The Collaborative Algorithmic Design Notebook

Renata Castelo-Branco[1], Inês Caetano[2], Inês Pereira[3], and António Leitão[4]
[1, 2, 3, 4] *INESC-ID, Instituto Superior Técnico, University of Lisbon, Portugal*
{renata.castelo.branco[1], ines.caetano[2], ines.pereira[3], antonio.menezes.leitao[4]}@tecnico.ulisboa.pt

Abstract: Design studios are increasingly interested in collaborative Algorithmic Design (AD) practices. However, AD uses algorithmic descriptions that are often difficult to understand and change, hindering the adoption of AD in collaborative design. We propose improving collaborative AD by adopting a programming environment that allows for (1) developing designs incrementally, (2) documenting the history of the design with visual artifacts, (3) providing immediate feedback during the development process, and (4) facilitating the comprehension of collaborative AD projects. Given that computational notebooks are currently being explored in scientific research to solve similar problems, in this paper, we adapt the notebook workflow to support collaborative AD processes, outlining three main collaboration strategies. We evaluate currently existing notebook environments regarding each of the strategies and the impact they have on the development of AD projects.

Keywords: Collaborative design; algorithmic design; computational notebook.

1. Introduction

The increasing complexity of current architectural design projects motivates the need for design teams to integrate different experts, creating more collaborative design environments (Laing, 2019). This allows the design process to benefit from more diverse knowledge and different problem-solving perspectives, leading to an increase in the quality of the solutions (Self, 2019). The collaborative scenario, however, requires design representations that are understandable by all participants.

Architects use sketches, drawings, models, diagrams, mock-ups, etc., to represent and communicate their ideas (Bresciani, 2019; Buxton, 2007). However, the introduction of Algorithmic Design (AD), which is based on algorithmic representations of the designs (Caetano *et al.* 2020), creates a problem, as algorithms tend to be both difficult to understand and develop by designers (Woodbury, 2010). This hinders the adoption of AD techniques within collaborative design environments, as the communication of ideas between team elements is a hard and far-from intuitive task. To address this, we need to bring AD closer to the architectural design needs and to the visual nature of the profession, making it suitable for collaborative design processes.

We believe improving AD representation methods requires a programming environment that not only allows for visually documenting the developed designs but also supports the incremental development of AD programs in an integrated storytelling experience that is critical for collaborative

Imaginable Futures: Design Thinking, and the Scientific Method. 54th International Conference of the Architectural Science Association 2020, Ali Ghaffarianhoseini, et al (eds), pp. 1056–1065.

design. The notebook philosophy (Rule *et al.* 2018), which is used in different scientific fields, addresses some of these issues, allowing users to intertwine text, formulas, data, code, and graphics in the same document, telling the story of an experiment and, thus, supporting its reproducibility, while improving its understandability and sharing among multiple participants.

2. Algorithmic design

AD is a design approach that relies on algorithms to create designs, making it possible to automate tedious and time-consuming tasks and generate more complex design solutions. By using algorithms, designers can parametrically describe designs, which means changes are automatically propagated, avoiding design errors as well as reducing the time needed to explore different solutions (Burry, 2011). AD increases architects' creativity and productivity (Terzidis, 2003) but requires programming knowledge and abstract thinking capabilities (Woodbury, 2010).

A recent study (Gardner, 2018) revealed that many practitioners find a barrier of technical complexity in AD and some still view it as a non-humane way to design and as an obstacle to creativity. The issue might be explained, in part, by the fact that conventional programming languages are difficult to learn and use, particularly so for non-expert programmers, which constitutes the great majority of the architectural community. Additionally, AD programs frequently turn up to be the unstructured product of successive *copy&paste* of program elements, which result from the experimentation process that characterizes design thinking (Woodbury, 2010). Hence, programming architects find it difficult to understand AD programs that represent complex designs, particularly those developed by others.

To improve program understanding, Donald Knuth proposed Literate Programming (1984) as a way to conceive comprehensible computer programs. Literate Programming is based on the idea that a program should be developed as a literary work, written to be read by humans and, thus, documenting not only the implementation details but, more importantly, the rationale behind it. Unfortunately, the idea is not directly applicable to architecture as most documentation strategies only provide textual explanations and architecture prefers visual ones, such as sketches and schematics.

3. The computational notebook

An interesting take on this topic has been forwarded by the computational notebook philosophy (Rule *et al.* 2018), starting with Mathcad and Wolfram's Mathematica, and following up to more recent tools like the Jupyter notebook. These tools allow for incremental development with immediate feedback on the program's results, as well as the intertwining of code, textual and visual documentation, namely mathematical formulas, plots, and rich media, generated by the program itself or loaded from external sources. Notebooks were designed to support computational narratives, allowing users to execute, document, and communicate their experiments, which appeals to a vast audience particularly in the academic and scientific fields.

In the case of AD, computational notebooks show great potential to aid the development of design projects, since architects can have their sketches and explanatory texts within the same environment where the algorithmic description is being created. Furthermore, the notebook philosophy fosters incremental development, which in itself motivates a more responsive and comprehensible coding activity. Finally, it allows users to make their work reproducible and, along with improved program comprehension, it has the potential to streamline collaborative development of AD projects.

4. The collaborative algorithmic design notebook

We propose extending the notebook concept to improve the experience of developing and sharing AD projects, reducing existing barriers to the adoption of AD. For that, the ideal AD notebook should allow for (1) interactive code development, enabling the incremental writing and testing of program fragments, and the reactive manipulation of variables to easily assess the impact of parameters in the design; (2) storytelling, preserving the trail of program changes and consequent tests available for later consultation; and, finally, (3) collaborative development, including the ability to create programs without local dependencies, whose results are reproducible anywhere by anyone.

Features 1 and 2 are already supported by many currently available notebook tools. Collaboration over notebooks, on the other hand, has several shortcomings (Chattopadhyay *et al.* 2020), particularly in the AD domain. This paper focuses on the collaborative capabilities of notebooks, through the analysis of three different collaborative strategies (Figure 1) and their evaluation in the development of a case study. (1) Pair, (2) synchronous, and (3) independent programming are common scenarios in any software development endeavour and several solutions exist to support them. The case of shared notebooks, however, introduces several issues that need to be addressed.

We analysed currently existing notebooks and respective collaboration mechanisms, tested different alternatives for each of the collaborative scenarios, and identified their advantages and disadvantages. This analysis allowed us to outline the characteristics we find necessary in a notebook capable of supporting all three collaborative scenarios in an intuitive and user-friendly manner.

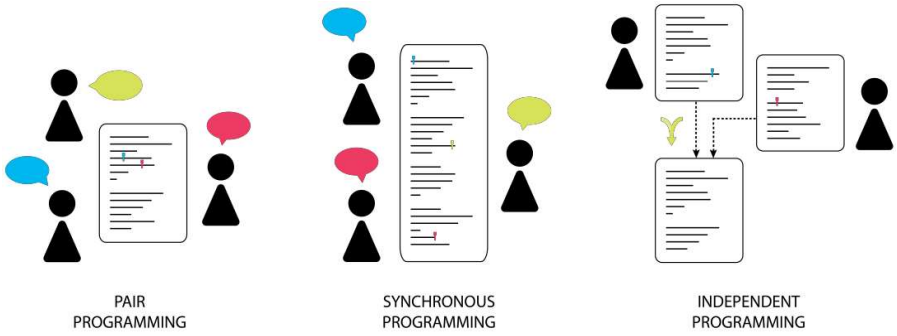

Figure 1: Collaborative scenarios: in (1) pair programming developers iterate synchronously over the same code parcel, in (2) synchronous programming they work concurrently in different sections of the same file, and in (3) independent programming they work separately and later merge their work.

4.1. Pair programming

With live collaboration in the workspace, traditional pair programming (Williams and Kessler, 2003) implies two developers working on the same computer simultaneously, discussing design ideas and then, while one implements the devised solutions (the driver), the other supervises what the former is doing (the navigator), providing advice and ensuring the implementation reflects the discussed ideas. Normally, after a while, they switch roles.

Remote collaboration, on the other hand, allows for other possibilities, namely distributed pair programming (Stotts *et al.* 2003; Baheti *et al.* 2003), which involves more than two developers collaborating remotely. There are two main strategies for this workflow: (1) terminal sharing and (2) screen sharing, with or without screen annotation capabilities. The first one allows colleagues to work on the same code base simultaneously, tracking each other in the document. In the second strategy, users do not share a terminal, but rather one of them simply shares his view of the code with others. This modality is frequently called the token system, wherein only one person can edit a file at a time. This eliminates some of the editing confusion that may arise with more than two participants. For remote collaboration, however, it may also be hard to express programming ideas without specifically pointing at code pieces or even demonstrating the idea textually.

The pair programming workflow is most beneficial for difficult developments where the expertise of several co-workers is required. Having several participants working on the same task usually ensures a better outcome, not only guaranteeing continuous code review, but also motivating a more focused mind-set. However, it is also a waste of resources; ergo, other collaborative workflows are required.

4.2. Synchronous programming

Synchronous programming implies having several developers simultaneously and concurrently working on the same program file, in this case a notebook. In synchronous programming, users are not only aware of each other, but also sharing the same programming environment in real-time and the same program state, meaning any changes made by one of the developers will automatically impact the program for all developers in session. This workflow is the coding equivalent to the shared document development popularized by GoogleDocs. Synchronous programming is most useful when time runs short, allowing multiple developers to edit different notebook sections simultaneously. However, it requires a more strategic work coordination (Wang *et al.* 2019) to avoid repeated work. To make the workflow more productive, co-workers should speak to each other during the shared session.

Guaranteeing reproducibility and consistency among different participants requires running the program on the server of the company providing the sharing service, to guarantee all users have access to the exact same program state. However, there are frequent limitations to what users can add or install on companies' servers, for both space and safety reasons. On account of communication delays, performance is also typically worse than on a local system. The first two problems are often resolved with more expensive versions of the collaborative software, but not the latter.

4.3. Independent programming

In this workflow, co-workers separately develop, in their own time, different program files or different sections of the same file and combine them from time to time. This technique is often used in large software development endeavours, where projects can be divided in several parts to be distributed among developers. Merging, however, can be challenging, hence, requiring elaborate version control systems capable of comparing different versions of the same file and settling out any conflicts that may arise. This provides a more productive workflow, since users can work simultaneously on the same file.

5. Related work

This section presents an overview of currently available notebook platforms and their native collaboration mechanisms, as well as available version control and collaboration alternatives to

complement them. We analysed seven different computational notebook platforms, namely Wolfram's Mathematica, Jupyter Notebook, NextJournal, Deepnote, CoCalc, Pluto, and MathCAD, to identify the options that are most suitable for a collaborative AD practice.

5.1. Computational notebook platforms

Most notebooks are based on input cells that, once run, produce immediate output. Only MathCAD uses a different approach, recreating the traditional math checkered notebook. Moreover, users do not need to know programming to use it. Both MathCAD and Pluto promote reactivity, meaning that when one expression is changed, the dependent expressions are automatically updated. With the exception of these two, all notebooks support multiple programming languages, particularly interactive ones that promote incremental development, such as Python, Julia, and R. Only Mathematica and Jupyter allow users to mix languages in the same notebook, and Jupyter even allows users to install new languages.

Regarding collaborative work, MathCAD does not support version control nor real-time collaboration, but the notebooks produced look like pages from a manual, which facilitates the comprehension of the project across the different collaborators. Jupyter and Pluto also do not allow different users to develop code simultaneously, but an approximation to collaborative work can be achieved by using the supported applications for version control and code sharing, such as GitHub or Binder. Mathematica Online allows users to add collaborators and work in real time, as well as save the version history of the notebook. Nextjournal, Deepnote, and CoCalc, are all cloud-based and natively prepared for real-time collaborative work. The three of them support versioning, with Nextjournal and CoCalc doing so automatically. Deepnote offers code review options and allows users to directly connect with other collaborative tools, such as GitHub and GitLab. CoCalc allows users to track each other and access the history of the project per collaborator and has a built-in text and audio chat, allowing for more diverse strategies of collaborative work, such as pair programming.

5.2. Collaboration mechanisms

Regarding screen sharing, plenty of communication tools will do the job (e.g., Skype, Teams, or Facebook's Messenger). Nevertheless, with this workflow, it frequently becomes hard to vocally express programming intentions when making suggestions. This can be addressed with more elaborate communication platforms that turn the screen into an interactive whiteboard (e.g., Zoom or TeamViewer), or that support a terminal sharing workflow (e.g., USE Together), which allows viewers to interact with the program as well, although it is running in the host's machine entirely. The latter solution facilitates interaction, but the guests are still limited to the view of the screen shared by the host at any moment, which means it does not suffice for Synchronous Programming.

Sharing notebooks can be achieved using simple file sharing applications (e.g., Dropbox or OneDrive) but since these do not have merging capabilities, users must manually handle versioning and conflict resolution. For the task at hand, it is simpler and more efficient to use specialized services such as GitHub. However, conflict resolution in notebooks can be tricky because the underlying representation does not meet the expectations of the typical comparison tools used in those services. For this reason, specialized comparison tools have been created to handle notebook version control (e.g., NBdiff and NBdime). Nevertheless, the learning curve for these tools should also be considered.

6. Evaluation

To evaluate the proposed collaborative workflows, we developed a case study inspired by the Irina Viner-Usmanova Rhythmic Gymnastics Centre, in Moscow, Russia, originally designed by the CPU Pride office. An AD notebook adaptation of this project was developed in five working days by three architects. Figure 2 presents a schematic chronogram of the collaborative development process. The project was modelled using the Julia programming language and Khepri (Sammer *et al.* 2019), an AD tool that is portable between multiple design tools and that was recently extended to be integrated in the notebook format. Based on the information collected previously, we selected the tools we considered most suitable for the three collaborative scenarios - Jupyter, Nextjournal, and Pluto -, as well as two version control systems - GitHub with and without NBdime - and two communication platforms - Skype and USE Together. The following paragraphs describe the application of the three collaborative scenarios in the development of the case study.

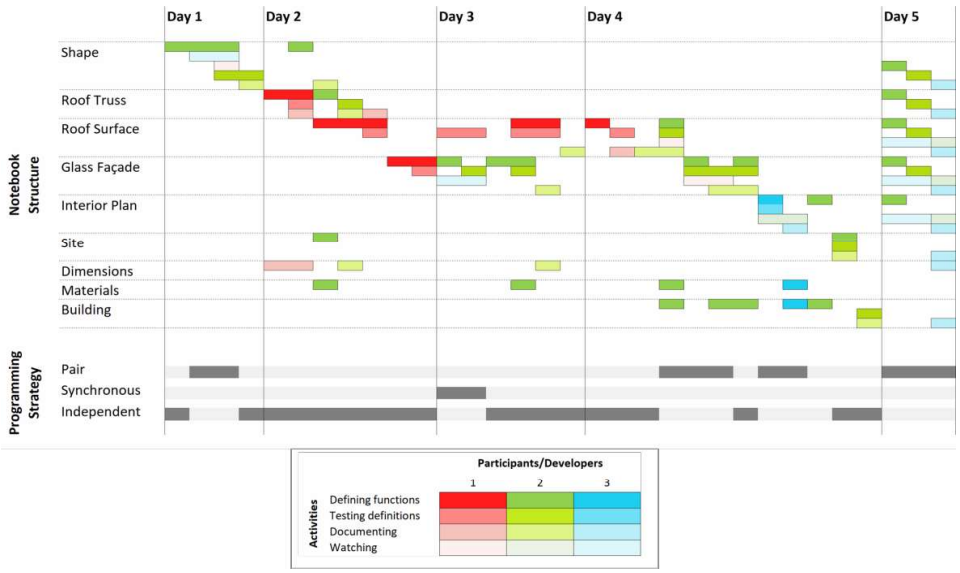

Figure 2: Diagrammatic schedule of the case study's development process.

Pair programming was used several times during the development process with two main tools: Skype for screen sharing and USE Together for terminal sharing. Screen sharing occurred when developers (1) felt they needed help with a code piece, (2) wanted to share ideas, or (3) had doubts about each other's work. Terminal sharing was used for more detailed help sessions. For instance, the USE Together tool was used by more experienced programmers to remotely guide less experienced ones. The tool allowed them to exemplify things directly in the latter's computer, which facilitated communication. However, this has some disadvantages, namely the performance penalty and the

confusions resulting from the occasional simultaneous inputs provided by two different users. Figure 3 shows a screenshot of this workflow.

For **synchronous programming** we used Nextjournal, which avoided the need for merges, as the developers were concurrently working on the same file in the server. However, since no two cells can run simultaneously, we could only push one execution onto the server at a time, thus having to repeatedly wait for each other's processes to finish. Because of this, we ended up doing more pair than synchronous programming.

Independent programming occurred throughout most of the project. All three developers kept committing their changes to the various notebooks in use in the same GitHub project. Using this platform, there were no restrictions to simultaneous file use, which led to several conflicts when using the Jupyter notebook. Even when users were developing separate areas of the notebook, usual text- based version control tools (e.g., Git) could not sort out the merge, which had, thus, to be settled using NBdime. Figure 4 shows the NBdime diff tool in use for the Jupyter notebook. We also tested the independent development workflow using Pluto. Since this notebook's underlying file representation is human-readable, merges are a lot easier and can frequently be automatically handled by GitHub. Figure 5 presents a successful merge of two committed versions of the Pluto notebook by developers 1 and 2.

The Pluto notebook, however, forced a rather different workflow to notebook development from the one we had been employing, for instance, in Jupyter. Due to Pluto's reactive evaluation strategy, it was not possible to keep the program's evolution history since repeated definitions are not allowed. In the end, we decided to maintain two concurrent versions of the notebook: one in Jupyter, where we kept the development history, and a summarized version in Pluto. The final product of this collaborative notebook sprint can be found at https://github.com/KhepriNotebook/GymnasticsPavilion_Moscow.

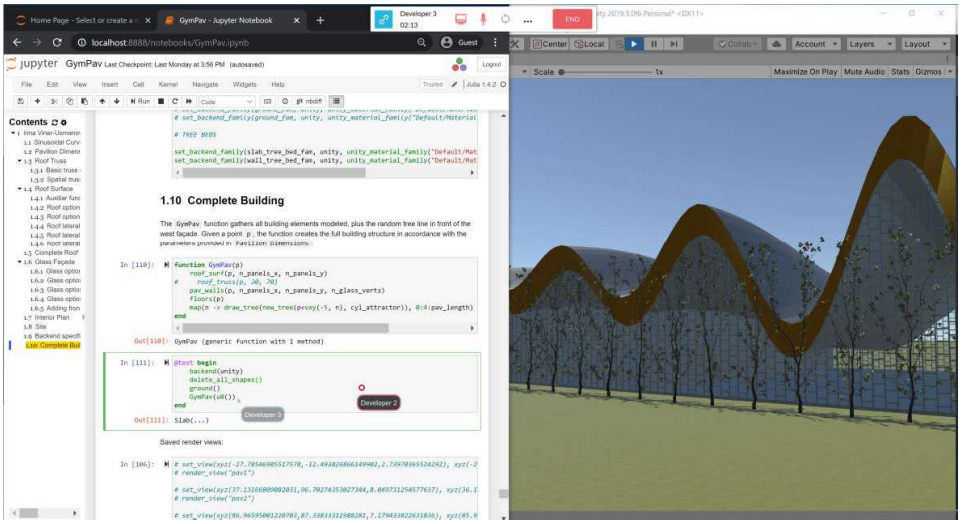

Figure 3: Developers 2 and 3 working on the Jupyter notebook with USE Together.

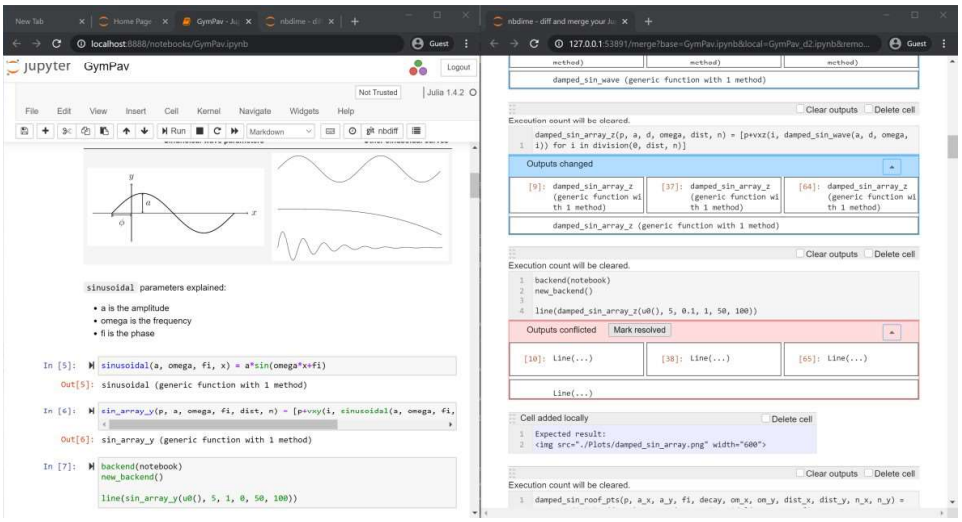

Figure 4: The NBdime tool comparing conflicting versions of the Jupyter notebook via GitHub.

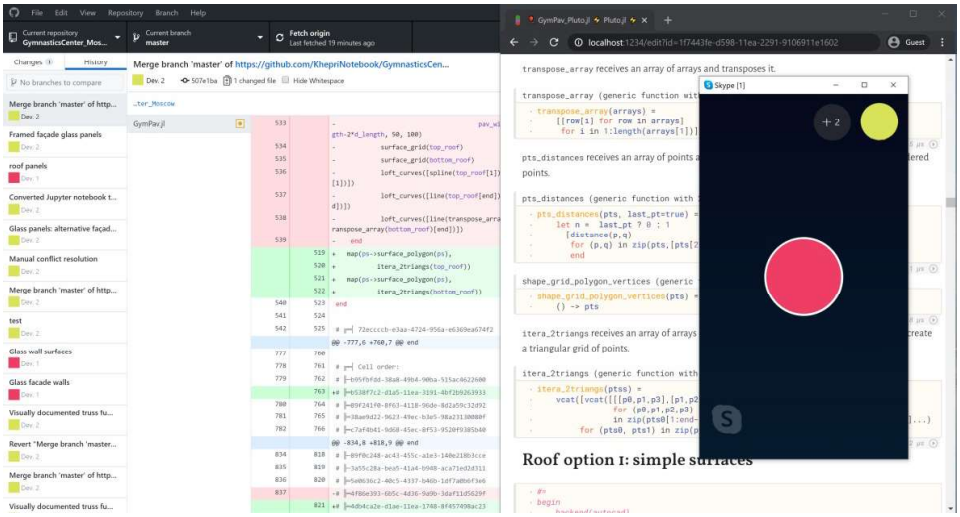

Figure 5: GitHub desktop application autonomously merging two independent versions of the Pluto notebook file submitted by developers 1 and 2.

7. Conclusion

Architecture and, by extension, Algorithmic Design (AD), requires the elaboration of design hypotheses, their evaluation, and their optimization to achieve designs with good performance. However, the design process is strongly based on visual and spatial reasoning, which is not always easy to translate onto computational descriptions. Consequently, AD programs are generally hard to develop and understand. These problems are aggravated in collaborative design, where multiple participants need to be involved in the development of an AD program and need to understand each other's work.

To address these problems we improved the understandability of algorithmic descriptions by adapting computational notebooks for AD development, allowing architects to collaboratively intertwine text, formulas, data, code, and graphics in a way that not only preserves the story of a design but also supports the reproducibility of the AD process.

7.1. Ideal notebook features

Given the different advantages and limitations of existing computational notebooks, we do not endorse any specific one, formulating instead three principles we believe an ideal notebook should include.

Interactive code development motivates users to immediately test program fragments, visualizing the corresponding result. These tests also serve as documentation for users who wish to verify if the program is yielding the proper results. All the notebooks used in the case study – NextJournal, Jupyter, and Pluto – possess this feature, although they deal with state and interactiveness in different ways. While the former two offer users complete freedom over the evaluation order of the cells, the latter restricts it for the sake of reactivity. The latter approach prevents the accumulation of bugs that arise in less restrained workflows: by automatically propagating the changes along the program, users are immediately informed about the result and errors that might have been introduced.

Storytelling preserves the development history, which includes program changes and tests, presenting it in a comprehensible way to help future developers understand and maintain the AD program. In this regard, while Jupyter and NextJournal allow users to more loosely define and redefine program structures, Pluto always imposes program consistency, not allowing outdated code fragments in the middle of the notebook. While the former two allow for telling the design development story by keeping the different stages of the program in the notebook itself, the latter requires the use of version control mechanisms. However, absolute freedom frequently motivates chaos that, in this case, results in inconsistent program state. Hereupon, Pluto leads to more organized and coherent programs that are easier to comprehend by third parties. We believe an ideal notebook should join the best of both: allowing for keeping the development history and, if desired, hiding it away for final presentation.

Collaborative development requires the ability to coordinate multiple developers. Although none of the used notebooks proved to suit the three outlined collaboration scenarios, we could achieve them by using different combinations of existing tools. We conclude an ideal environment for collaborative AD should provide efficient mechanisms for pair, synchronous, and independent programming.

7.2. Future work

As future work, we plan on researching outlining techniques capable of hiding intermediate program developments or program extraction techniques that can retrieve a summarized version of an AD

notebook containing the final program only. Refactoring techniques may also help improve the typical scruffy nature of incrementally developed AD notebooks for explanation purposes.

We also plan to address education scenarios, an area where computational notebooks have been showing promising results. One of the main setbacks for the applicability of AD approaches in architectural projects lies in the difficulty of learning to represent designs in the form of algorithms. Using notebooks as learning environments, along with the proposed collaboration strategies, can help overcome this barrier, particularly when it comes to remote teaching.

Acknowledgements

This work was supported by national funds through FCT, *Fundação para a Ciência e a Tecnologia*, under project references UIDB/50021/2020 and PTDC/ART-DAQ/31061/2017, and by the PhD grant under contract of FCT with reference SFRH/BD/128628/2017.

References

Baheti, P., Gehringer, E. and Stotts, D. (2002) Exploring the Efficacy of Distributed Pair Programming, in Wells, D. and Williams, L. (ed.), *Extreme Programming and Agile Methods - XP/Agile Universe 2002*, Springer Berlin Heidelberg, 208–220.

Bresciani, S. (2019) Visual Design Thinking: A Collaborative Dimensions framework to profile visualisations, *Design Studies*, 63, Elsevier Ltd., 92–124.

Burry, M. (2011). *AD Primers: Scripting Cultures: Architectural Design and Programming*, West Sussex, UK : John Wiley & Sons Ltd.

Buxton, B. (2007). *Sketching User Experience: Getting the Design Right and the Right Design*, Morgan Kaufmann. [Online]. Available at: doi:10.1016/B978-0-12-374037-3.X5043-3.

Caetano, I., Santos, L. and Leitão, A. (2020) Computational Design in Architecture: Defining Parametric, Generative, and Algorithmic Design, *Frontiers of Architectural Research*, 9, 287-300.

Chattopadhyay, S., Prasad, I., Henley, A., Sarma, A. and Barik, T. (2020) What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities, in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ACM, New York, USA, 1–12.

Gardner, N. (2018) Architecture-Human-Machine (re)configurations Examining computational design in practice, in *Computing for a better tomorrow - Proceedings of the 36th eCAADe Conference*, Lodz, Poland, 139–148.

Knuth, D. (1984) Literate Programming, *The Computer Journal*, 27(2), 97–111.

Laing, R. (2019) *Digital Participation and Collaboration in Architectural Design*, Routledge: Taylor & Francis Group.

Rule, A., Tabard, A. and Hollan, J. D. (2018) Exploration and explanation in Computational notebooks, in *Conference on Human Factors in Computing Systems – Proceedings*, ACM, Montreal, Canada, 1–12.

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

Stotts, D. et al. (2003) Virtual Teaming: Experiments and Experiences with Distributed Pair Programming, in Maurer, F. and Wells, D. (ed.), *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, Springer Berlin Heidelberg, 129–141.

Terzidis, K. (2003) *Expressive Form: A Conceptual Approach to Computational Design*, Spon Press, New York.

Wang, A. Y., Mittal, A., Brooks, C. and Oney, S. (2019) How Data Scientists Use Computational Notebooks for Real-Time Collaboration, in *Proceedings of the ACM on Human-Computer Interaction*, ACM, New York, NY, USA.

Williams, L. and Kessler, R. (2003) *Pair Programming Illuminated*, Person Education, Inc.

Woodbury, R. (2010). *Elements of Parametric Design*, New York : Routledge.