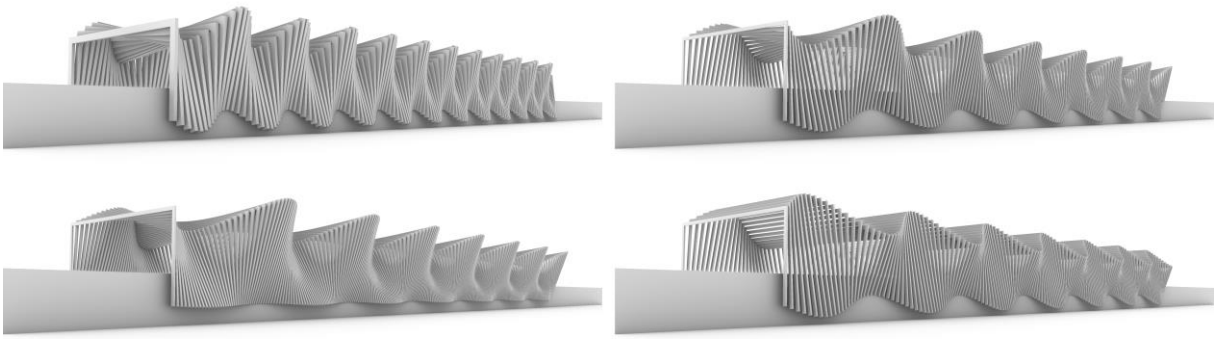

Figure 4: Parametric variations of the Reggio Emilia station, with different numbers and sizes of beams, and different amplitudes and signs of the sinusoidal movement.

To convert a digital model into an AD description, there are two main steps: extraction and refactoring. Extraction entails the automatic generation of instructions that can reproduce an exact copy of the model being extracted. The resulting AD description, however, is non-parametric and of difficult comprehension. This is where refactoring comes in [20,21], a technique that helps to improve the AD description, increasing its readability and parametricity. While the first task can be almost entirely automated, and is currently partially supported by some AD tools, the second part depends heavily on the architect's design intent and, thus, will always be a joint effort between man and machine. In either case, it is important that the ARS adapts to the multiplicity of digital design tools and representation systems that architects often use during their design process. They can use, for instance, 3D modelling tools, such as CADs or game engines, to geometrically explore their designs more freely, or BIM tools to enrich the designs with construction information and to produce technical documentation.

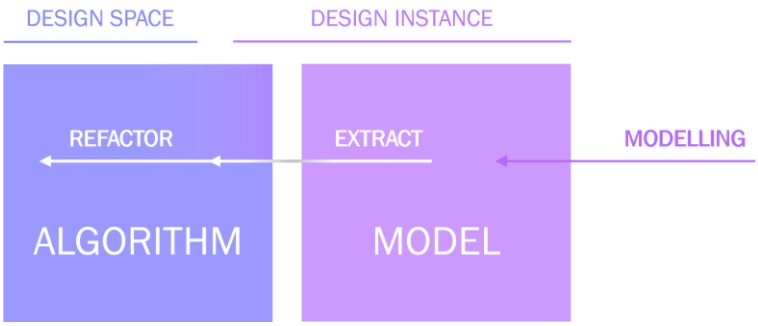

Figure 5: Entering the ARS through modelling.

4. Navigating the ARS

As mentioned in the previous section, there are two main elements in the ARS: algorithms abstractly describing design spaces and digital models representing concrete instances of those design spaces. Either one can be accessed from either end of the spectrum, i.e., by programming and running the algorithm to generate digital models, or by manually modelling designs and then converting them into an algorithm. To allow for this bidirectionality between the two sides, the ARS relies on three main mechanisms: (a) traceability, (b) extraction, and (c) refactoring. The first allows the system to expose the existing relationships between algorithm and model in a visual and interactive way for a better comprehension of the design intent. The latter two allow us to traverse the ARS from model to algorithm, a less common crossing but an essential one, nevertheless. The following sections describe these three mechanisms in detail.

4.1. Traceability

For a proper comprehension of ADs, architects must construct a mental model of the design space, comprehending the impact each part of the algorithm has in each instance of the design space. To that end, a correlation must be ever present between the two core elements of the ARS – algorithm and model – matching the abstract representation with its concrete realisation. Traceability establishes relationships amongst the instructions that compose the algorithm and the corresponding geometries in the digital model. This is particularly relevant when dealing with complex designs, as it allows architects to understand which parts of the algorithm are responsible for generating which parts of the model.

With traceability, users can select parts of the algorithm or parts of the model and see the corresponding parts highlighted in the other end. Grasshopper for Rhinoceros 3D and Dynamo for Revit, two visual AD tools, present unidirectional traceability mechanisms from the algorithm to the model. Figure 6 shows this feature at play in Grasshopper: users select any component on the canvas and the corresponding geometry is highlighted in the visualised model.

Regarding bidirectional traceability, there are already visual AD tools that support it, such as Dassault Systèmes' xGenerative Design tool (xGen) for Catia and Bentley's Generative Components, as well as textual AD tools, such as Rosetta [22], Luna Moth [23], and Khepri [24]. Figure 7 shows the example of

Khepri, where the user selects either instructions in the algorithm or objects in the model and the corresponding part is highlighted in the model or algorithm, respectively. Programming In the Model (PIM) [25], a hybrid programming tool, offers traceability between the three existing interactive windows: one showing the model, another the visual AD description, and a third showing the equivalent textual AD description.

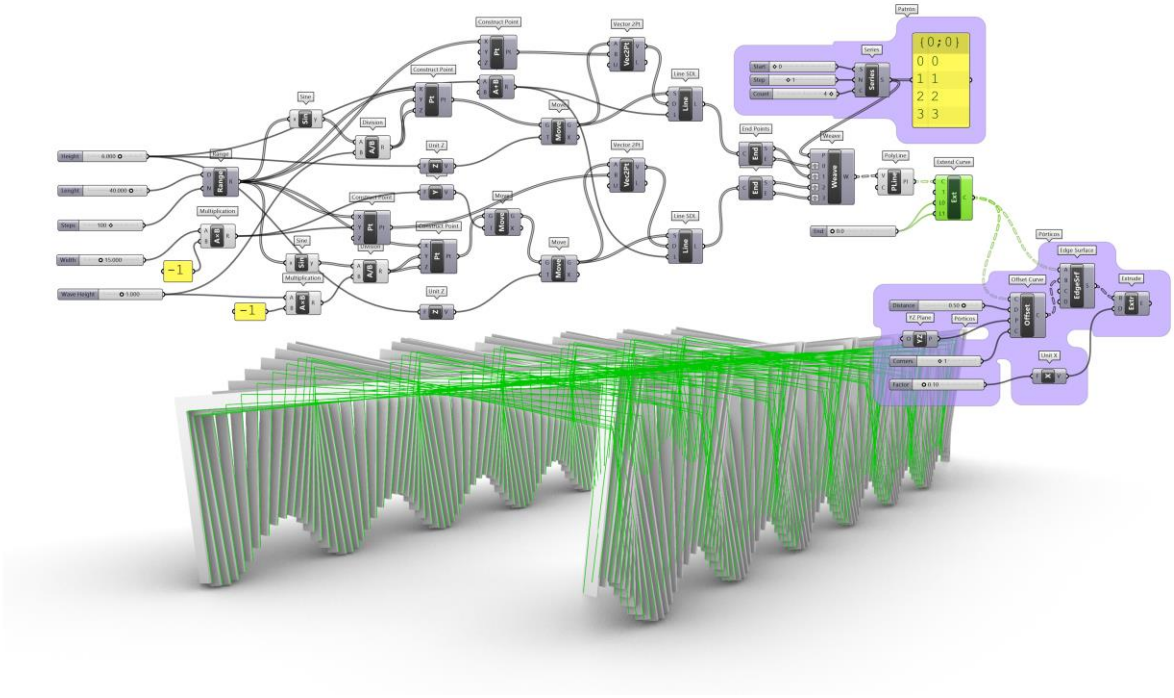

Figure 6: Traceability in visual AD tools – the case of Grasshopper.

Unfortunately, traceability is a computationally intensive feature that hinders the tools' performance with complex AD programs – especially model-to-algorithm traceability, which explains why some commercial visual-based AD tools avoid it. Those that provide it inevitably experience a decrease in performance as the model grows. All referred text-based and hybrid options are academic works, built and maintained as proof of concept and not as commercial tools, which explains their acceptance of the imposed trade-offs. A possible solution for this problem is to allow architects to decide when to use this feature and only switch it on when the support provided compensates for the computational overhead [26]. In fact, traceability-on-demand is Khepri's current approach to the problem.

4.2. Extraction

Extraction is the automatic conversion of a digital model into an algorithm that can faithfully replicate it. Previous studies [27,28] focused on the generation of 3D models from architectural plans or on the conversion of CAD to BIM models, using heuristics and manipulation of geometric relations. Sadly, the

result is not an AD description, but rather another model, albeit more complex and/or informed. One promising line of research is the use of the probabilistic and neural-based machine learning techniques (e.g., convolutional or recurrent neural networks) that address translation from images to textual descriptions, [29] but further research is needed to generate algorithmic descriptions.

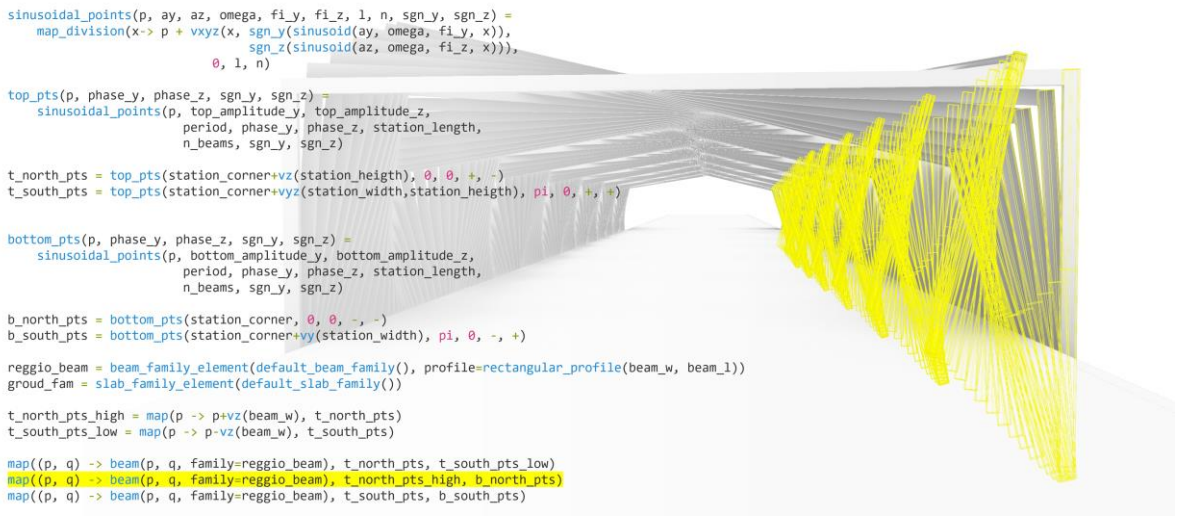

Figure 7: Traceability in textual AD tools – the case of Khepri.

The main problems with extracting a parametric algorithm lie, first, in the assumptions the system would need to make while reading a finished model: for instance, distinguishing whether two adjacent volumes are connected by chance or intentionally and, if the latter, deciding if such connection should constitute a parametric restriction of that model or not. Secondly, it is nearly impossible to devise a system that can consider the myriad of possible geometrical entities and semantics available in architectural modelling tools.

Some modelling tools that favour the VP paradigm avoid this problem by placing the responsibility on the designer from the very start, restricting the modelling workflow and forcing the designer to provide the missing information. In xGen and Generative Components, the 3D model and the visual algorithm are in sync, meaning changes made in either one are reflected in the other. PIM presents a similar approach, extending the conversion to the textual paradigm as well, although it was only tested with simple 2D examples.

In practice, these tools offer real-time conversion from the model to the algorithm. However, either solution requires the model to be parametric from the start. Every modelling operation available in these tools has a pre-set correspondence to a visual component, and designers must build their models following the structured parametric approach imposed by each tool, almost as if they were in fact constructing an algorithm but using a modelling interface. As such, the system is gathering the information it needs to build parametric relations from the very beginning. This explains why neither xGen, nor Generative Components, nor PIM, can take an existing model created in another modelling software or following other modelling rules and extract an algorithmic description from it.

This problem has also been addressed in the TP field and promising results have been achieved in the conversion of bi-dimensional shapes into algorithms [24,30]. However, further work is required to recognise 3D shapes, namely 3D shapes of varying semantics, since architects can use a myriad of digital design tools to produce their models, such as CADs, BIMs, or game engines. Figure 8 presents an ideal scenario, where the ARS is able to extract an algorithm that can generate an identical model to that being extracted.

In either case, even if we arrive at the extraction of the most common 3D elements any time soon, the resulting algorithm will only accurately represent the extracted model, and it will comprise a low-level program, which is very hard for humans to understand. To make the algorithm both understandable and parametric, it needs to be further transformed according to the design intent envisioned by the architect. Increasing the algorithm's comprehension level and the design space it represents is the goal of refactoring.

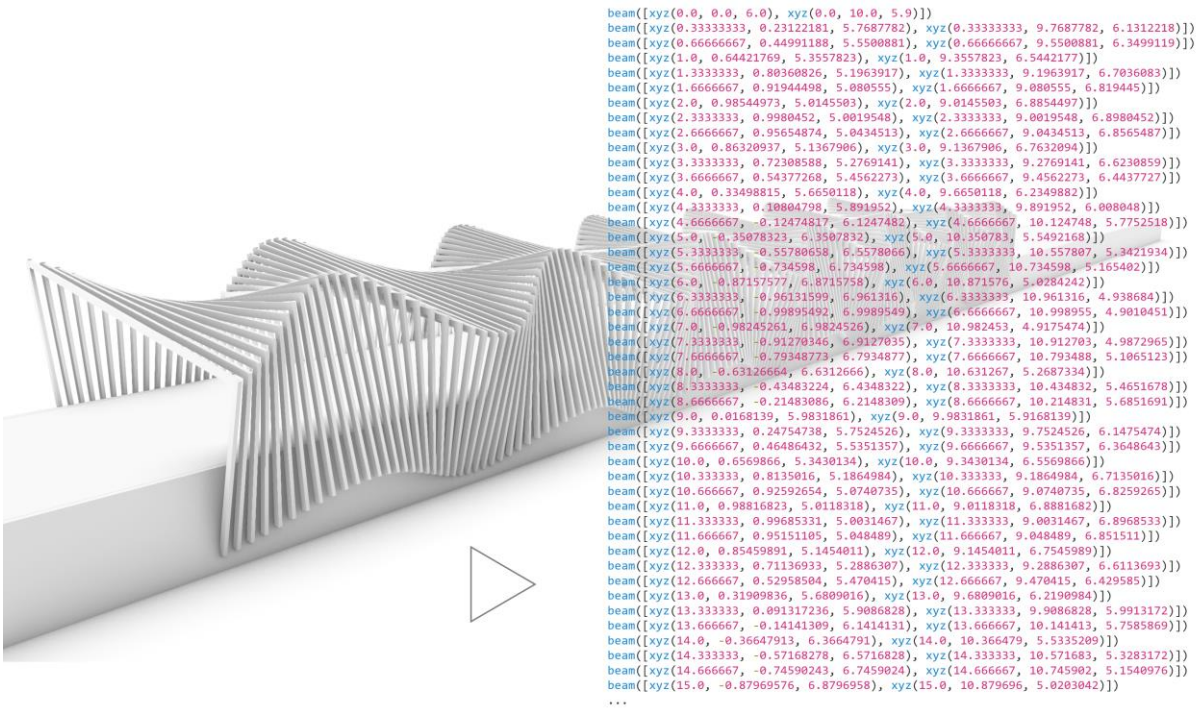

Figure 8: Extraction process – on the left the digital model, and on the right the sequence of instructions resulting from the extraction process.

4.3. Refactoring

Refactoring (or restructuring) is commonly defined as the process of improving the structure of an existing program without changing its semantics or external behaviour [20]. There are already several semi-automatic refactoring tools [21] that help to improve the readability and maintenance of algorithmic descriptions and increase their efficiency and abstraction level. Refactoring is an essential

follow-up to an extraction process, since the latter returns a non-parametric algorithm that is difficult to decipher.

Figure 9 shows an example of a refactoring process that could take place with the algorithm extracted in Figure 8. The extracted algorithm contains numerous instructions, each responsible for generating a beam between two spatial locations defined by XYZ coordinates. It is not difficult to infer the linear variations presented in the first and fourth highlighted columns, which correspond to the points' X values. To infer the sinusoidal variation in the remaining values, however, more complex curve-fitting methods would have to be implemented [31].

In either case, refactoring tools seldom work alone, meaning that a lot of user input is required. This is because there is rarely a single correct way of structuring algorithms, and the user must choose which methods to implement in each case. Refactoring tools, beyond providing suggestions, guarantee that the replacements are made seamlessly and do not change the algorithm's behaviour. When trying to increase parametric potential, even more input is required, since it is the architect who must decide the degrees of freedom shaping the design space.

In our example (Figure 9), the refactored algorithm shown below has a better structure and readability but is still in an infant state of parametricity. As a next stage, we could start by replacing the numerical values proposed by the refactoring tool with variable parameters to allow for more variations of the sinusoidal movement.

5. Discussion and Conclusion

Architecture is an ancient profession, and the means used to produce architectural entities have constantly changed, not only integrating the latest technological developments, but also responding to new design trends and representation needs. Architects have long adopted new techniques to improve the way they represent designs. However, while, for centuries, this caused gradual changes in the architectural design practice, with the more accentuated technological development witnessed since the 60s, these modifications have become more evident. The emergence of personal computers, followed by the massification of Computer-Aided Design (CAD) and Building Information Modelling (BIM) tools, allowed architects to automate their previously paper-based design processes [1], shaping the way they approached design issues [32]. However, these tools did little to change the way designs were represented, only making their production more efficient. It did not take long for this scenario to rapidly evolve with the emergence of more powerful computational design paradigms, such as Algorithmic Design (AD). Despite being more abstract and thus less intuitive, this design representation method is more flexible and empowers architects' creative processes.

Given its advantages for architectural design practice, AD should be a complement to the current means of representation. However, to make AD more appealing for a wider audience and allow architects to make the most of it, we must lower the existing barriers by approximating AD to the visual and concrete nature of architectural thinking. To that end, we proposed the Algorithmic Representation Space (ARS), a representation approach that aims to replace the current one-directional conception of AD (going from algorithms to digital models) with a bidirectional one that additionally allows architects to arrive at algorithms starting from digital models. Furthermore, the ARS encompasses as means of representation not only the algorithmic description but also the digital model that results from it, as well as the mechanisms that aid the comprehension of the design space it represents.

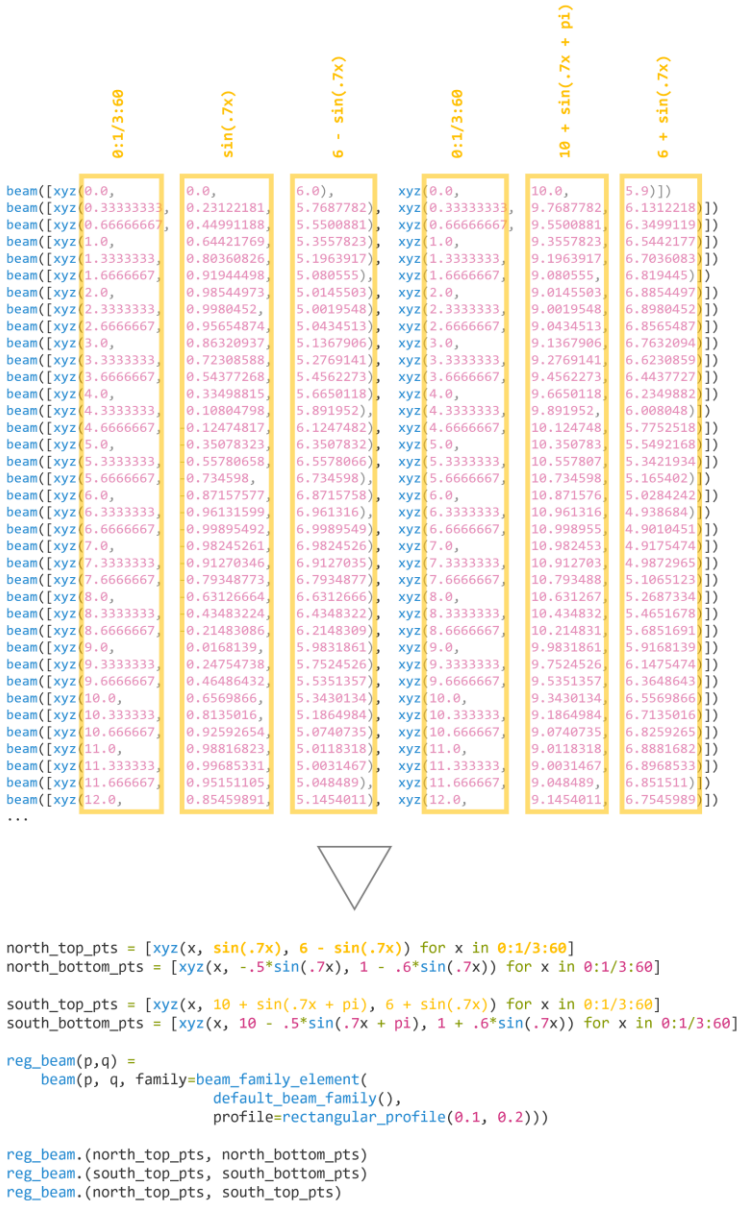

Figure 9: Refactoring process – the sequence of extracted instructions (on top) is converted onto a more comprehensible and parametric algorithm (on the bottom).

The proposed system is based on two fundamental elements – the algorithm and the digital model – and architects have two ways of arriving at them – programming and modelling. Considering the first case, programming, the ARS supports the development of algorithms and the subsequent visualisation of the design instances they represent by running the algorithm with different parameters. In the second case, modelling, the ARS supports the conversion of digital models into algorithms that reproduce them. The first scenario allows AD representations to benefit from the visual nature of digital design tools,

reducing the innate abstraction of algorithms and obtaining concrete instances of the design space that are more perceptible to the human mind. The second case enables the conversion of a concrete representation of a design instance into an abstract representation of a design space, i.e., a parametric description that can generate possible variations of the original design, benefiting from algorithmic flexibility and expressiveness in future design tasks.

To allow for this bidirectionality, the ARS relies on three main mechanisms: (a) traceability, (b) extraction, and (c) refactoring. Traceability addresses the non-visual nature of the first process – programming – by displaying the relationships between the algorithm and the digital model. Extraction and refactoring address the complexity of the second process – going from model to algorithm – the former entailing the extraction of the algorithmic instructions that, when executed, generate the original design solution, and the latter solving the lack of parametricity and perceptibility of the extracted algorithms by helping architects restructure them. The result is a new representation paradigm with enough (1) expressiveness to successfully represent architectural design problems of varying complexities; (2) flexibility to parametrically manipulate the resulting representations; and (3) concreteness to easily and quickly comprehend the design space embraced.

The proposed ARS intends to motivate a more widespread adoption of AD representation methods. However, it is currently only a theoretical outline. To reach its goal, the proposed system must gain a practical character. As future work, we will focus on applying and evaluating the ARS in large-scale design scenarios, while retrieving user feedback from the experience.

6. Acknowledgments

This work was supported by national funds through *Fundação para a Ciência e a Tecnologia* (FCT) (references UIDB/50021/2020, PTDC/ART-DAQ/31061/2017) and PhD grants under contract of FCT (grant numbers SFRH/BD/128628/2017, DFA/BD/4682/2020).

7. References

[1] S. Abubakar and M. Mohammed; Halilu, "Digital Revolution and Architecture: Going Beyond Computer-Aided Architecture (CAD)". In *Proceedings of the Association of Architectural Educators in Nigeria (AARCHES)* Conference (2012)., 1–19.

[2] R. Oxman, "Thinking difference: Theories and models of parametric design thinking". *Design Studies* (2017), 1–36. DOI:http://doi.org/10.1016/j.destud.2017.06.001

[3] K. Terzidis, "Algorithmic Design: A Paradigm Shift in Architecture ?" In *Proceedings of the 22nd Education and research in Computer Aided Architectural Design in Europe (eCAADe)* Conference, Copenhagen, Denmark (2004), 201–207.

[4] I. Caetano, L. Santos, and A. Leitão, "Computational design in architecture: Defining parametric, generative, and algorithmic design." *Frontiers of Architectural Research* 9, 2 (2020), 287–300. DOI:https://doi.org/10.1016/j.foar.2019.12.008

[5] P. Janssen, "Visual Dataflow Modelling: Some thoughts on complexity". In *Proceedings of the 32nd Education and research in Computer Aided Architectural Design in Europe (eCAADe)* Conference,

Newcastle upon Tyne, UK (2014), 305–314

[6] E. Lee and D. Messerschmitt, "Synchronous data flow". *Proceedings of the IEEE* 75, 9 (1987), 1235–1245. DOI:https://doi.org/10.1109/PROC.1987.13876

[7] D. Davis, "Modelled on Software Engineering: Flexible Parametric Models in the Practice of Architecture". PhD Dissertation, RMIT University (2013).

[8] A. Leitão and L. Santos, "Programming Languages for Generative Design: Visual or Textual?" In *Proceedings of the 29th Education and research in Computer Aided Architectural Design in Europe (eCAADe)* Conference, Ljubljana, Slovenia (2011),139–162.

[9] M Zboinska, "Hybrid CAD/E Platform Supporting Exploratory Architectural Design". *CAD Computer Aided Design* 59, (2015), 64–84. DOI:https://doi.org/10.1016/j.cad.2014.08.029

[10] D. Rauch, P. Rein, S. Ramson, J. Lincke, and R. Hirschfeld, "Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code". *The Art, Science, and Engineering of Programming,* 3, 3 (2019), 9:1-9:39. DOI:https://doi.org/10.22152/programming-journal.org/2019/3/9

[11] H. Abelson, G.J. Sussman, and J. Sussman (1st ed. 1985), *Structure and Interpretation of Computer Programs* (Cambridge, Massachusetts, and London, England: MIT Press, 1996) DOI:https://doi.org/10.1109/TASE.2008.40

[12] B. Cantrell and A. Mekies (Eds.), *Codify: Parametric and Computational Design in Landscape Architecture*. (Routledge, 2018). DOI:https://doi.org/10.1017/CBO9781107415324.004

[13] A. Al-Attili and M. Androulaki, "Architectural abstraction and representation". In *Proceedings of the 4th International Conference of the Arab Society for Computer Aided Architectural Design*, Manama (Kingdom of Bahrain) (2009), 305–321.

[14] M. Vitruvius, *The Ten Books on Architecture*. (Cambridge & London, UK: Harvard University Press & Oxford University Press, 1914).

[15] K. Zhang, *Visual languages and applications*. (Springer Science + Business Media, 2007).

[16] N. Shu, 1986, "Visual Programming Languages: A Perspective and a Dimensional Analysis". In *Visual Languages. Management and Information Systems*, SK. Chang, T. Ichikawa and P.A Ligomenides (eds.). (Boston, MA: Springer, 1986). DOI: https://doi.org/10.1007/978-1-4613-1805-7_2

[17] E. Do and M. Gross, "Thinking with Diagrams in Architectural Design". *Artificial Intelligence Review*. 15, 1 (2001), 135–149. DOI:https://doi.org/10.1023/A:1006661524497

[18] M. Carpo, *The Alphabet and the Algorithm*. (Cambridge, Massachusetts: MIT Press, 2011).

[19] I. Caetano, G. Ilunga, C. Belém, R. Aguiar, S. Feist, F. Bastos, and A. Leitão, "Case Studies on the Integration of Algorithmic Design Processes in Traditional Design Workflows". In *Proceedings of the 23rd International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)*, Hong Kong (2018), 129–138.

[20] M. Fowler, *Refactoring: Improving the Design of Existing Code*. (Reading, Massachusetts: Addison-Wesley Longman, 1999)

[21] T. Mens and T. Tourwe, "A survey of software refactoring". *IEEE Transactions on Software Engineering*.

30, 2 (2004), 126–139. DOI:https://doi.org/10.1109/TSE.2004.1265817

[22] A. Leitão, J. Lopes, and L. Santos, "Illustrated Programming". In *Proceedings of the 34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, Los Angeles, California, USA (2014), 291–300.

[23] P. Alfaiate, I. Caetano, and A. Leitão, "Luna Moth Supporting Creativity in the Cloud". In *Proceedings of the 37th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, Cambridge, MA (2017), 72–81.

[24] M. Sammer, A. Leitão, and I. Caetano, "From Visual Input to Visual Output in Textual Programming". In *Proceedings of the 24th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)*, Wellington, New Zealand (2019), 645–654.

[25] M. Maleki and R. Woodbury, "Programming in the Model: A new scripting interface for parametric CAD systems:". *In Proceedings of the Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, Cambridge, Canada (2013), 191–198.

[26] R. Castelo-Branco, A. Leitão, and C. Brás, "Program Comprehension for Live Algorithmic Design in Virtual Reality". In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion)*, ACM, New York, NY, USA, Porto, Portugal, (2020), 69–76. DOI:https://doi.org/10.1145/3397537.3398475

[27] L. Gimenez, J. Hippolyte, S. Robert, F. Suard, and K. Zreik, "Review: Reconstruction of 3D building information models from 2D scanned plans". *Journal of Building Engineering* 2, (2015), 24–35. DOI:https://doi.org/10.1016/j.jobe.2015.04.002

[28] P. Janssen, K. Chen, and A. Mohanty, "Automated Generation of BIM Models". In *Proceedings of the 34th Education and research in Computer Aided Architectural Design in Europe (eCAADe)* Conference, Oulu, Finland, (2016) 583–590.

[29] J. Donahue, L. Hendricks, M. Rohrbach, S. Venugopalan, S. Guadarrama, K. Saenko, and T. Darrell, "Long-Term Recurrent Convolutional Networks for Visual Recognition and Description". *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 39, 4 (2017), 677–691. DOI:https://doi.org/10.1109/TPAMI.2016.2599174

[30] A. Leitão and S. Garcia., "Reverse Algorithmic Design". In *Proceedings of Design Computing and Cognition (DCC'20*) Conference, Georgia, Atlanta, USA (2021). p. 317–328. DOI: https://doi.org/10.1007/978-3-030-90625-2_18

[31] P. Mogensen and A. Riseth, "Optim: A mathematical optimization package for Julia". *Journal of Open Source Software*. 3, 24 (2018), 615. DOI:https://doi.org/10.21105/joss.00615

[32] T. Kotnik, "Digital Architectural Design as Exploration of Computable Functions". *International Journal of Architectural Computing* 8, 1 (2010), 1–16. DOI:https://doi.org/10.1260/1478-0771.8.1.1