Reverse Algorithmic Design of Buildings

Daniel Matos Fermoselle da Silva

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof. João Carlos Serrenho Dias Pereira

October 2019
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

First of all I would like to thank my supervisor Prof. Dr. António Menezes Leitão for suggesting me this project, but mostly because of his patience, advices, life knowledge and friendship. I also want to thank all my friends in the ADA group. I will not name everyone, otherwise this would turn into half a page of listing names, but they know who they are. Each one of them helped me in their own way, either with work-related tasks or in our relaxing, full of jokes and laughter moments.

Last but certainly not the least I want to thank my family. Including here my closest friends that have been with me since the beginning of my journey at Instituto Superior Técnico. Specially I want to thank João Marçal, José Semedo, and Tiago Rodrigues. The moments I spent with these three are priceless and helped me keep going. Either the leisure moments or the university-related ones helped me grow and learn. Finally, I would like to thank my parents and sister for their constant love, encouragement and caring over all these years, for always being there for me through thick and thin and without who I wouldn’t be half the person I am. I also want to thank my grandmothers, aunts, uncles and cousins for all the support and joy throughout all these years.

To each and every one of you – Thank you.
Abstract

Nowadays architects create almost all of their projects using digital modelling tools like Computer-Aided Design (CAD). However, this is not the best approach since every shape in the model is created manually. This makes it easier to introduce errors in the model. Over the years, research was done to automate the creation of models and ease their future reutilization and optimization. This led to the creation of Procedural Modelling (PM), which allows users to generate the desired models by creating descriptions that represent them, procedural models. The most used approaches for procedural models are grammars and their variations, however these present some problems, namely hard intelligibility, which makes them difficult to use and understand. A recent approach has been proposed to solve this problem: Algorithmic Design (AD), instead of grammars, uses algorithms as the procedural model. The problem thus becomes: how can existing models, manually created in a CAD tool, also take advantage of AD when no algorithm was created? The answer is Inverse Procedural Modelling (IPM), which consists in obtaining the procedural model from an already existing model. Once more, the research conducted in this area is mainly focused on grammars. In this dissertation, we propose Reverse Algorithmic Design (RAD), a specific methodology of IPM where algorithms are used as the procedural model extracted from existing models, allowing users to reuse and optimize those models. This is achievable by using bidirectional traceability between the model and the algorithmic description which can then be refactored with appropriate techniques, for improved intelligibility.

Keywords

Procedural Modelling; Inverse Procedural Modelling; Algorithmic Design; Traceability; Refactoring; Reverse Algorithmic Design.
Resumo

Atualmente os arquitetos criam a maioria dos seus projectos usando ferramentas de modelação digital como CAD. Contudo, esta não é a melhor escolha visto que toda a geometria tem de ser criada manualmente, facilitando a introdução de erros no modelo. Ao longo dos anos, investigações foram realizadas com o intuito de automatizar a criação de modelos, bem como facilitar a sua reutilização e optimização. Isto levou à criação de PM, que permite a geração dos modelos desejados através da criação de uma descrição que os representa, o modelo procedimental. As aproximações mais usadas para modelos procedimentais são gramáticas e suas variações, contudo estas apresentam alguns problemas, nomeadamente difícil inteligibilidade, tornando-as difíceis de usar e perceber. Recentemente uma aproximação foi proposta para resolver este problema, AD, que ao invés de usar gramáticas usa algoritmos como modelo procedimental. O problema resume-se então a: como é que modelos criados manualmente numa ferramenta de CAD podem também beneficiar das vantagens de AD quando não existe um algoritmo? A resposta é IPM, que consiste em extrair um modelo procedimental a partir de um modelo já existente. Novamente, pesquisas conduzidas na área focaram-se principalmente em gramáticas. Esta dissertação ambiciona ir para além dessa extracção. Propomos RAD, uma metodologia específica de IPM onde usamos algoritmos como o modelo procedimental extraído dos modelos digitais existentes, permitindo aos utilizadores reutilizá-los e optimizá-los. Isto é alcançável através de rastreabilidade bidirecional entre os modelos e a descrição algorítmica, que pode depois ser refactorizada com técnicas apropriadas, para melhor inteligibilidade.

Palavras Chave

Modelação Procedimental; Modelação Procedimental Inversa; Design Algorítmico; Rastreabilidade; Refactorização; Design Algorítmico Inverso.
# Contents

1 Introduction .................................................. 1
   1.1 Objectives ............................................... 4

2 Related Work .................................................. 7
   2.1 Procedural Modelling .................................... 9
   2.2 Inverse Procedural Modelling ......................... 14
   2.3 Algorithmic Design ..................................... 16
   2.4 Traceability .............................................. 19
   2.5 Refactoring ............................................. 21
   2.6 Summary .................................................. 23

3 Solution ......................................................... 25
   3.1 Methodology and Implementation ....................... 27
      3.1.1 Methodology ......................................... 27
      3.1.2 Traceability .......................................... 30
      3.1.3 Implementation ....................................... 33
   3.2 Features .................................................. 34
      3.2.1 Algorithm Extraction ............................... 34
      3.2.2 Algorithm Execution ................................. 34
      3.2.3 Traceability .......................................... 34
      3.2.4 Refactoring: Function Extraction ................. 40
      3.2.5 Refactoring: Free-Refactoring mode ............... 40
   3.3 Traceability Implementation Comparison ............... 40
   3.4 Workflow ................................................ 45
   3.5 Summary .................................................. 45

4 Evaluation ..................................................... 47
   4.1 User Testing ............................................. 49
5 Conclusion

5.1 Future Work
5.1.1 Improved refactoring methods
5.1.2 Improve refresh algorithm
5.1.3 Improve control flow order
5.1.4 Select control flow size
# List of Figures

2.1 Procedural Modelling representation. ........................................... 9
2.2 Parameter change mode. Original from [Yumer et al., 2015] .............. 12
2.3 Interpolation mode. Original from [Yumer et al., 2015] .................... 13
2.4 Inverse Procedural Modelling representation. .................................... 14
2.5 Castle with all the algorithm’s steps. Original from [Bokeloh et al., 2010] 15
2.6 Visual example of a grammar generated by Bokeloh et al.’s approach. Original from [Bokeloh et al., 2010] ........................................... 15
2.7 Rosetta’s architecture showing both front-end programming languages and back-end modelling tools. ................................................. 18
2.8 Single-script methodology. Original from [Castelo Branco and Leitão, 2017] 19
2.9 Traceability between the algorithm and the model in Luna Moth. Original from [Alfaiate, 2017] ................................................................. 20
2.10 Sketches within Integrated Development Environment (IDE). Original from [Leitão et al., 2014] ................................................................. 21
2.11 Traceability between the algorithm and the model. Original from [Leitão et al., 2014] ................................................................. 22

3.1 Usual AD methodology. The cycle goes from 1 through 4. .................... 28
3.2 RAD methodology. The cycle goes from 1 through 4. ......................... 29
3.3 Our implementation of the RAD methodology, Script-It. The cycle goes from 1 through 4. On the left we have the Atom programming environment, on the right the CAD tool AutoCAD and finally on top we have the AD tool Khepri. ......................... 30
3.4 Digital model and respective extracted algorithm. .................................. 31
3.5 Extraction of an algorithm being done by Script-It. ............................... 35
3.6 Execution of an algorithm being done by Script-It. ............................... 36
3.7 Traceability from the text editor to the digital modelling application. .......... 37
3.8 Reverse highlight feature from the digital modelling application to the text editor. 38
3.9 Traceability being applied to a compact and refactored algorithm. ............ 39
3.10 Function being extracted from pieces of an algorithm, after using reverse highlight to obtain the relevant code lines. 41
3.11 Function being extracted from pieces of an algorithm in free-refactoring mode. 42
3.12 Comparison of the execution of a raw algorithm with different traceability implementations. 43
3.13 Comparison of the execution of a refactored algorithm with different traceability implementations. 43
3.14 Comparison of the extraction of an algorithm with different traceability implementations. 44
3.15 Typical Script-It’s user workflow. 45

4.1 This figure shows both the model used for testing Script-It extraction and the extracted algorithm. 50
4.2 This figure shows both the model used for testing manual extraction and the extracted algorithm. 51
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Procedural models explored. Meaning: $\times$ - Poor; $\bigcirc$ - Medium; $\checkmark$ - Good</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Traceability options.</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Script-It’s test results. All the time measurements are displayed as MM:SS, where MM are the minutes and SS the seconds. The accuracy is measured in %.</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>Manual Extraction test results. All the time measurements are displayed as MM:SS, where MM are the minutes and SS the seconds. The accuracy is measured in %.</td>
<td>51</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------</td>
<td></td>
</tr>
<tr>
<td>IPM</td>
<td>Inverse Procedural Modelling</td>
<td></td>
</tr>
<tr>
<td>PM</td>
<td>Procedural Modelling</td>
<td></td>
</tr>
<tr>
<td>AD</td>
<td>Algorithmic Design</td>
<td></td>
</tr>
<tr>
<td>RAD</td>
<td>Reverse Algorithmic Design</td>
<td></td>
</tr>
<tr>
<td>CAD</td>
<td>Computer-Aided Design</td>
<td></td>
</tr>
<tr>
<td>BIM</td>
<td>Building Information Modelling</td>
<td></td>
</tr>
<tr>
<td>VPL</td>
<td>Visual Programming Language</td>
<td></td>
</tr>
<tr>
<td>TPL</td>
<td>Textual Programming Language</td>
<td></td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
<td></td>
</tr>
</tbody>
</table>
Introduction

Contents

1.1 Objectives ................................................................. 4
In the past, architects started planning their projects by sketching on paper, and only after the sketch was complete, with some degree of certainty, would the architect begin the drawing phase with the technical representation of the project, also on paper. However, with the advent of computers, Computer-Aided Design (CAD) tools started to appear [Kalay, 2004]. The first versions of these tools were still two-dimensional (2D), but quickly evolved to a three-dimensional (3D) representation. After that, Building Information Modelling (BIM) tools were introduced, as well as tools for lighting, structural, and energy analysis, among others, which could be integrated with existing design tools. The models produced with computer-aided tools are known as digital models.

Digital models helped architects with their work, as they allowed, for instance, greater accuracy, which was harder to achieve with pen and paper. Furthermore, it is easier to modify a digital model than paper-based technical documents.

Even though digital modelling tools smoothed the process, the creation of digital models was still a gruelling task, since all the elements had to be created manually in a repeated manner. In order to improve the model creation process and make it more autonomous, Procedural Modelling (PM) tools were created. These allowed not only the architect to design their models faster but also to explore more variations without spending much time generating those. However, the most common PM tools use design grammars, which are not easy to work with [Müller et al., 2006, Correia, 2013] and, because of that, new approaches started to be investigated, such as, Algorithmic Design (AD). AD allows the architect to produce digital models by creating an algorithm that generates them [Terzidis, 2006]. The use of AD has several advantages for the design process, since it allows an easier exploration of design alternatives by changing the parameters of the algorithm. This not only gives architects more time for their creative process, but also simplifies their work, especially when optimizing a digital model, removing the burden of changes that had to be done manually in the past.

AD is still a recent design method, and since it requires the use of programming techniques, it presents a barrier for architects who lack programming skills. This barrier can become thinner with visual AD tools, which are already available, as these tools allow users to create models by simply connecting multiple algorithms through wires. Nevertheless, when they are used for a complex project, the algorithmic description can become difficult to understand. However, these tools can be a starting point for allowing architects to correlate the algorithms directly with the produced model for a better understanding of it. Therefore, due to the difficulties that are introduced by adopting an algorithmic approach, the majority of architects do not want to start designing their models using AD.

In order to help architects reuse and optimize older digital models, which did not have a procedural model representation, a new technique started to be considered: Inverse Procedural Modelling (IPM). IPM allows the architect to obtain the procedural model of an already existing digital model and reuse it, either to test new versions of the same model or to optimize it. The problem with this approach, however,
is that the usually chosen PM representation, grammars, suffers from hard intelligibility among others. This thesis focus in overcoming some of these problems encouraging and making it easier to learn, use, modify and understand the recent representation, algorithms.

To accomplish our goal, several areas must be explored, namely:

- PM: Generating a digital model from a description, the procedural model;
- IPM: Extracting the procedural model from an existing digital model;
- AD: A specific use of PM, consisting in creating a digital model using an algorithm;
- Traceability: Creating a connection between code and other elements. For instance, connect a piece of code with its respective documentation or, for the purpose of this project, connect a set of shapes to specific pieces of code;
- Refactoring: Changing a software without damaging its behaviour, yet improving its structure.

In this dissertation, a new methodology, implemented as a tool, is presented. It aims to allow its users to extract an algorithm from a digital model, particularly, a non-algorithmically generated one. The extraction tool will also have traceability features that establish a connection between the generated algorithm and the different parts of the digital model. This capacity will help users better understand the algorithm’s behaviour, and thus modify it to their content. Furthermore, it also allows users to generate new traceable digital models from existing algorithms. Since the resulting algorithm is readable, parametric, and editable, it can not only generate the original model, but many other variations of it. This makes it possible for the user to change and reuse this model in the future.

1.1 Objectives

The main goal of this thesis is to introduce a methodology for IPM of architectural projects using algorithms. This methodology will be implemented as a tool that can extract the generator algorithm of a digital model, as well as help the user to refactor it. The tool will allow its users to reuse digital models with an AD approach, without having to write the whole algorithm manually, thus being faster and less error prone. Refactoring and traceability methods will help users to interactively produce a cleaner and structured algorithm, which can then be modified to create and explore more variations of the model. Finally, it is expected that both the methodology and the tool can be used by architects with just a modicum of programming knowledge, thus, improving the modelling workflow of different architectural practices.

The remainder of this document is structured as follows: Chapter 2 contains related work in the area of PM, IPM, AD, Traceability, and Refactoring; Chapter 3 presents the solution to address the identified
problems; Chapter 4 includes an evaluation of this thesis; Finally, Chapter 5 contains the conclusion and a discussion of how this project will ease the work of architects.
# Related Work

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Procedural Modelling</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Inverse Procedural Modelling</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Algorithmic Design</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Traceability</td>
<td>19</td>
</tr>
<tr>
<td>2.5 Refactoring</td>
<td>21</td>
</tr>
<tr>
<td>2.6 Summary</td>
<td>23</td>
</tr>
</tbody>
</table>
In this section, previously developed work and tools related to this project will be presented.

To better understand this section, it is important to explain what a grammar is. A simple grammar is represented as $G = \{S, \Sigma, N, R\}$, where $S$ is a special set of non-terminal symbols, the starting symbols; $\Sigma$ is the set of terminal symbols, to which no rules can be applied; $N$ is the set of non-terminal symbols that can be transformed into other terminal and/or non-terminal symbols; Finally, $R$ is the set of existing rules that can be applied to the non-terminal symbols. Many variations of this simple grammar were created to allow a better representation for each circumstance. For instance, there are grammars, such as the Shape Grammars [Stiny, 1975], where the symbols are represented as shapes with variable complexity.

### 2.1 Procedural Modelling

Nowadays there is an increasing demand for computational models, each time with increasing detail. Although some industries want these models to be produced quickly, they do not want them to be expensive. PM is a good approach to solve these problems.

There is a considerable amount of research on PM and despite presenting different definitions of PM, they tend to agree that PM consists in generating content (digital models) by writing and interpreting a model description written in a procedure or program [Beneš et al., 2011], as shown in Figure 2.1. This description is the key challenge of PM and it is also known as procedural model or procedural description, which consists on a set of rules that represent the digital model [Št'ava et al., 2010]. In general, it is easier to create and modify these rules than building a model by hand in a digital design application. Nevertheless, the creation of rules is not an easy task. Nowadays the most common representation for procedural models is grammars, even though they can also be represented by algorithms.

![Procedural Modelling representation.](image)

Currently, there are many variations of design grammars, i.e., grammars that generate designs. Many of these grammar variations descend from a parallel string rewriting system known as Lindenmayer systems or L-Systems [Lindenmayer, 1968, Prusinkiewicz and Lindenmayer, 1990]. This type of grammar works only on strings and, as such, every derivation consists on replacing the previous string with the result of applying a rule to that string. L-systems are a very good approach to model plants, since they can easily simulate an open-space generation, but they are not as appropriate when modelling buildings.
since these do not have a growth-like process as plants do [Wonka et al., 2003].

An approach which differs from the L-systems is the shape grammar introduced by Stiny [Stiny, 1975]. This grammar presented the novelty of operating with the concept of shape, rather than working with strings. With these grammars, the rule derivation is usually done manually or by using a computer while a human decides which rules to apply. This leads to a lack of automatism, which does not allow the fast modelling of larger areas. The derivation process is usually slow due to emergence, which is the ability to recognize and use subshapes, i.e., parts of a shape, in the derivation process [Duarte, 2001, Garcia and Leitão, 2018], thus also slowing down the generation of the models. Emergence also makes the rule derivation process non deterministic. However, this ambiguous derivation is considered by some as responsible for the production of models with unexpected differences [Knight, 2003, Garcia and Leitão, 2018]. Another approach Stiny explored was Set Grammars [Stiny, 1980, Wonka et al., 2003], which treat shapes as symbolic objects. This removed the shape grammar’s sub-shape matching procedures, thus making the grammar more automatic and computationally easier to process.

In order to achieve the best possible automatism many papers with new types of grammars were published. Split grammars [Wonka et al., 2003] are a descendent of Set Grammars where the main novelty is the restrictions that can be associated with the rules, in order to better specify when to use them in each step of the derivation. This grammar allows for the use of two types of rules only: the split rules and the conversion rules. The split rules, as the name suggests, split a shape in a number of elements that fill the space occupied by the original shape. The conversion rules convert shapes into other shapes which must fit in the space previously occupied by the original shape but do not have to fill it completely. These two rules will be used by other grammar variations. Even though this new grammar was a great step forward it still has some downsides. Due to the expected complexity of architectural models it is not easy to modify the grammar rules without previous extensive knowledge of Split Grammars. The authors expected a future where a Split Grammar would be created as a joint work between a designer and an architect and, when other users wanted to modify the model, they would just change the grammar attributes.

Another explored approach was to extend the already existing L-Systems in order to take advantage of its capabilities, while overcoming its down-sides. This approach was embodied by the first version of CityEngine [Parish and Müller, 2001], a system for modelling cities. One of the biggest disadvantages of L-Systems is the addition of constraints, since every time the user wants to add a constraint to the system several rules have to be added. A major goal the CityEngine authors wanted to achieve was to be able to add multiple modules to a city model and, as such, they had to create a method to overcome the addition of constraints in the L-Systems. To that end, they proposed a generic template of an L-System module that allows the setting of parameters in the L-System to be postponed and ported to external functions. With this methodology, instead of trying to write all the constraints in the L-System
beforehand, users just write a generic template and only when a rule is applied will the global and local constraints be checked to understand when and where to apply the rules. These two types of constraints are defined in external functions. However, the explored tool still uses L-Systems, which are better at simulating growth in open spaces. This is not the case when modelling buildings where stricter spatial restrictions are imposed. These restrictions make it difficult to adapt L-Systems to the modelling of buildings [Wonka et al., 2003].

To solve the problem, the authors of the two papers referred above [Wonka et al., 2003, Parish and Müller, 2001] cooperated and created the CGA shape [Müller et al., 2006], a new extension of the Split Grammars previously introduced [Wonka et al., 2003]. As an addition to Split Grammar, CGA shape contributed to the definition of split, repeat, and scale rules. One advantage that this Split Grammar has over usual L-Systems, when modelling buildings, is rules being applied sequentially and not parallel, as in L-Systems. Parallel grammars are better for natural growth, but sequential ones are better for structure characterization, which is the case of buildings [Prusinkiewicz et al., 2001, Müller et al., 2006]. This new grammar was later integrated in CityEngine along with many shape rules that were not present on L-Systems. The ideas behind this new grammar were pertinent and the authors wanted to create an easy-to-read grammar with good controllability. However, carelessly written rules will be hard to understand by anyone except the creator of the grammar [Müller et al., 2006]. This lack of understandability makes it a difficult task for new users who want to start modifying previously written grammars. As a final note, the authors of the CGA shape claim that it has a learning curve similar to scripting languages.

One major problem of grammars is that they are generally hard to read and edit. Most of the times the only person that can comprehend the grammar is its creator. That is because grammars are hard to understand and do not scale well with the complexity of the model. Several researchers tried to overcome this problem, for example, using a variation of L-Systems for each part of a model, which are known as “guides” [Beneš et al., 2011]. Each system inside a guide can communicate with the others through connections, known as “links”. This separation, in a divide-to-conquer style, allows the user to focus either on the global view of the model or on a specific guide. With this approach the model will be divided and will be easier to control. However, since the procedural models used are still L-Systems, they still need to be written manually. In addition, this approach has scalability problems since it requires the addition of links for each pair of guides that need to communicate with each other.

Although grammars are the most used PM approach, there are others with more appealing interfaces and which are easier for a non-expert user who is starting to work with PM. Moreover, even with an already existing grammar, due to its large number of rules, it becomes hard to control it and thus difficult to explore new models. A new approach is, for instance, the visual interface presented in [Yumer et al., 2015], which allows the intuitive exploration of PM spaces in real time using autoencoder neural networks. Usually, a grammar has a lot of parameters scattered around, which makes it difficult to change
all of them in order to explore new models. This approach applies a dimensionality reduction in order to decrease the number of parameters available to be explored. The reduction is applied through an autoencoder which learns what is the best dimensional space to work on, fusing or eliminating parameters if needed. The best contribution of this approach is the explore-and-select interaction, where the user is able to explore models by indirectly modifying the parameters with an arrow interface as shown in Figure 2.2, choosing the best model in the end. Furthermore, users can save some models and use them in the dragging approach where the new model being built is in the middle of the saved models and will be created as an interpolation of the saved models, as we can see in Figure 2.3. A study with users showed that users preferred the visual interface and can produce the models faster when compared to a conventional PM system. However, this interface has a disadvantage related with scalability, since the more complex the original design space is, the harder it is to reduce its dimensionality. Doing so would imply the loss of parameters, thus, some shapes would no longer be possible to create.

![Figure 2.2: Parameter change mode. Original from [Yumer et al., 2015]](image)

After analysing several PM approaches, we reached some conclusions. First, with PM, models are tested much faster, allowing designers to better explore their creativity. Second, almost all referred projects opted for grammar approaches, which present disadvantages when compared to other available approaches, as we will see in the following sections. Finally, an ideal approach would imply an interface that allows users to adapt the procedural model to their liking, as well as observe the connec-
tions between the procedural model and the digital model. This ideal procedural model must overcome grammars’ disadvantages by offering users better control of their models. This control intends to ease the editing task, as well as the readability task and, if necessary, refactor the model in order to adapt it to a new user.

PM in general seems to be a good approach, however it also has some disadvantages, namely the need to manually write the procedural model, which is not an easy task. Another difficulty is not being able to see the digital model changing in real time as changes are applied, as in a digital modelling application, since it will only be generated after the procedural model is interpreted. Some newer tools are trying to overcome these disadvantages, for instance the one proposed by [Yumer et al., 2015] which offers a real-time visualization of the model.

To close this sub-chapter it is important to state a few things. First, almost all the grammars found in PM literature are mostly only tested by its creators and have not been used in a real design scenario, with a few exceptions such as CityEngine [Garcia and Leitão, 2018]. Second, the usual application of shape grammars can cause the derivation of an enormous number of shapes due to the emergence ability, which can hinder good results [Garcia and Leitão, 2018, Correia, 2013]. Also, even though some people defend the production of unexpected models, rule ambiguity is not a good property to be added to a computer application. Currently, computers, unlike humans, cannot understand incomplete or ambiguous rules and therefore are not able to infer what should be the correct rule to apply [Correia, 2013].
This makes the creation of computer-based shape grammar applications very difficult.

2.2 Inverse Procedural Modelling

IPM has been a known problem for many years. As the name states, IPM consists of doing the inverse of PM, in other words, extracting the procedural description from a digital model given as input. Basically, with IPM, users want to obtain the set of rules which can generate the digital model they give as input. This definition is represented in Figure 2.4.

With IPM, a designer can reuse previous models which did not have an associated procedural representation. Optimally, this procedural model will allow the user to change parameters of the model, which would be harder without the procedural representation since the model would have to be manually recreated in a digital modelling application.

Most of the related work we explored focused on extracting grammars with IPM, since the most used procedural descriptions are grammars. However, not all authors used the same types of grammars. The types of grammars used include L-Systems [Šťava et al., 2010], Split Grammars [Wu et al., 2013], and Shape Grammars [Bokeloh et al., 2010]. Although not all authors mentioned above chose the same representation, they all agreed that the key factors for an easier IPM are repetition and symmetry.

As pointed out by [Šťava et al., 2010], models are usually a composition of repeated elements, a phenomenon commonly found in nature as well. Another thing that is recurrent, both in nature and in models, is symmetry. If we look at a plant’s leaf it tends to be symmetric and the same happens with most buildings. Because of this notion, many researchers have been studying how to use symmetry and repetitions for IPM purposes. Some applied grouping methods with clustering knowledge to create rules based on similarity and symmetry. In [Šťava et al., 2010], similar elements are grouped and, iteratively with an importance function, from the most to the least important a cluster is chosen to be transformed into a rule. However, this methodology uses L-Systems which are not the best approach to model buildings, as we showed previously. Also, at the moment, the strategy they presented only works for 2D scenes, which is not what we aim for, even though they want to extend the system to allow 3D structures.

Another approach that also intends to use symmetry for IPM and that already works with 3D models

Figure 2.4: Inverse Procedural Modelling representation.
is explored in [Bokeloh et al., 2010]. Here, the main objective is to find similarities in the input model, that will be used as an example to produce a shape grammar capable of reproducing the test model and add some new elements, while still preserving its similarity. As shown in Figure 2.5, the main focus of this approach is finding symmetric elements in the model, which are separated by some non-symmetric areas. These can then be replaced by other elements without losing the similarity of the model. In the end, the grammar can be viewed, as shown in Figure 2.6, with multiple elements, which fit in specific areas without breaking the symmetry of the original model.

At the moment, the major disadvantage of this approach is the fact that it only works with models that are almost exactly symmetric and similar. Also, since it uses Shape Grammars as the procedural model, it will suffer from the previously discussed issues.

![Figure 2.5: Castle with all the algorithm’s steps. Original from [Bokeloh et al., 2010]](image)

![Figure 2.6: Visual example of a grammar generated by Bokeloh et al.’s approach. Original from [Bokeloh et al., 2010]](image)

The last approach we will discuss consists in generating the procedural model, in this case, a Split Grammar, which represents the facade layout given as input [Wu et al., 2013]. In this research, the Occam’s razor is an important reference, and is used as a metric for the test results. With this in mind, the smaller and more general grammars are rated best. In order to compare grammars, the authors created cost functions, which would check the best grammar they could produce with their algorithm.
To build this grammar they used an iterative algorithm, which at each step produces a deterministic grammar. In the end the chosen grammar will be the one with the lesser cost. For every iteration of the algorithm, an element of the model will be split and a rule representing that action is created. This will be repeated until nothing more can be split. To minimize the number of rules, they use the repetitions and symmetries of the layout, thus, identical elements can be derived with the same rules. This approach was one of the best we found and was confirmed with the test they reported, where the quality of the grammars produced with this algorithm were only behind the ones built manually by expert users. However, similarly to all the work presented in this area, this approach only works for facades with almost identical element sizes. Furthermore, since they use Split Grammars as the procedural description, this approach suffers from the same disadvantages we referred previously.

In summary, symmetry, similarity, repetitions, and patterns are the important keywords, as they make the process of extracting rules easier. After having created all the rules for a specific model, the output will usually be a grammar. Finally, with this grammar, which is the most used procedural model, the user can modify some parts of it in order to vary the initial model. However, as we explained in the previous section, grammars have several problems and, because of that, we will explore different approaches for the development of procedural models, which will be presented in the following section.

### 2.3 Algorithmic Design

AD is an approach that uses algorithms in order to create digital models [Terzidis, 2006], which can be considered as a specific use of PM. Instead of the common PM, using shape grammars or similar approaches, AD allows users to take advantage of an algorithm that will be executed in order to produce a model in a digital modelling application. To create the algorithm, users can opt for one of two alternatives: Visual Programming Languages (VPLs) or Textual Programming Languages (TPLs). The first alternative allows users to build an algorithm more intuitively and with little programming knowledge, since it can be built by simply connecting components through wires. As a result, VPLs are the best option for beginners who want to start using AD. An example of a VPL is Grasshopper 3D. Although VPLs are a great option for beginners, they do not scale easily, as for more complex projects the algorithms will be composed by large numbers of components and wires, which are difficult to analyse and manipulate. The second option, TPLs, are more demanding in the beginning, since they require the user to memorize more than a simple set of components. With this option, users require more time to write the whole algorithm, since it is more difficult to study a TPL and find the necessary components than a VPL, where instead the user just drags and connects the visual and more appealing components. TPLs require users to have more extensive programming knowledge, and because of that, this alternative is suitable for more experienced programmers. However, for beginners, VPLs offer a more intuitive approach to algorithmic design.
not the most used AD option by beginners. However, due to the textual nature of TPLs, they not only scale better with the complexity of the wanted model but also allow finer control of the program, when compared to VPLs. TPLs also allow for code reutilization, refactoring, and debugging with more ease than VPLs. Therefore, in the long run, TPLs seem to be a better trade-off than VPLs.

Due to its algorithmic nature, AD allows the user to have better control of the produced model. Although it is not easy to build an entire model from scratch with an algorithmic approach, it is a good compromise, since after the first model is created many variations can be done just by modifying parameters in the algorithm [Leitão et al., 2013].

However, current AD approaches have some disadvantages, namely what is known as vendor lock-in [Lopes and Leitão, 2011]. Even though nowadays there are CAD and BIM applications that allow the use of programming languages for automation and AD, they are specific for each tool, hence, limiting program portability. For instance, if we are using the programming language of Rhinoceros 3D, RhinoScript, and for some reason we want to use this model on another CAD application, it will not work unless we rewrite the whole program in that CAD programming language. Rosetta [Lopes and Leitão, 2011], a portable AD tool, overcomes this difficulty by providing a hub with multiple front-ends (programming languages) and back-ends (digital modelling applications), as shown in Figure 2.7. With Rosetta, designers can choose which programming language they want to use and in which application they want to generate the resulting model. Also, if needed, designers can use Rosetta to analyse the model described in the algorithm with analysis tools like Radiance\(^2\) or Robot\(^3\).

Moreover, Rosetta can also solve other issues present in current architecture workflow. Nowadays, architects often use multiple software, such as CAD, BIM, and analysis tools to build high quality models. However, using multiple software often creates portability issues, since conversion mechanisms between tools tend to cause problems [Castelo Branco and Leitão, 2017, Feist et al., 2016]. A recent research [Castelo Branco and Leitão, 2017] shows the advantages of using Rosetta and AD in order to decrease the endeavour of using multiple modelling software. It presents a methodology (depicted in Figure 2.8), which is composed of 3 stages. Firstly, for the initial exploration, a CAD model is built. After that, for more detail, a BIM model is created. Finally, with this approach, analysis can be done within any of these two steps. The simultaneous use of these tools is possible while using only a single script developed with Rosetta, since the tool is capable of adapting the script to the requirements of each software. Architects using this approach have an eased design process, since they do not have to spend time on conversions or suffer from the issues these conversions tend to introduce.

One disadvantage of the AD approach is the difficulty users have in understanding the relation between algorithm and model. While using modelling tools, users can immediately see the results of their work. Unfortunately, this does not happen with usual AD approaches. In the common approach,
users have to execute the algorithm, in order to see the impact of the changes made in the model. Furthermore, with AD sometimes the designer might not know what a specific element of the algorithm does to the model. These two drawbacks can be overcome with two features present in tools like Luna Moth [Alfaiate, 2017]: traceability and immediate feedback.

Luna Moth offers traceability between the algorithm and the generated model, as well as quick feedback when changes are made to the algorithm. Traceability allows the user to know how some specific element of the model is represented in the algorithmic description, and how some element of the algorithmic description is represented in the model. This feature helps Luna Moth's users understand the implications the changes they introduce in the algorithm have in the model (as shown in Figure 2.9). Traceability also helps minimize the probability of introducing errors in the model description.

In conclusion, AD is a recent approach to represent digital models, which overcomes many of the limitations presented by other means of representation. Although it still presents some disadvantages, current research shows multiple AD tools being developed in an attempt to mitigate its shortcomings, as well as ways to better introduce the algorithmic approach in the user’s workflow. Two of the capabilities which can be used to minimize the disadvantages of AD are traceability and refactoring.
2.4 Traceability

The broader definition of the word traceability is a relation between two parts. This is important for our thesis since we aim to allow users to understand what each part of the algorithm represents in the model and the opposite. Without this feature users are not able to transform the extracted algorithm into an intelligible one, thus they will not easily take advantage of an AD approach. In a similar context, and according to [Leitão et al., 2014], traceability consists in creating a relation between program and model. This connection is very important to allow program comprehension, which is defined by Rugaber [Rugaber, 1995, Leitão et al., 2014] as “the process of acquiring knowledge about a program”. Program comprehension allows for easier maintenance, debugging and refactoring. With these, users can understand their program and more easily modify it if needed.

In the area of AD, Illustrated Programming [Leitão et al., 2014] is a methodology that allows users, most specifically architects, to connect algorithms with sketches and digital models. Users are able to compare the written algorithms with sketches embedded within the Integrated Development Environment (IDE) (shown in Figure 2.10), as well as, trace the control flow of the program that generates the model.
However, in order to achieve this level of traceability we must store the entire control flow, which leads to considerable overheads.

Another interesting example is the already mentioned Luna Moth [Alfaiate, 2017], which creates a traceability relationship between the code and the shapes it produces. This allows users to better understand how their coded algorithm is related with the produced model, as well as what parts of the program they must change to achieve their goals.

In conclusion, it is important to allow users to have a connection between pieces of code and the shapes it produces, thus achieving program comprehension. Despite the proven overheads, we consider that the ability to visualize the complete program flow is useful. Hence, it might be relevant to allow users, when possible, to select which degree of traceability depth they want, i.e., how complete they want the traceability to be.

Figure 2.9: Traceability between the algorithm and the model in Luna Moth. Original from [Alfaiate, 2017] (shown in Figure 2.11).
2.5 Refactoring

Refactoring, according to Martin Fowler and Kent Beck [Fowler et al., 1999], is defined as a process that changes the software, improving its internal structure. However, these changes should not modify the external behaviour of the already existing code. Also, according to the same authors, refactoring is a well-organized method to clean up code and minimize the insertion of bugs. Refactoring is an important practice that can improve readability and future maintenance. Nowadays, many IDEs have integrated refactoring tools that allow the user to easily refactor the code. There are many refactoring techniques: from simply renaming a function everywhere it appears in the code, to convert a considerable amount of conditional statements (for instance many if statements) into a polymorphism approach.

Refactoring is important for our thesis since we aim to allow users to refactor the algorithm extracted
from the model. Algorithms automatically extracted from digital models are quite low level, being only composed by primitive function calls, therefore, not being intelligible for users to understand. Therefore, refactoring will be of great importance in order to introduce semantics in the algorithm. For instance, users can select a part of an extracted algorithm and create a function which represents a wall.

Throughout the years, refactoring tools evolved and some research has been done [Murphy-Hill et al., 2012] in order to understand which refactoring techniques are most used, how tools should be built to ease the users’ tasks, and which type of users use refactoring the most. In [Murphy-Hill et al., 2012], eight datasets from different types of users are analysed. These datasets vary from common users to tool creators (toolsmiths). This analysis provide relevant conclusions to who intend to build a refactoring tool. The most important ones are listed below:

- The most used refactoring tool is the renaming tool. This goes against what the refactoring toolsmiths thought, which means it is important to improve underused tools and their documentation in order to increase their usage;

- The use of refactoring tools which do not require initial input has been increasing. However, they do not fit all scenarios. Hence, it is important to make smarter tools that accept initial input but have some kind of intelligent default state that allows users to skip this step in the most common situations;

- Tools that allow the programmer to refactor multiple elements at a time are preferable;
• Tools must be built in order to allow three factors to be achieved by the user: (Awareness) the user should know the tool exists; (Opportunity) the user should know when to use the tool; (Trust) the user should trust that the tool will not damage existing code.

The conclusions mentioned above will guide programmers who want to build a good refactoring tool, by helping them focus on the most important aspects of a tool besides refactoring itself.

2.6 Summary

With this research, it is possible to conclude that PM has its advantages when compared with modelling manually in a digital modelling application, such as a more efficient model variation comparison. However, building a procedural model by hand takes a lot of effort and a lot of time. To surpass this difficulty, IPM was introduced, providing a way of creating the procedural model automatically from an existing digital model. This allows users to reuse older models by modifying the needed parameters in order to optimize the model or produce a new one.

There are several representations for procedural models, such as grammars, which are the most used and investigated. However, using grammars as a procedural model has some disadvantages that make them less practical to use, such as its hard intelligibility, and the emergence capability, i.e., the capability that allows rules to be applied to parts of shapes, leading to a combinatorial explosion of rule application, which in turn leads to a combinatorial explosion of shapes [Garcia and Leitão, 2018]. To solve some of the problems introduced by grammars, new representations were created, namely algorithms, which are now starting to be used in the AD approach. Yet, AD also has some points to improve, namely the learning curve, for which traceability and refactoring can provide considerable help.

In table 2.1 the most explored procedural models are compared in three usability metrics, chosen in accordance with this dissertation objectives. Understandability is important since we want users to be able to modify the model made using our approach easily. Thus, an understandable procedural model is one where it is easy to read and decipher existing code, as well as to write new code, or modify existing parts. Scalability and learning curve are appropriate measures since we want users to build more complex models without much effort or difficulty, as well as allow them to learn the new approach easily and as fast as possible.

In table 2.1 we can observe the evolution of procedural models over the years. It started with L-Systems, which lack in almost all the metrics but offer a good solution for plant modelling. Then, Stiny’s Shape and Set Grammars, which improved upon the understandability. Wonka’s Split Grammars brought the restriction novelty, which also allowed for better understandability. The last grammar variation is the CGA Shape Grammar, the base grammar in the CityEngine, which is being used commercially and allowed a better scalability while diminishing the learning curve. To finish table 2.1 we have AD which
improved in almost every aspect lacking only on the learning curve, a metric we aim to improve with our methodology, using traceability and refactoring capabilities. We explored these two last areas and understood that traceability will allow users to have a better knowledge of the connection between the digital model and procedural model, thus ease the creation of variations. We also learnt important guidelines which will lead us on the creation of refactoring tools.

<table>
<thead>
<tr>
<th></th>
<th>L-Systems</th>
<th>Shape Grammars</th>
<th>Set Grammars</th>
<th>Split Grammars</th>
<th>CGA Shape Grammars</th>
<th>Algorithmic Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understandability</td>
<td>×</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>√</td>
</tr>
<tr>
<td>Scalability</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>○</td>
<td>√</td>
</tr>
<tr>
<td>Learning Curve</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

Table 2.1: Procedural models explored. Meaning: × - Poor; ○ - Medium; √ - Good
3

Solution

Contents

3.1 Methodology and Implementation ............................. 27
3.2 Features ......................................................... 34
3.3 Traceability Implementation Comparison ..................... 40
3.4 Workflow ......................................................... 45
3.5 Summary ......................................................... 45
To start this chapter, we recap the existing problem and we describe the purpose of this dissertation. The majority of the existing digital models are created with manual digital modelling applications like CADs and BIMs. This is not very efficient and, in fact, some new methods like AD are better. However, rebuilding a model with AD is also not efficient, as we would have to create it from scratch. To solve this problem, we propose to ease the conversion of existent non-algorithmic digital models into algorithmic ones. This is similar to some of the work we explored in the previous section, namely, the IPM methodology, where existent models were converted to procedural models. However, since we are working with algorithms, just as we call AD the specific use of PM with algorithms, we will introduce Reverse Algorithmic Design (RAD), the specific generation of algorithms in an IPM methodology. RAD consists of obtaining the algorithm that generates a digital model given as input. To help with this task, a new methodology was created, based on the concept of traceability. It consists of creating a bidirectional traceability mechanism between the algorithm and the digital model. This feature is intended for helping the program comprehension task that will then allow us to improve the extracted algorithm by using refactoring operations. In this chapter, we present our methodology and a tool which was created to support it, Script-It.

3.1 Methodology and Implementation

3.1.1 Methodology

As explained in chapter 2, current AD methodologies consist in using algorithms to create digital models, as we can see in Figure 3.1.

The goal of RAD is essentially to invert the AD workflow, as shown in Figure 3.2. With this new methodology, users are able to obtain the algorithm which represents the digital model they created in a digital modelling tool. This algorithm is generated with meta-programming techniques, by which a program is written by another program. The process starts by observing the model and acquiring the necessary parameters for each observed shape. Then, an equivalent program is generated that reproduces the model, for instance, when given a circle with radius \( r = 1 \) and centre in the 2D coordinates \( x = 0 \) and \( y = 0 \), the top-level primitive function generated will be: \( \text{circle}(xy(0.0, 0.0), 1.0) \).

As we can see in Figure 3.2, to implement this process we need some additional tools besides our RAD tool. We require:

- An AD tool which is capable of extracting, through meta-programming techniques, a base algorithm from a digital modelling application, i.e., an algorithm composed by simple primitive function calls. It is also necessary for this tool to be able to generate the digital model obtained from an algorithm given as input;
• A programming environment which is capable of running user-made packages. More specifically this programming environment will be modified to allow the **refactoring** of the algorithm, the connection with the AD tool, and the bidirectional **traceability** between the algorithm and the digital model;

• A digital modelling tool to contain and present the model that will be transformed into an algorithm. This tool will also participate in the bidirectional traceability, highlighting specific shapes when needed.

There are multiple possible choices for each of these tools. For the purposes of this thesis, we will use Khepri, the newest version of Rosetta [Lopes and Leitão, 2011], for the AD tool, since it is an already explored tool in the AD area, and it is still evolving, allowing our implementation to evolve alongside it. Khepri already allows users to obtain a meta programmed version of an algorithm that generates the digital model given as input. However, the extracted algorithm will be consisted only by
primitive function calls, therefore will have low readability, will be harder to reason about, modify, and has no traceability features associated. With our methodology, we want to be able to overcome these drawbacks. For the programming environment, we will use Atom\(^1\), since it is one of the most recent text editors, almost fully programmable and capable of running user-made packages. Moreover, since Khepri is, at the moment, optimized for the Julia programming language, this was also the programming language used to implement Script-It. Atom was also based on this choice, since Juno\(^2\) is one of the most used Julia IDEs'. Finally, for the digital modelling tool, we will use the known digital modelling application AutoDesk's AutoCAD\(^3\).

When we apply our choices for each required tool, we will obtain our implementation of the methodology, Script-It, which will be depicted as in Figure 3.3.

---

\(^1\) Atom is an open source text editor created by GitHub with support for user-made packages. Atom's web page: https://atom.io/

\(^2\) Juno is an Atom package. Juno's web page: https://junolab.org/

\(^3\) AutoCAD's website: https://www.autodesk.com/products/autocad/overview
3.1.2 Traceability

At the moment, Khepri is already able to extract a simple version of an algorithm from a digital model. However, this algorithm is not intelligible, as we can see in Figure 3.4. In order to improve its readability and usability, refactoring methods should be applied. Nonetheless, it is not an easy task, to say the least, to apply refactoring when it is not possible to understand what each part of the algorithm represents. Therefore, it is important to create a mechanism which allows us to establish a bidirectional connection between the model and the algorithm. This capability is known as traceability, without it, users would not be able to use the extracted algorithm, thus they would not take advantage of the AD approach.

After establishing the connections between Atom and Khepri, for extraction and execution of algorithms, we need to make it easier to extract an intelligible algorithm from an already existent digital model. For this, it is necessary to have two important capabilities: (1) traceability, (2) refactoring.

We explored multiple traceability approaches in order to find the option which would fit our require-
ments. For that we had to take into account all the trade-offs for each option.

Each programming language has different introspection capacities (the ability of a program to examine its own structure and behaviour), consequently there are different approaches to implement traceability. We studied and explored four different approaches to achieve traceability:

- **Code Redefinition**;
- **Code Injection**;
- **Interpreter**;
- **Stack Inspection**;

**Code Redefinition** consists in redefining all the necessary parts of the code in order to get the traceability needs fulfilled. An example for this approach could be Python. We can use Python to go through a module and encapsulate each function inside a new one, which will execute the necessary traceability code before executing the rest of the original function.

**Code Injection**, as the names states, consists in injecting snippets of code around the existing code in order to obtain the necessary information. An example of this option is Racket with hooks. These hooks are special functions, which by default do nothing, however, they are always executed at a specific time of the process, for instance when a process starts. We can redefine these hook functions and inject with them the necessary instructions to retrieve information to allow the wanted traceability
features. Another example is Java with Javassist library\(^4\). This library allows the manipulation of the Java bytecode at load time. Therefore, we could use Javassist to inject the needed instructions to obtain the information for traceability in the methods of a specific class.

**Interpreter** is very different from the two options above, since there is no code modification or injection. With this option, an interpreter is used in order to interpret the code. When, within the interpretation process, we detect a specific function producing a particular result in which we are interested in, we store information regarding that to allow future traceability. One of the advantages this option provides is not requiring more than read-only access to the code. A downside of this approach is usually having slower execution times since the code is being interpreted. An example of this option is Julia with the JuliaInterpreter package.

**Stack Inspection** consists in obtaining the control flow and locations (files and code lines) needed to provide traceability features from inspecting the function call stack. With this option, when specific functions are called, the function stack is inspected. An example of this option is Julia, since it allows the inspection of the stack fairly easily, and Java, which also allows us to inspect its stack with ease. An advantage of this approach is not requiring external libraries to retrieve the algorithm’s control flow. A disadvantage is being sensible to compiler optimizations, for instance function inlining, which avoids creating call frames for certain functions, thus making it impossible to detect that they were called.

Given the constraints of the tools we were using, namely, the Julia language, we decided to focus on the two most promising approaches, that were using an **Interpreter** or **Stack Inspection**. Having these options, we were able to test both and compare them, in order to understand which one is the best for our goal. Even though these two traceability options differ, for their implementation there are some elements that will be the same or similar.

When building these two options, there were some problems introduced by the current package in the **Interpreter** approach, and we also opted to also test some differences in implementations. For the **Interpreter** approach with JuliaInterpreter, at the moment, it is only possible to start the interpretation with an entry point function, within which the whole algorithm will be written, just like the main function in C or Java. In the **Interpreter** option, we also opted to save only the last shape returned by a call. Later, we realized that it would be better to highlight the whole control flow, or at least have the option to access it, so we did it in the **Stack Inspection** approach. Since the **Interpreter** approach needs to use an external package, we also needed to change the IDE’s commands to let the interpreter access the code instead of the Julia REPL.

All the traceability alternatives have their pros and cons. We analysed them and present a summary in table 3.1, showing what each of the approaches is capable to do or not.

---

\(^4\)Javassist library website: https://www.javassist.org/
### Metrics

<table>
<thead>
<tr>
<th>Needs an entry point</th>
<th>Code Redef</th>
<th>Code Injection</th>
<th>Interpreter</th>
<th>Stack Inspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.1: Traceability options.

#### 3.1.3 Implementation

The implementation of our methodology consists of two main packages. One for Atom (or another text editor which meets the requirements) and another one for Julia (or another programming language that meets the requirements for the chosen traceability option). Atom’s package will be dealing with the front end. Users will only interact directly with this package. Through it, they can ask Script-It to extract the generator algorithm (i.e. the basic algorithm extracted, composed only by primitive shape creation function calls) and/or activate traceability either on that algorithm or on already existing code. For each request, the Atom package will send a message to the package on the Julia side. This side is responsible for the computation of results and the activation of the data structures needed to activate traceability. This package is also the one which communicates with Khepri to get all the information needed and, in the Stack Inspection option, it is also Khepri which stores the traceability information. On the other hand, in the Interpreter option, the traceability information is stored by the Script-It Julia package.

In terms of data structures, our Interpreter solution stores a dictionary that saves for each location all the shapes generated. This location consists of the file path and code line. For the Stack Inspection, two dictionaries are used, one which stores, for each location, all the shapes it generated and another which stores, for each shape, all the control flow locations that generated it.

To implement traceability with the Interpreter approach, we used the JuliaInterpreter package\(^5\). JuliaInterpreter is a very optimized interpreter, thus allowing the execution of a program to continue without having huge overheads. This allows us to interpret every function call and update our location dictionary everytime a shape is created. However, in the current state of the JuliaInterpreter, we were only able to use this approach by having an entry point function to start the interpretation.

On the other hand, to implement traceability using Stack Inspection, we modified the existing Khepri implementation in a specific point. Since everytime a shape is created the control flow goes through a common point, we added some traceability data collection from the function-call stack at that point so that we can update our data structures. However, this option required the modification of Khepri, which would be harder if we were not allowed to access it directly. This is possible since Julia allows us to access the information of the stack fairly easily, requiring us only to filter out unnecessary information.

Even though we used the referred tools and programming language to meet the requirements, it is possible to change them. For instance, if we wanted to, we could use Racket instead of Julia and take...

---

\(^5\)JuliaInterpreter’s github: https://github.com/JuliaDebug/JuliaInterpreter.jl
advantage of hooks to inject our traceability code into the algorithm. Java also allows us to inspect the
stack so it could also be an option in which we would apply Stack Inspection.

3.2 Features

Our solution allows users to extract code from an existing digital model and generate a digital model
from existing code. Additionally, it gives users traceability features. Script-It also allows users to extract
functions from selected parts of code, as well as a free-refactoring mode, where users can modify the
algorithm to their content without depending on predefined refactoring methods. Since we focused more
on traceability, we limited the refactoring methods to function extraction and free-refactoring mode. This
works as follows.

3.2.1 Algorithm Extraction

To extract an algorithm, users press the key binding to do so. The generator algorithm is then extracted
with Khepri and written to the text editor. While this extraction is being executed, the necessary trace-
ability information is being stored in the respective data structures. When the extraction finishes, the
traceability mode is automatically activated. As we can see in Figure 3.5, first we have a model within
a digital modelling application and an empty text editor, when we activate the algorithm extraction all
the code is written in the text editor and presented to the user. The result is a bidirectionally traceable
algorithm which can connect each and every line of code to the respective shapes in the model.

3.2.2 Algorithm Execution

To execute an algorithm, users press the key binding to do so. Then, the algorithm is executed, gener-
ating the model in the digital modelling application. This is also done while the traceability information is
being processed and stored in the respective data structures. When the execution finishes, the traceabil-
ity mode is automatically activated. This is shown in Figure 3.6, where we start with an already existing
algorithm which will be executed. After the execution, we will have a traceable model.

3.2.3 Traceability

Traceability, as shown in Figures 3.7, 3.8 and 3.9, is bidirectional. As soon as we highlight a piece of
code in the text editor, the corresponding shapes are highlighted in the digital modelling application.
In the same way, when users want to reverse highlight one or more shapes, by selecting them in the
modelling application, they just have to press the respective key binding. This activates the reverse
highlighting of shapes, so that, when the user selects the shapes in the application, the corresponding
Figure 3.5: Extraction of an algorithm being done by Script-It.
Figure 3.6: Execution of an algorithm being done by Script-It.
Figure 3.7: Traceability from the text editor to the digital modelling application.
Figure 3.8: Reverse highlight feature from the digital modelling application to the text editor.
Figure 3.9: Traceability being applied to a compact and refactored algorithm.
lines of code in the text editor are highlighted. Figure 3.9 shows what a complete refactored algorithm looks like with traceability applied. In it, a single line of code is responsible for the creation of multiple shapes, therefore multiple shapes are highlighted.

3.2.4 Refactoring: Function Extraction

To extract a function, users simply select the lines that will be a part of the extracted function’s body, and then press the Function Extraction key binding. This creates the function, as well as replaces the original code with a function call. The traceability is then updated to reflect the changes. As we shown in Figure 3.10 we selected a few lines of code and then extracted a function from them.

3.2.5 Refactoring: Free-Refactoring mode

Free-refactoring mode happens when a user turns-off the traceability mode, changes the algorithm and finally turns-on the traceability mode again. This allows users to change specific parts of the code which are not available as already existing refactoring methods or are too detailed to do using a refactoring method. After all the changes are made to the code, when the traceability mode is turned on again, the whole code will be re-executed and the traceability information needed will be updated. This can be seen in Figure 3.11, where we manually extracted a function, changed its name, and rewrote the function call wherever we wanted to.

3.3 Traceability Implementation Comparison

As said before, we tested two traceability alternatives: Stack Inspection and Interpreter. For these two implementations, we tested all the features Script-It is capable of at the moment, and compared the execution time of each one, with both a refactored and a raw algorithm, where the latter is an algorithm consisting only of primitive function calls. We also compared the time taken with the increasing number of shapes that were created in the digital modelling tool. All these tests were done in a computer with the following specifications: CPU - Intel Core i5-5287U 2.9GHz; RAM - 8Gb; Operating System - Windows 10 64 bits.

As we can see in Figure 3.12, both approaches take almost the same time to execute the raw version of the models and introduce a big overhead when compared with running the code with no traceability. However, in Figure 3.13, we can see that, if the code is already refactored, Stack Inspection becomes a worse option for more complex models. On the other hand, when we test algorithm extraction, we get the opposite result, as shown in Figure 3.14. The Interpreter approach is initially better but starts to get much worse as the number of shapes increases. These initially better results happen since the
Figure 3.10: Function being extracted from pieces of an algorithm, after using reverse highlight to obtain the relevant code lines.
Figure 3.11: Function being extracted from pieces of an algorithm in free-refactoring mode.
implementation of the **Interpreter** option does not need to rerun the algorithm after the extraction, unlike the **Stack Inspection** option. Nevertheless, this **Interpreter** "shortcut" is only possible if all the extracted code lines are top-level primitive function calls. What we are doing to get this improvement is to set the dictionary entries to the according location and shape, when we extract the shapes. At the moment we
Figure 3.14: Comparison of the extraction of an algorithm with different traceability implementations.

After all the comparisons above, we concluded that the best trade-off for our solution will be obtained by using a Stack Inspection approach. We decided this because, when compared with the Interpreter extraction approach, despite being worse for a smaller number of shapes, it performs better in the long run. Additionally, it is more general in the case where the extraction does not only generate primitive function calls. Furthermore, unlike the Interpreter approach, with Stack Inspection we do not need an entry point to have traceability. For the worse execution times we can write all the main code outside functions to obtain better results. With the Stack Inspection approach, we also implemented complete control flow, i.e., we store more information about all the functions that had to be called in order to create a specific shape. This allows us to obtain more information when applying reverse highlight from the digital model to the algorithm. We are able, with reverse highlight, to highlight all the functions that had to be called to create the shapes we highlighted in the digital model. Furthermore, with Stack Inspection we do not have to create many new IDE’s handlers, thus having more independency.
3.4 Workflow

Now that we have already presented the features available in Script-It we can show a typical user workflow, as we can see in Figure 3.15.

This workflow has two different starting points. On the one hand, users can begin by extracting an algorithm from a digital model (step 1(a)). In this step the elements that represent the digital model are extracted, such as boxes and pyramids. The first script is produced although not yet with a clean final structure. For this extraction, Khepri will be used, which will allow the algorithm to be extracted. On the other hand, users can start with an already existing algorithm (step 1(b)). Regardless of the chosen path for step 1, the algorithm in use will be executed in order to obtain the necessary traceability features.

In step 2, users can refactorize the algorithm, taking advantage of the traceability capabilities in order to generate a more intelligible version of the algorithm. After this step the algorithm generated will still have traceability capabilities but will also be easier to test, optimize and understand when comparing with the extracted algorithm from step 1. This will allow users to test model variations much faster.

Finally, in step 3, users will be able to test new model variations, based on the initial input they used, yet with a cleaner structure and traceability features associated.

3.5 Summary

In this chapter, we presented the solution to our problem: the RAD methodology, which we then implemented into a tool, Script-It. This tool was built taking into account different alternatives to support...
traceability. We also showed the options we implemented and their advantages and disadvantages. We then proceed to test both the traceability options we implemented and explained why we chose the **Stack Inspection** one. Finally, we exemplified a possible workflow, going through each step of it.

The major advantages that this methodology aims to achieve are listed below, some of them were tested objectively while others were only tested subjectively:

- Generate a well-structured extracted procedural model;
- Ease the controllability of the procedural model;
- Ease the reutilization of non-algorithmic digital models;
- Ease the optimization process of the model;
- Ease the architects’ AD learning phase;
- Ease the conversion from 2D to 3D models;
- Hinder the insertion of errors when changing the procedural model;
4 Evaluation

Contents

4.1 User Testing ......................................................... 49
This chapter outlines the evaluation of the proposed solution. We want to verify our hypothesis: users will get better results by using Script-It, when compared with a manual AD approach. However, it is important to remark that, at the moment, there is a limited number of users who know how to use AD with Khepri, so we could not apply the most appropriate test analysis methods.

The user tests were done in the same computer on which the traceability testing was performed, being its specifications the following: CPU - Intel Core i5-5287U 2.9GHz; RAM - 8Gb; Operating System - Windows 10 64 bits.

The evaluation of our solution was conducted through different tests. The metrics considered for this study were (1) time and (2) accuracy. The first was tested using a digital model, to compare how long it took users to extract an algorithmic description while using our tool, with the time taken by extracting it manually with a digital modelling application. The same test was applied to evaluate accuracy: in addition to checking the time spent extracting the algorithm representing the model, we also observed the differences between the digital models produced by the extracted and modified algorithm, and the original and most compact one.

Using the two metrics described above, we assessed the usefulness of the proposed methodology. As optimization is a significant component of building design, we also evaluated the potential of our methodology for allowing users to optimize an already existing digital model, with or without an algorithmic representation associated. It was confirmed that modifications to the algorithm were achievable, being easily performed and verified in the model, not taking more than half a minute.

### 4.1 User Testing

While some users started by extracting the algorithm from a digital modelling application using Script-It, others started by extracting the algorithm manually. For the Script-It part, we let users have a warm-up phase where they would learn how to use the tool with a model different from the one used in the test. Afterwards users would start extracting the model, followed by a refactoring process and, finally, the application of changes to test other possible versions of the model.

The test example used for Script-It is represented in Figure 4.1, alongside an instance of extraction from a user. Here we can identify three equidistant abstract trees with the same number of components, a few circles and lines. The example used to test the manual extraction of code is represented in Figure 4.2. In this case, we presented a different model to prevent users from memorizing previous tests, yet maintaining the same components of the previous model and difficulty.

Tables 4.1 and 4.2 show the obtained results from the evaluation process. In the first one, representing the Script-It tests, we can see that the code extraction is considerably fast with a minimum of 5 seconds, and an average of 11 seconds. However, the code extracted in this step is not intelligible.
Figure 4.1: This figure shows both the model used for testing Script-It extraction and the extracted algorithm.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Extraction</th>
<th>Refactorization</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>00:06</td>
<td>07:20</td>
<td>100</td>
</tr>
<tr>
<td>#2</td>
<td>00:09</td>
<td>09:35</td>
<td>100</td>
</tr>
<tr>
<td>#3</td>
<td>00:17</td>
<td>10:00</td>
<td>80</td>
</tr>
<tr>
<td>#4</td>
<td>00:20</td>
<td>11:30</td>
<td>100</td>
</tr>
<tr>
<td>#5</td>
<td>00:05</td>
<td>05:55</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 4.1: Script-It’s test results. All the time measurements are displayed as MM:SS, where MM are the minutes and SS the seconds. The accuracy is measured in %.

Consequently users spent the majority of the time in the next step, trying to refactor their code. The average time for this task was of 8 minutes and 52 seconds. In order to have a base time reference for this test, we also asked a user to extract this model manually, which took him 16 minutes and 21 seconds to finish. Having this reference we understand that using Script-It was already helpful, since on average all the other users took less time.

The process of extraction and refactorization for the manual testing, shown in Table 4.2, is more time-consuming when compared to the use of Script-It, with an average time of 11 minutes and 7 seconds. In this assessment, the time spent on the extraction was not separated from the time spent refactoring, given that when users extract the algorithm manually they tend to compact and refactor the code from the beginning. Even though the time difference is not substantial for the presented tests, it can be significant for projects with a higher level of complexity, in which even a time-saving factor of 1% can result in considerable reductions on cost. When it comes to accuracy, we can identify that on average,
Figure 4.2: This figure shows both the model used for testing manual extraction and the extracted algorithm.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Extraction + Refactorization</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>07:50</td>
<td>100</td>
</tr>
<tr>
<td>#2</td>
<td>12:30</td>
<td>100</td>
</tr>
<tr>
<td>#3</td>
<td>14:36</td>
<td>100</td>
</tr>
<tr>
<td>#4</td>
<td>12:21</td>
<td>100</td>
</tr>
<tr>
<td>#5</td>
<td>08:19</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 4.2: Manual Extraction test results. All the time measurements are displayed as MM:SS, where MM are the minutes and SS the seconds. The accuracy is measured in %.

the model extracted automatically with Script-It has an accuracy of 96%. The lacking accuracy of 4% is introduced with the free-refactoring mode. With this process it is more common for users to introduce errors since it is not an automatic refactoring process. This can be overcome through the introduction of additional refactoring techniques, such as loop rerolling.

Considering the obtained results we conclude that Script-It is useful and capable of reducing the time needed to generate an algorithm from an existing digital model. We informally asked the test subjects to participate in this study, and learnt that even trained architects with many hours of digital modelling applications knowledge preferred and obtained better results by using our tool. We predict that for more complex models the difference between our approach and the manual one would be bigger.
## 5 Conclusion

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Future Work</td>
<td>56</td>
</tr>
</tbody>
</table>

53
With the evolution of technology, new techniques are being developed to help people achieve their goals. Such is visible in many research fields, one of them being Architecture, where the tools used to design models evolved from just pen and paper to computer applications. These applications allow not only the production of digital models, but also their analysis regarding different building performance criteria.

One powerful method of generating digital models is **PM**. With **PM**, a digital model is generated by evaluating the procedural model that was written. The most explored approaches are grammars and its different variations, which are not easy to read or edit, particularly if the user is not their original creator.

To tackle some of the limitations of grammars, a newer approach was proposed: **AD**, which consists in developing an algorithm that generates a digital model. Through **AD**, the modelling process becomes more methodical and intelligible in comparison with other **PM** approaches. Moreover, this approach is intrinsically parametric, allowing the creation of several variations of the same model just by modifying the parameters’ values in the algorithm. This is considered one of the main advantages of **AD**, since it allows not only architects to effortlessly test a wider range of design alternatives, but also to implement optimization processes.

Although **PM** and **AD** methodologies show several advantages, such as not having to create the whole model by hand, in a digital modelling application, its use still lacks in efficiency when dealing with models that have no representation associated to. To overcome this issue, **IPM** was created. This methodology focus on extracting the procedural model of an already existing digital model. However, most of the literature in this field also focuses on grammars. This dissertation proposes a new specific use of **IPM** with algorithms, a methodology we call **RAD**. Through this approach, we allow users to extract algorithms from existing digital models, but we decided to take it further. Additionally, we established a bidirectional traceable relation between the algorithm and the digital model in order to ease the refactoring process. The extracted algorithm is not in an intelligible state and without traceability the refactorization would not be an easy task, thus the user would not take full advantage of **AD**. The applicability of this methodology lies on the fact that regardless of its advantages, **AD** approaches are still not widely used, not only for being fairly recent but also because architects have little experience in programming. Therefore, architects still prefer to model their designs manually, for instance in a **CAD** or **BIM** tool. Moreover, the proposed approach strives to ease the users’ learning curve, by assisting in the recognition of connections between the algorithm and the model.

The main goal of this dissertation is to ease the extraction of algorithms from existing digital models, particularly, non-algorithmically generated ones. It also aims to help architects in their current design workflow: both (1) the ones who do not use **AD** as their main modelling approach but intend to do so, and (2) the ones who already take advantage of this approach but desire to take it further. With the proposed methodology, architects can obtain a well-structured algorithm representing an existing
digital model, taking advantage of AD when there was no algorithm and only a digital model to start with. The obtained algorithm is easy to read and edit, and can be adapted to each user’s needs semi-autonomously through refactoring techniques, leading to easier and less error prone models than those created manually. Since it is easier to explore new variations of the model using AD, users have more time to explore their creativity and can optimize the models they produce with lesser effort.

In this dissertation, we explored two of four different approaches to traceability presented in chapter 3, namely the **Interpreter** and **Stack Inspection** approaches. We concluded that **Stack Inspection** is the most adequate trade-off towards our requirements. It allows more flexibility for the user, as well as, a more informed design process due to its ability to highlight the control flow that generates a shape. We demonstrated through user testing that our goal of easing the extraction of algorithms from existing digital models is attainable. As hypothesised, users were able to extract the algorithm faster when compared to the manual version. Moreover, while testing and showing Script-It to users, we noticed that they were learning how to better refactor the code and, in later tries, they actually produced cleaner code.

Script-It produced helpful results leading the testing subjects to achieve the goal in shorter time periods. Although the time difference in the given examples was minor, for more complex projects we predict that the difference will increase. The evaluation results show that Script-It supports the application of RAD and, thus, allows designers to benefit from its advantages, particularly the quick generation of AD models from existing digital models.

### 5.1 Future Work

This dissertation advances the state of the art in RAD but does not end it: there are many ideas which we did not fully implement and that we think might be important to explore in the future. In this section we discuss some of them.

#### 5.1.1 Improved refactoring methods

The main focus of this dissertation was traceability. At the moment, we implemented only the following refactoring mechanisms: function extraction and free-refactoring mode. As future paths for improvement, we emphasize the need for adding more automatic refactoring methods to Script-It. For instance, a refactoring technique we identified as significant is the loop rerolling, which works by doing the opposite of what some compilers do to optimize the code, known as loop unrolling. While some compilers in order to optimize code switch a loop by a repetition of similar instructions that would be generated by the cycle, what loop rerolling does is, as the name suggests, transform a group of similar instructions into a cycle. As such, with this technique we can transform parts of the extracted code into cycles more easily.
5.1.2 Improve refresh algorithm

In the free-refactoring mode we deactivate the traceability, modify the code to our content and, lastly, when we activate the traceability all the code will be rerun in order to update the traceability information needed. This allows users to apply, manually, all the required refactoring techniques. However, if all the code has to be rerun every time a user applies the free-refactoring mode, the efficiency of this approach decreases and, for more complex projects, it will require longer time periods to finish the execution. With this in mind, a relevant optimization to test is to only rerun a specific part of the code, the modified part. This allows the user to see the free-refactoring modifications faster, thus allowing users to check if it is the correct modification or if it contains errors.

5.1.3 Improve control flow order

At the moment, when we use the reverse highlight (traceability from the digital model to the algorithm), we are able to obtain the control flow which generated a shape. However, this control flow is not ordered. An improvement for this feature would be to add some method of ordering the control flow, for instance we could lower the intensity of the line highlighting.

5.1.4 Select control flow size

While using the reverse highlight, users are able to see all the control flow which generates a shape. However, sometimes users might only want, for instance, obtain the last 3 calls in the function call stack, either because they are focusing in shallower range of the algorithm or because they want better performance, which is acquired by storing less information in the data structures.
Bibliography


