Reverse Algorithmic Design

António Leitão[1] and Sara Garcia[2]
[1,2]*INESC-ID,* [1]*Instituto Superior Técnico, Universidade de Lisboa*
antonio.menezes.leitao@tecnico.ulisboa.pt

Algorithmic Design (AD) is an approach that uses algorithms to represent designs. AD allows for a flexible exploration of complex designs, which helps not only the designer but also optimization methods that autonomously search for better-performing solutions. Despite its advantages, AD is still not widely used. This is owed in part to the large amount of time, effort, and expertise required for the development of an AD program, a problem that grows with the complexity of the design. To overcome this issue, this paper proposes Reverse Algorithmic Design (RAD), which infers AD programs from existing CAD or BIM models. RAD comprises two main steps: the automatic generation of an initial low-level AD program from a CAD/BIM model, followed by a semi-autonomous refactoring step that improves the generated program. The benefits of the RAD approach are demonstrated with its application in two use-case scenarios.

Introduction

Nowadays, several digital modeling strategies are available in the design domain, comprising different complexity levels: (1) *geometry-based modeling* allows the manual production of static models in Computer-Aided Design (CAD) tools; (2) *semantic-based modeling* allows the generation of geometric models with added meaning in Building Information Modeling (BIM) tools; (3) *parametric modeling* supports a dynamic representation of designs using parameters, obtaining different static models by assigning different values to the parameters; and (4) *Procedural Modeling* (PM) comprises the generation of models from a procedure or a program. In general, PM has more flexibility to generate design variations than *geome-*

Design Computing and Cognition DCC'20. J.S. Gero (ed),
pp. 323-334.

try- or *semantic-based modeling*, making it much easier to optimize designs or simply adjust them to changing requirements. Moreover, besides supporting parametric changes as happens with *parametric modeling,* PM also allows the application of topological changes.

Despite its advantages, PM is hard to learn and use and, thus, the development of a program from scratch is time- and cost-consuming [1]. To mitigate this problem, one possible approach is to use Inverse Procedural Modeling (IPM), which is the automatic inference of a reusable procedural model from an existing model or set of models [2].

Most research on IPM addresses the inference of grammar-based generative systems, such as parametric L-systems [1], probabilistic grammars [3], shape grammars [4–6] and set grammars [2]. When compared to human-generated grammars, automatic-generated grammars require less time and programming skills to develop, are not influenced by the developers' biases, and can be more efficient [2, 4]. However, although grammar inference has been pointed out as one of the main challenges in the field [7], little progress has been made so far [3]. This may be explained by the fact that (a) it is a computationally expensive process, and (b) there is no precise way to measure a grammar's quality [5]. Stochastic methods were used to solve the first problem [3, 4], and the search for the smallest grammar (which could then be generalized) was used to address the second problem [2, 3, 5]. Despite the scientific usefulness of these approaches, their applicability to real-world problems is limited, as they were only tested with simple designs composed of repeating elements [2, 3, 5, 6].

In this paper, we explore the inference of a different PM approach, namely Algorithmic Design (AD). In an AD approach, digital models are created from algorithmic descriptions [8], allowing the automation of repetitive tasks, facilitating the exploration of design variations, and, most importantly, promoting design optimization. This, therefore, results in significant savings in costs and resources.

However, AD suffers from the same problems that affect PM: the process of manually creating an AD program from scratch not only is often time-consuming and hardworking, but also requires a high level of programming expertise. This problem occurs when using AD for both creating original designs or exploring existing designs that were originally generated with AD (e.g., MVRDV's *Market Hall* [9]) or not (e.g., Gaudi's *Sagrada Família* [10]). In either case, the development of the AD program required a large amount of work by seasoned AD practitioners, which may explain why, despite its advantages, AD is still sparsely used.

To overcome this problem, we propose an approach called Reverse Algorithmic Design (RAD), which automatically translates a design manual-

ly produced in a CAD/BIM tool into a parametric AD representation of itself. RAD borrowed its name from reverse engineering [11], as both intend to infer a process that replicates an existing design. Note that the inferred process may or may not match the original one, as is often the case of RAD, which starts from a design that, in almost all cases, was not the result of an AD process. RAD can be applied at any stage of the design development to examine, recreate, or improve existing solutions.

RAD is composed of two main steps: (1) *extraction*, to generate a first approximation of the corresponding AD program; and (2) *refactoring*, to improve and parametrize the generated AD program. To be time- and cost-effective, RAD needs to be automated to the largest possible extent. Some AD tools can extract geometry into an AD program (e.g. GenerativeComponents), although with a low degree of automation, but are unable to infer parameters from patterns in the code. Since AD is a deterministic approach, its inference will be less ambiguous and thus able to deal with more complex designs, when compared to grammar inference processes.

RAD can bring several benefits, including: (1) the less error-prone and time-consuming generation of AD programs (when compared to manual AD processes) and, consequently, the faster exploration and optimization of complex designs; (2) the correction of mistakes/inconsistencies in a CAD/BIM model that are harder to detect manually; (3) the semi-automatic translation of designs from CAD to BIM [12]; and (4) the exploration of different parametric interpretations of a static model.

In the next sections, we detail the RAD methodology and illustrate its benefits by demonstrating its applicability in two case studies.

Methodology

RAD is the process of creating a reusable parametric AD description from an existing design created in CAD/BIM applications. The RAD approach entails two main steps:

1. *Extraction*: the automatic generation of an AD program from relevant information extracted from a digital design represented in a CAD or BIM tool, namely, geometric information (e.g., positions and dimensions) in both cases and semantic information (e.g., functions and materials) in the BIM case. Essentially, the information is retrieved from the CAD/BIM Application Programming Interface (API). Firstly, it is necessary to identify the type of entity we are dealing with (e.g., circles and walls), and then the associated information that defines that entity (e.g. dimensions and positions). Obviously, this information depends on what is stored in the API,

which may be a problem for more complex shapes, for instance the ones resulting from Boolean operations. The identified information is then recursively transformed to an executable program through *metaprogramming*, which is the use of programs to create other programs [13]. The resulting extracted AD program has a low abstraction level, as it can only generate the input design, and is hard to read and modify, as it is made of an arbitrarily ordered sequence of expressions (one per geometric/design element).

2. *Refactoring:* the semi-automatic improvement of the extracted AD program of the previous step, by increasing its abstraction level and legibility, while ensuring it still matches the design from which it was created. This task is performed by *refactoring*, which is the modification of the structure of a program without changing its semantics or external behavior [14]. To this end, well-known *refactoring operations* are used, which replace expressions by semantically equivalent ones. However, there are some situations where changing the semantics is desired, either to fix problems in the original design or to simply create a different design. Although *refactoring* is typically manually applied by the programmer, there are *refactoring tools* that verify the applicability of a given refactoring operation and make the corresponding changes automatically. Still, these tools require human guidance to better express the designer's intent, as it is hard to automatically guess the best way to improve a given program. To help the user to identify the parts of the program that need to be refactored and to facilitate the application of the refactoring operations, we rely on *traceability*, which is the ability to establish a relationship between a part of the program and the corresponding part of the generated design [15]. The resulting refactored AD program has a higher abstraction level and is easier to comprehend and modify than the extracted AD program, being able to generate not only the original design but also variations of it.

For testing purposes, this methodology was implemented in the Khepri AD tool [16]. By using the same algorithm, Khepri allows the visualization of designs in different CAD tools (e.g., AutoCAD and Rhinoceros), BIM tools (e.g., Revit and ArchiCAD), and game engines (e.g., Unity and Unreal). Moreover, it also supports the evaluation of designs, using different analysis tools (e.g., Robot and Radiance), and their optimization. Khepri's programing environment provides extraction, refactoring and traceability mechanisms.

Case Studies

The RAD methodology was applied to an automotive design project and a building renovation project. In the first case, the input was a 2D CAD model and the output was an algorithmic 2D CAD design. In the second case, the input was a 2D CAD building plan and the output was an algorithmic 3D BIM design. While the first case was selected to illustrate the RAD steps in detail, the second case was chosen as an example of the usefulness of the RAD approach in more complex designs. Both examples are described in the next sections.

Car Wheel

The first case study is the car wheel design presented in Fig. 1, which was manually produced in a CAD tool. The design is composed of circles, arcs, and lines. We applied the RAD methodology to add the following parameters to the design: center point (`p`), outer radius (`or`), inner radius (`ir`), hub radius (`hr`), starting angle (`sa`), outer arcs amplitude (`oaa`), inner arcs radius (`iar`), and spokes number (`n`).

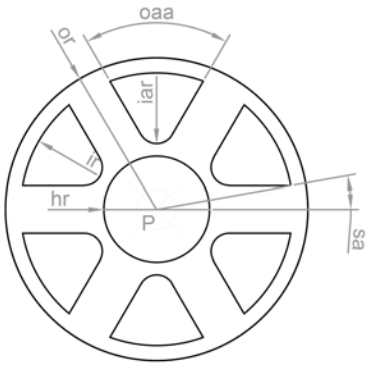

Fig. 1 Car wheel design manually produced in a CAD tool

In the RAD extraction step, the geometric entities presented in Fig. 2 (on the left) were converted into the AD program represented in Fig. 2 (on the right). Despite being a faithful representation of the original design, the extracted program is unquestionably hard to understand by humans. Therefore, for the RAD refactoring step, it is critical to visually relate the program with the design to help the designer understand the extracted program and decide the best refactoring operation to apply. This endeavor was facilitated by the traceability mechanism visible in Fig. 2, which shows the

relation between the selected shapes in the CAD tool and the corresponding program fragment highlighted in the AD tool (and vice versa).

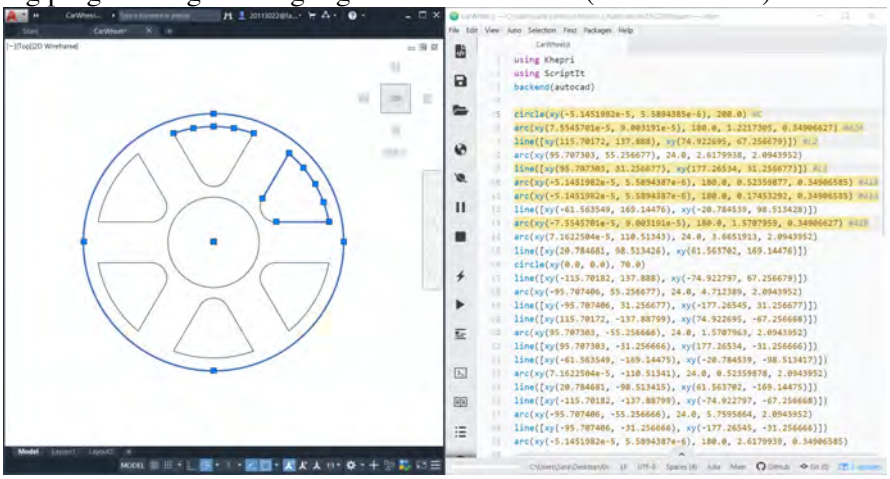

Fig. 2 Traceability between the shapes selected in a CAD tool (on the left) and the corresponding expressions highlighted in the AD program (on the right)

Several *refactoring operations* were applied to the extracted program, according to the terminology described in [14]. In this paper, we exemplify some of them by using the highlighted extracted code of Fig. 2:

```
circle(xy(0, 0), 200)
arc(xy(0, 0), 180, 7pi/18, 2pi/18)
line(xy(74.92, 67.26), xy(115.70, 137.89))
line(xy(95.71, 31.26), xy(177.27, 31.26))
arc(xy(0, 0), 180, 3pi/18, 2pi/18)
arc(xy(0, 0), 180, pi/18, 2pi/18)
arc(xy(-5.15e-5, 5.59e-6), 180, 9pi/18, 2pi/18)
```

As observable, the generated program presents geometric primitives (circles, arcs, and lines) in sequences of expressions with an arbitrary order. The following syntax was used: a circle centered at (0, 0) with a radius of 200 is represented as `circle(xy(0, 0), 200)`; an arc centered at (0, 0), with a radius of 180, a starting angle of π/18, and an arc amplitude of 2π/18, is represented as `arc(xy(0, 0), 180, pi/18, 2pi/18)`; and a line starting at (1, 2) and ending at (3, 4) is represented as `line(xy(1, 2), xy(3, 4))`.

Given the obvious radial symmetry of the design, we started by using the refactoring operation *Substitute Algorithm* to convert the Cartesian co-

ordinate system (represented by the operation `xy`) into the polar coordinate system (represented by the operation `pol`), obtaining the following result:

```
circle(pol(0, 0), 200)
arc(pol(0, 0), 180, 7pi/18, 2pi/18)
line(pol(100.68, 0.73), pol(180, 5pi/18))
line(pol(100.68, 0.32), pol(180, pi/18))
arc(pol(0, 0), 180, 3pi/18, 2pi/18)
arc(pol(0, 0), 180, pi/18, 2pi/18)
arc(pol(0, 0), 180, 9pi/18, 2pi/18)
```

After this refactoring, we can see that some patterns emerge, namely the start and endpoints of the lines having the same radius. Moreover, precision errors were corrected, as it happened with the center of the last arc.

The next operation applied was the *Slide Statements* refactoring to sort the expressions in ascending order, as shown below:

```
arc(pol(0, 0), 180, pi/18, 2pi/18)
arc(pol(0, 0), 180, 3pi/18, 2pi/18)
arc(pol(0, 0), 180, 7pi/18, 2pi/18)
arc(pol(0, 0), 180, 9pi/18, 2pi/18)
circle(pol(0, 0), 200)
line(pol(100.68, 0.32), pol(180, pi/18))
line(pol(100.68, 0.73), pol(180, 5pi/18))
```

Note that the unordered nature of the original design allows us to freely reorder the program expressions without side-effects. From this arrangement, we can see that the code can be simplified: as the first two arcs are contiguous, the second arc can be removed if we double the angle amplitude of the first. The same reasoning was applied to the third and fourth arcs:

```
arc(pol(0, 0), 180, pi/18, 2pi/9)
arc(pol(0, 0), 180, 7pi/18, 2pi/9)
circle(pol(0, 0), 200)
line(pol(100.68, 0.32), pol(180, pi/18))
line(pol(100.68, 0.73), pol(180, 5pi/18))
```

To parametrize the program, we used the refactoring operation *Extract Variable*, which extracts parameters from identical expressions. In this case, we added the center point (`p`), the inner radius (`ir`), and the outer arcs amplitude (`oaa`) parameters, which are then automatically replaced in the expressions. This refactoring operation formalizes what could be inferred

by looking at the original design: the circle and the outer arcs have the same center. The resulting expressions are shown below:

```
p = pol(0, 0)
ir = 180
oaa = 2pi/9
arc(p, ir, pi/18, oaa)
arc(p, ir, 7pi/18, oaa)
circle(p, 200)
line(pol(100.68, 0.32), pol(ir, pi/18))
line(pol(100.68, 0.73), pol(ir, 5pi/18))
```

The arcs described above belong to the group of six outer arcs that form the dashed circle. If we compute the angular separation between the arcs, we find a constant value that is equal to 2π divided by six. This means that we can apply a *Loop Re-rolling* refactoring to simplify the six arc expressions in a *for* loop. Also, we added two new parameters: the starting angle (`sa`) of the first arc and the number of arcs (`n`):

```
sa = pi/18
n = 6
for i in division(0, 2pi, n) arc(p, ir, sa+i, oaa) end
circle(p, 200)
line(pol(100.68, 0.32), pol(ir, pi/18))
line(pol(100.68, 0.73), pol(ir, 5pi/18))
```

The same reasoning was applied to the inner arcs and lines, and then the loops were combined with the *Loop Fusion* refactoring. We also added the remaining variables described in Fig. 1 (`or`, `hr`, `oaa`, `iar`). To complete the program transformation, we applied *Extract Function*, which groups the selected expressions in a function, and *Parameterize Function*, which transforms the variables into function parameters. The resulting algorithm is shown below (the internal variables are omitted for clarity):

```
car_wheel(p, ir, oaa, sa, n) =
  ...
  circle(p, or), circle(p, hr)
  for i in division(0, 2pi, n)
    arc(p, ir, sa+i, oaa)
    arc(p+vpol(iad, sa+oaa/2+i), iar, sa+oaa/2+pi-iaa/2+i, iaa)
    line(p+vpol(ld, sa+oaa/2-la+i), p+vpol(ir, sa+i))
    line(p+vpol(ld, sa+oaa/2+la+i), p+vpol(ir, sa+oaa+i))
  end
```

```
car_wheel(pol(0, 0), 200, 180, 70, pi/18, 2pi/9, 24, 6)
```

The result of the refactoring process is a parametric algorithmic version of the design presented in Fig. 1 that is almost three times smaller than the first extracted version. The refactored program is much easier to understand and more abstract than the extracted one. When executed, the program can generate not only the original design but also parametric variations of it, as shown in Fig. 3. The resulting AD program can now be used, for example, to explore variations within a brand's language, or for optimization of cost or performance.

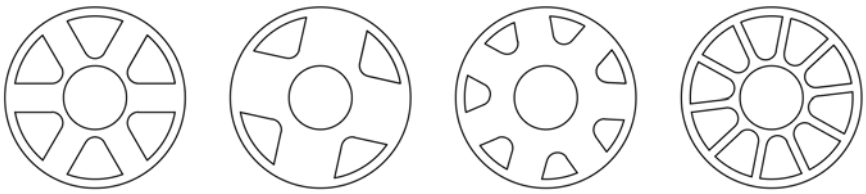

Fig. 3 Original design (first on the left) and parametric variations (remaining)

Bauhaus Building

The second case study is a part of the Bauhaus building in Dessau. The starting point was a CAD floor plan of the school (visible in Fig. 4). This time, the RAD methodology was applied to parameterize the door's width, in order to then optimize the building regarding its evacuation performance. For this problem, the model was reduced to walls and doors.

In this case, the extraction produced a series of lines, such as the ones presented below:

```
line(xy(0, 0), xy(4, 0), xy(4, 0.3), xy(0, 0.3), xy(0, 0))
line(xy(5, 0), xy(8, 0), xy(8, 0.3), xy(5, 0.3), xy(5, 0))
```

To make the program easier to understand, intermediate abstractions were introduced to clarify what the program was doing. In the expressions above, the shape of the polygonal line is not entirely obvious. However, equivalent expressions can represent the same shapes in a much more understandable way:

```
rectangle(xy(0, 0), xy(4, 0.3))
rectangle(xy(5, 0), xy(8, 0.3))
```

In this case, we explored traceability not only to help refactor the program but also to help establish a correspondence between geometric ele-

ments (such as lines and rectangles) and building elements (such as walls and doors). The conversion of the aforementioned rectangles to walls with default height and represented by their axis is straightforward:

```
wall(line(xy(0, 0), xy(4, 0)))
wall(line(xy(5, 0), xy(8, 0)))
```

To generate doors, the program identifies empty spaces between contiguous walls (such as the ones presented above) and, with the user's consent, joins the walls and adds a door in the empty space:

```
door(wall(line(xy(0, 0), xy(8, 0))), xy(4, 0))
```

The same reasoning was applied throughout the program, obtaining a refactored program that is more than three times smaller than the extracted one. The final refactored AD program can be executed to generate 3D designs in a CAD tool or a BIM tool, as illustrated in Fig 5.

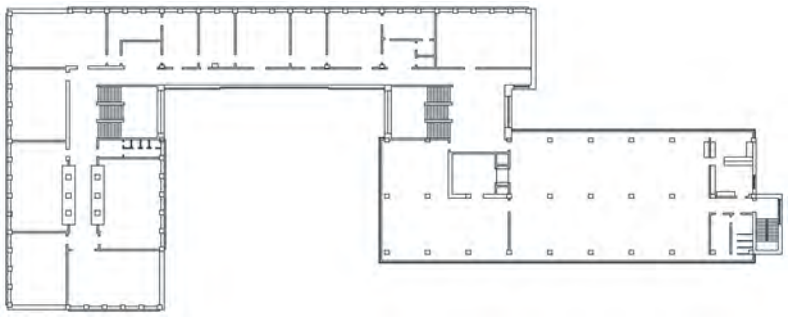

Fig. 4 2D CAD model used to generate the AD program

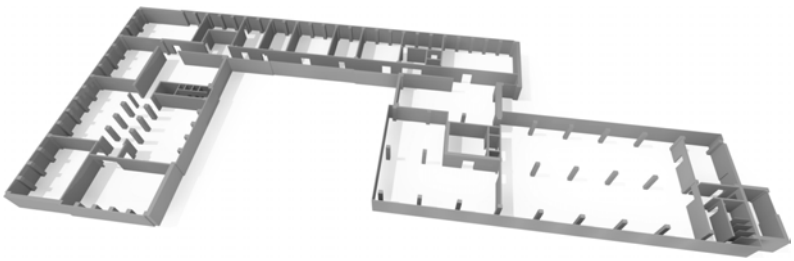

Fig. 5 3D BIM model generated by the refactored AD program

The resulting parametric AD program was used to optimize the evacuation performance of the building, showing that it is possible to have a 20%

reduction in the evacuation times just by enlarging the doors' width by 5% [17]. With this case study, the RAD approach proved to be efficient in the semi-automatic conversion from CAD models to BIM models, with the bonus of making the model parametric. Further studies will have to be conducted to demonstrate what we can perceive from this case: that RAD can significantly reduce the time and effort in producing an AD program.

Conclusion

Algorithmic Design (AD) is an approach where multiple design variations are represented by a parametric algorithm. AD automates repetitive tasks, facilitates the exploration of complex designs, and supports design optimization. Despite its advantages, most designs are not created using AD, since their development takes a considerable amount of time, effort, and expertise. Instead, they are manually produced in CAD/BIM tools.

In this paper, we presented Reverse Algorithmic Design (RAD), an approach to semi-autonomously translate a design produced in a CAD/BIM tool into an equivalent AD program. The RAD approach comprises two main steps: (1) the automatic extraction of a low-level AD program from relevant information described in a CAD/BIM model; and (2) the human-guided semi-autonomous refactoring of the previous program, resulting in a parametric AD program that has a higher abstraction level and is more comprehensible than the former one. Traceability, a visual relation between model and program, is used in the second step to help the designer choose which parts of the program to refactor and which refactoring operations to apply. These operations then autonomously transform the program while ensuring that no errors are introduced.

The RAD methodology was successfully applied in the generation of AD programs from a 2D car wheel and a 2D building plan. The experiments showed that the RAD approach can efficiently convert CAD/BIM models into AD programs, while suggesting parametric interpretations of static models. Moreover, RAD promotes the detection of errors in existing models and speeds up the conversion of CAD to BIM models.

For future work, we plan to apply the RAD methodology to a 3D model input. Moreover, we intend to automatize even further the conversion from CAD to BIM, including the use of machine learning techniques to automatically infer semantic information from geometric information.

Acknowledgements: This work was supported by national funds through FCT (Fundação para a Ciência e a Tecnologia), under projects UIDB/50021/2020 and PTDC/ART-DAQ/31061/2017.

References

1. Šťava O, Beneš B, Měch R, Aliaga DG, Krištof P (2010) Inverse Procedural Modeling by Automatic Generation of L-systems. In: Comput. Graph. Forum. pp 665–674
2. Wu F, Yan D-M, Dong W, Zhang X, Wonka P (2014) Inverse Procedural Modeling of Facade Layouts. ACM Trans Graph 33:121:1–121:10
3. Talton J, Yang L, Kumar R, Lim M, Goodman N, Měch R (2012) Learning Design Patterns with Bayesian Grammar Induction. In: Proc. 25th Annu. ACM Symp. User Interface Softw. Technol. pp 63–73
4. Orsborn S, Cagan J, Boatwright P (2008) A methodology for creating a statistically derived shape grammar composed of non-obvious shape chunks. Res Eng Des 18:181–196
5. Whiting M, Cagan J, LeDuc P (2016) Automated Induction of General Grammars for Design. In: Gero JS (ed) DCC 16. Springer, pp 283–296
6. Bokeloh M, Wand M, Seidel H-P (2010) A Connection between Partial Symmetry and Inverse Procedural Modeling. ACM Trans Graph 29:104:1-104:10
7. Gips J (1999) Computer Implementation of Shape Grammars. Workshop Shape Comput.
8. Terzidis K (2006) Algorithmic Architecture. Architectural Press, Oxford, UK; Burlington, USA
9. Leitão A, Fernandes R, Santos L (2013) Pushing the Envelope: Stretching the Limits of Generative Design. In: Knowl.-Based Des. Proc. 17th Conf. SIGraDi, pp 235–238
10. Burry M (2011) Scripting Cultures: Architectural Design and Programming. John Wiley & Sons, United Kingdom
11. Chikofsky EJ, Cross JH (1990) Reverse engineering and design recovery: a taxonomy. IEEE Softw 7:13–17
12. Lim J, Janssen P, Stouffs R (2018) Automated Generation of BIM Models from 2D CAD Drawings. In: Learn. Adapt. Prototyp. Proc. 23rd CAADRIA, pp 61–70
13. Czarnecki K, Østerbye K, Völter M (2002) Generative Programming. In: Hernández J, Moreira A (eds) Object-Oriented Technol. ECOOP 2002 Workshop Read. Springer-Verlag, Berlin, Heidelberg, pp 15–29
14. Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman, Reading, Massachusetts
15. Leitão A, Lopes J, Santos L (2014) Illustrated Programming. In: Proc. 34th Annu. Conf. ACADIA. pp 291–300
16. Leitão A, Castelo-Branco R, Santos G (2019) Game of Renders: The Use of Game Engines for Architectural Visualization. In: Haeusler MH, Schnabel MA, Fukuda T (eds) Proc. 24th CAADRIA Conf, pp 655–664
17. Leitão A, Sousa S, Loio F (2018) SafePath: An Agent-Based Framework to Simulate Crowd Behaviors. In: Kępczyńska-Walczak A, Bialkowski S (eds) Comput. Better Tomorrow Proc. 36th ECAADe Conf, pp 621–628