Reverse Algorithmic Design

Daniel Fermoselle
daniel.fermoselle@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2019

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; Reverse Algorithmic Design.

1. Introduction

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, 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, 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, AD. AD allows the architect to produce digital models by creating an algorithm that generates them [Terzidis, 2008]. 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: 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.

The main goal of this thesis is to introduce a methodology for IPM of architectural projects. This methodology will be implemented as a tool that can extract an algorithm from a digital model, particularly, a non-algorithmically generated one, 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: Section 2 contains related work in the area of PM, IPM, AD, Traceability, and Refactoring; Section 3 presents the solution to address the identified problems; Section 4 includes the evaluation of this thesis; Finally, Section 5 contains the conclusion and a discussion of how this project will ease the work of architects.

2. Related Work

To achieve our goal we explored previous researches in the areas of Procedural Modelling, Inverse Procedural Modelling, Algorithmic Design, Traceability and Refactoring.

PM consists in generating content (digital models) by writing and interpreting a model description written in a procedure or program [Beneš et al., 2011]. 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]. It is easier to create and modify these rules than building and changing 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. These grammars evolved from simple L-systems [Lindenmayer, 1968, Prusinkiewicz and Lindenmayer, 1990], a parallel string rewriting system, to more complex representations like the CGA shape grammars [Müller et al., 2006], which already have a specific shape representation and apply rules sequentially, thus being better at modelling buildings. However 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 [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. 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. We want to find a better solution to be used as the procedural model.

Although PM is a good approach to create and manipulate digital models, it is not efficient to manually create a procedural model for an already existing model. Therefore, new methodologies were created in order to extract the procedural model from existing digital models, a process known as IPM. With IPM a designer can reuse previous models which did not have an associated procedural representation. Most of the related work we explored focused on extracting grammars, since they are the most used procedural descriptions. However, not all authors used the same types of grammars. The types of grammars used include L-Systems [Št’ava et al., 2010], Split Grammars [Wu et al., 2013], and Shape Grammars [Bokeloh et al., 2010]. Even though not all the mentioned authors used the same types of grammars, they all agreed that the key factors for an easier IPM are repetition and symmetry.
The implementations of this methodology consist mainly in creating clusters of similar shapes from which significant rules can be extracted.

PM and IPM are important, however their main approaches and implementations use grammars, which have many disadvantages, mainly being non deterministic by default. A newer approach intends to overcome the problems of previous approaches. AD is an approach that uses algorithms in order to create digital models [Terzidis, 2006], it 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. Due to its algorithmic nature, AD allows the user to have better control of the produced model. Despite the difficulty of building 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. 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). 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\(^1\). 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, which does not happen with usual AD approaches. Typically 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 [Alfaiaete, 2017]; immediate feedback and traceability.

Traceability, in a broader definition, 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 better 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), 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.

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. 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. We learnt from previous research [Murphy-Hill et al., 2012], which refactoring techniques are most used, how tools should be built to ease the users’ tasks, and which type of users practice refactoring the most. Namely, 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

\(^1\)Radiance allows the analysis and visualization of lighting. Radiance’s web page: https://www.radiance-online.org
know when to use the tool; (Trust) the user should trust that the tool will not damage existing code.

3. Solution
With this dissertation we introduce RAD, the specific use of algorithms in an IPM methodology. It consists in obtaining the algorithm that generated a digital model given as input from a digital modelling application. 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 section, we present our methodology and a tool which was created to support it, Script-It.

3.1. Methodology

As explained in section 2, current AD methodologies consist in using algorithms to create digital models. The goal of RAD is essentially, to invert the usual AD workflow and it is shown in Figure 1. 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 1, 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, as well as participate in the traceability, highlighting specific shapes when needed.

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$^2$, 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$^3$ 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$^4$.

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 2.
This dissertation focused more on traceability, because without it users will not be able to easily refactor the algorithm. We explored multiple alternatives to create the traceability option which would fit our requirements. 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 exam 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, and 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.

**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. 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 Julia Interpreter, 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 analyzed them and present a summary in table 1, showing what each of the approaches is capable to do or not.

3.3. 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 Interpreter 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 Julia Interpreter package. Julia Interpreter 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 every time a shape is created. However, in the current state of the Julia Interpreter, 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 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.

3.4. 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. Furthermore, users can also use already existing algorithms and execute them with Script-It, still obtaining in this approach the same traceability features as an extraction would. Since we focused more on

5Julia Interpreter’s github: https://github.com/JuliaDebug/JuliaInterpreter.jl
traceability, we limited the refactoring methods to function extraction and free-refactoring mode. We show in Figure 4 how traceability works.

### 3.5. Traceability Implementation Comparison

As said before, we tested two traceability options: **Stack Inspection** and **Interpreter**. For these two implementations, we tested all the features ScriptIt is capable of at the moment, and compared the execution time of each traceability option, 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.

![Figure 4: Traceability from the text editor to the digital modelling application.](image)

As we can see in Figure 5, 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 6, 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 7. 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 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 extract the first shape, we know exactly where it will be written in the file, so we just write that location and shape in the dictionary. In future work, we can also implement this in the **Stack Inspection** option when the extracted code lines are top-level primitive function calls.

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

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Code Redef</th>
<th>Code Injection</th>
<th>Interpreter</th>
<th>Stack Inspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Needs an entry point</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can highlight the control flow</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Needs to modify IDE’s commands</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: Traceability options.
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.

3.6. Workflow

![Figure 8: Typical Script-It’s user workflow.](image)

Script-It’s workflow, as we can see in Figure 8, 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.

4. Results & discussion

This section 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.

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 assessed by comparing how long it took users to extract an algorithmic description from a digital model 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 on average.

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 test, 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 variations of the model.

Tables 2 and 3 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. Since the extracted code is not intelligible, 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 3, 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, the model extracted automatically with Script-It has an accuracy of 96%. The lacking accuracy of 4% resulted from 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 in the future 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 learnt that even trained architects with many hours of digital modelling applications and AD usage 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. Conclusions
Although PM and AD methodologies reveal 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 focuses 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, by establishing a bidirectional traceable relation between the algorithm and the digital model. 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. 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.

We explored two of four different approaches to traceability, namely the Interpreter and the Stack Inspection. From section 3.5, we concluded that the Stack Inspection approach 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.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Extraction</th>
<th>Refactoring</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 2: Script-It 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 %.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Extraction + Refactoring</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 3: 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 %.
when compared to the manual version. 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. The most important of them is associated with refactoring. In this dissertation we focused mainly on the traceability side of the problem. 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. That is, 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.

References


