Introducing RepoCad

A prototype of the Internet of Digital Design

Ole Egholm Jackson[1], Jens Egholm Pedersen[2]
[1]Aarhus School of Architecture [2]CERN
[1,2]www.repocad.com
[1]oleegholm.jackson@aarch.dk [2]jens.egholm@cern.ch

Based on a state of the art analysis of computational design technologies and collaborative software development practises, this paper proposes a synthesis of existing strategies in an Internet of Digital Design. This paper introduces the experimental design platform RepoCad, which consists of three elements: a simplified scripting language, an online library and a drawing interface. The result is an online platform where tools, design processes and design results are accessible and editable from a web browser.

Keywords: *CAD software, online design tools, digital design, scripting, parametric design, online drawing library, open-source, computation*

1. INTRODUCTION: AN INTERNET OF DIGITAL DESIGN

The benefits of designing with calculus as opposed to tracing, makes a viable case for questioning the state of the design logic currently in the hand of designers. Potentials of utilising computation in design include individualization through versioning, addressing material evidence as parameters and the direct link to digital production without the need for human interpretation (Rocker 2008, Pedersen 2010). This paper proposes a strategy to further exploit the advancements in digital technologies, by demonstrating how collaborative design and development can assist in the development of an Internet of Digital Design.

In section two of this paper the use of digital technologies in existing software design tools are analysed. Emphasis is put on software which implements computation as a way of creating designs, software which make sharing and collaboration possible and software which aims to ease the task of programming designs. Section three describes a concept for moving even more of the creative process into the digital universe. The section sets out to explore how to minimize the gap between designer and computer, and enhance the collaboration between individuals in the whole design community. Subsequent, a strategy to use computational ideas in combination with modern programming language techniques and web-technologies is explored. In section four RepoCad is presented as a prototypical implementation of the concepts defined in section three, using a basic 2D environment.

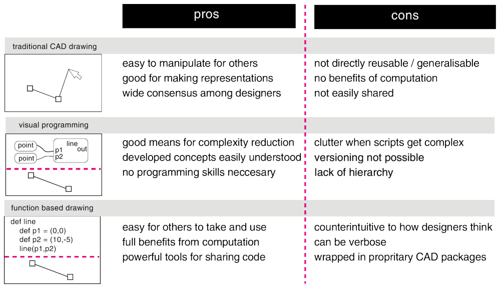
2. STATE OF THE ART

Since the 1990's computer scientists, architects and researchers have worked on expanding CAD drawing beyond the pure mimicking of analogue drawing, and utilise the techniques of computation as well as the possibilities for collaboration over the internet (Shelden et. al 1995, Koutsabasis et. al. 2012). The tools that are available at present offer only some of these features. Autodesk (R) has developed AutoCad 360 (R) which is directed towards engineering and late stage building design. Using the drawing tools is both free and offers collaborative traditional CAD drawing, but the core is closed and users cannot add or modify drawing tools. The platform does not encourage sharing of drawings, and scripting is not possible. SketchUp (R) allows scripting using a Ruby-script plugin. SketchUp is also closed source, and generally not set up for web-based development and sharing of scripts. An exception is the project WikiHouse (Galilee 2012), which gives an idea of the potential of sharing design knowledge online. Several programming-based design tools exist, eg. Python-scripting in Rhinocerous 3d (R) which is used widely in design research (TM) and OpenGL Shaded Language (GLSL). GSLS is used to generate procedural, GPU accelerated graphics for computer games and the language is verbose and not very user friendly. Rhinoceros (R) designers and researchers often do not generalise and parametrise their tools when scripting. Sharing of code is not an integral part of the software packages used, but requires active participating and deploying of code in a community. Visual programming add-ons such as Grasshopper (R) for Rhinoceros (R) and Dynamo (R) for Revit (R) are successful in presenting a workflow which allows non-programmers to understand the syntax, assisted by the fact that code is evaluated in real time. These languages, however, support mostly procedural programming. The scripts and drawings developed are often generalized and parameterized, and scalability through use of import statements is not available. The Java-based language and rendering engine Processing is widely used for interaction design and computation. It is however mainly developed and used for interactive and graphic design. Internet based and open-source apps include paperjs.org (9) and openJSCad (10), none of which are developed specifically for design or architecture. The introduced platform draws inspiration from the above mentioned software packages, proposing a synthesis of their features.

3. METHODOLOGY

This section discusses practices in the field of computer science and how they can be applied in computational design processes. Section 3.1 explores contemporary practices of programming languages as the basis of human-computer interaction. Following this, version control management is presented as a method for textual collaboration. Lastly, the two concepts are merged into a methodology for supporting global collaboration on computational design.

3.1. Language as human-computer interaction


point
point
line
p1
p2
out

	pros	cons
traditional CAD drawing	easy to manipulate for others good for making representations wide consensus among designers	not directly reusable / generalisable no benefits of computation not easily shared
visual programming	good means for complexity reduction developed concepts easily understood no programming skills neccesary	clutter when scripts get complex versioning not possible lack of hierarchy
function based drawing	easy for others to take and use full benefits from computation powerful tools for sharing code	counterintuitive to how designers think can be verbose wrapped in propritary CAD packages

```
def line
   def p1 = (0,0)
   def p2 = (10,-5)
   line(p1,p2)
```

Figure 1
Three different methods for creating digital design: analogue, graphical and code based.

Three modes of human-computer interaction for CAD software exist (figure 1).The traditionally visual approach of drawing seen in for instance AutoCAD uses a "point and click" interface to generate objects on a canvas. As this approach does not allow for parametric connections and computed variations, it does not readily support computational design. Software solutions such as Grasshopper (R) use visually programming to 'wire' discrete functions together, allowing the designer to create intricate parametric designs. This approach lacks some key features before it may be used for sharing code. Scripts are wrapped in encrypted files and is not versioned (see section 3.2 below). Further it is not possible to split large scripts into importable functions, and scripts tend to clutter due to the lack of a hierarchy. Other tools such as Shadertoy (8) have recently taken a different approach by using code to programmatically draw demanding graphics, such as fluid simulation and landscape generation, on a canvas in real-time. The parameterisation inherent in the language makes such intricate and large sceneries possible; to recreate this with manual digital drawing processes would be extremely time consuming. By writing code as reusable components, complex behaviour can be derived from predefined discrete blocks of functionality and spare the designer from reinventing functionality.

The use of programming language for design requires that the written code is acting upon a graphical surface. This is typically done by first breaking the written code into tokens that can be understood by the computer (parsing) and second by executing the found instructions on a media (evaluation, Sestoft 2012). After the parsing phase a set of instructions is produced (an Abstract Syntax Tree, AST, Pfenning 1988). By separating the instructions from the surface media, the code can effectively be applied on any graphical surface, such as a PDF or even a browser window.

Programming languages are defined by Ben-Ari (1996) as an "abstraction mechanism" that is forced to balance between a flat learning curve and a large repertoire of functionality (Ben-Ari 1996, p. 3). A language with a large set of syntax rules handles tasks with higher precision, than a less expressive language. Unfortunately such a rich language can be hard to learn and maintain, due to the complex and confusing program-flow.

A common technique to achieve high expressiveness with a few simple rules is the use of programming paradigms (Ben-Ari 1996). Procedural and functional programming are two such paradigms (Sestoft 2012). The procedural code executes state changing instructions from top to bottom, but uses functions to define reusable behaviour. The functional programming paradigm uses a similar concept of functions, but only allows state changes via immutable alteration of information (ibid.). In essense both paradigms pose a number of restrictions on the programmer to avoid writing ambiguous or erroneous code. Such restrictions can cause inconvenience, but are an important part of structured programming to improve the quality and development time of the code (Sestoft 2012, Ben-Ari 1996).

3.2 Version control

The biggest challenge of online collaboration in general is the consensus building of a multi stakeholder project. The time delay in the submission of changes can lead to conflicting versions of the same piece of information (Rochkind 1975). Successful collaboration platforms such as Google docs and Microsoft 365 have proven that it is feasible to interchangeably update text in real-time and synchronise changes to avoid this issue. This is typically done by ranking the changes and increasing the granularity of data (Lamport 1978). The latter is used to minimize the extent of conflicts; by reducing the size of a unit from an entire document to lines or even characters, the chance of conflicts with other units decreases. Combined with a time-stamp of the changes, conflicts can be solved by overwriting changes with a lower priority.

This method, also referred to as version control, is a prerequisite for parallelised work processes in modern software projects (Chacon and Straub 2014). Version control stores unique versions of a document since its creation and incorporates merging policies for conflicting text (ibid.).

3.3 Combining computation and collaboration

To combine the concepts of language and version control in an online digital design environment, three technologies are needed: A graphical interface, a scripting language and a library of scripts (figure 2).

The development of web technologies has recently made it possible to efficiently visualise graphics from within a browser (12). HTML5 and JavaScript (ibid.) provide a standard to draw graphics (hardware accelerated and not), which is adopted in the large majority of browsers (13). Using such a browser based environment for coding tasks requires a rapid feedback loop, to show the effect of the code changes in close to real-time. A scripting language for computational design requires it to be sufficiently readable for the average designer with only little programming knowledge. This rules out complicated syntax rules as seen in the GSLS language (8).

Both the procedural and functional programming paradigms are suited for such a language, because they break software into discrete functions. Such functions can be designed to elegantly perform complex calculations by the combination of many simple functions. To allow this functionality to be reused indefinitely, a library is required to store them. The library should be globally accessible via the web and be able to handle concurrent access and manipulation.

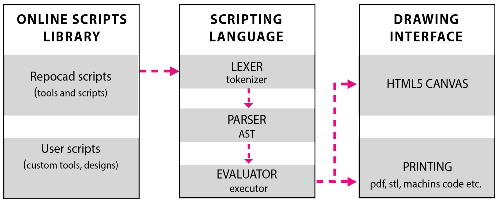


Figure 2
Schematic of the connection between an online library, a scripting language and a browser-based drawing interface.

4 EXPERIMENTAL IMPLEMENTATION

The concepts explained are implemented in a platform dubbed RepoCad, available on the webpage repocad.com. "Repo" is short for repository, signifying the importance of online storage and accessibility. The title also contain a pun on "repossession": the reclaiming of tools used for creating designs. A majority of the software used in design practices is closed source, which effectively represents a barrier for allowing users to influence their design tools. RepoCad and all the drawings in its library are open-source.

4.1 A scripting language: RepoScript

Common scripting languages like Python or Lua are traditionally developed for heavy-weight desktop applications, and contain many advanced programming techniques (1, 2). Because the language does not solely target design platforms, this can pose a challenge for designers with little programming experience. As the traditional choice of scripting for browsers, JavaScript similarly contains many programming techniques which are superfluous for the purpose of computational design (3). While it is possible to use Python and Lua online, the complexity of the syntax has lead us to discard these three language as a scripting language for the platform.

Because of this lack of simple scripting languages targeted a graphical web-based platform, an experiment is described in which such a language is proposed and tested: A domain-specific procedural language with limited functionality, but which is more expressive than Design Script (Aish 2013). It is inspired by Python, Scala and C (and others) and aims to enable users to construct drawings and tools with a minimum of programming experience. Reposcript is interpreted in three phases as shown in the language column of figure 2.

First it converts the text into syntactic tokens, as defined in figure 4 to 6. Second, it reads the tokens and compiles them into a number of instructions stored in an abstract syntax tree. This tree of syntax commands is lastly applied on a graphical surface by the evaluator. Currently HTML5 and PDF are the only supported graphical end-points.

The primary means to achieve simplicity is a small number of simple syntax rules. They are simple enough to omit irrelevant details, but expressive enough to provide access to necessary tools for graphical manipulation. Figure 3 displays the full set of abstract syntax definitions in the language.

```
comparison-statement ::= value-reference == | > | < | <= | >= value-reference
operation-statement ::= value-reference + | - | * | / value-reference
value-statement ::= def value-name = value
block-statement ::= statement [ statement [ … ]]
function-statement ::= def function-name( [parameter [ parameter [ … ]]] ) statement
```

Figure 4
Definition of the value- and function-statements in Extended Backus-Naur form.

A language statement consists of either a comparison, operation, value definition, function definition or a block of statements shown in figure 4. A block contains one or more statement, but are only executed once. Such a block is used in functions

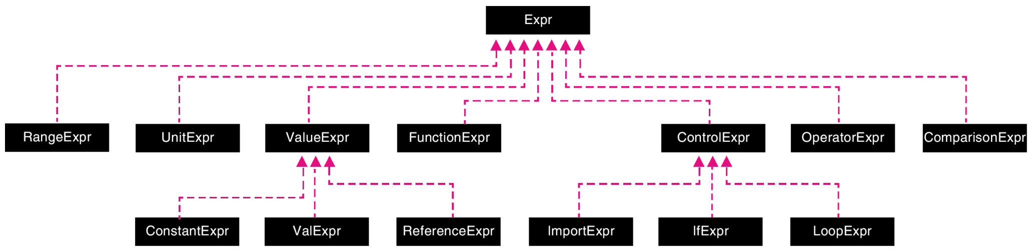


Figure 3
The abstract syntax tree definitions of RepoScript.

where a number of statements is defined to be executed many times with zero or more variations. A block also controls the visibility of variables, or scope, of the script. A value defined inside a block cannot be seen outside the block. This prevents variable cluttering seen in many imperative languages (Ben-Ari 1996). A function specifies a number of parameters that changes the behaviour of the statement within the function. Like many functional languages, every expression in RepoScript results in a value after its execution. A function statement can then be used to calculate a value and give the result back to, say, a value definition. However, some statements like the drawing functions below, do not have a meaningful return-value. Instead they explicitly return the empty value 'Unit'.

Figure 5
Definition of the drawing-functions in Extended Backus-Naur form.

$$\begin{gathered}
\textit{arc-statement} ::= \textbf{arc}(x_0\ y_0\ \textit{radius start-angle end-angle}) \\
\textit{bezier-statement} ::= \textbf{bezier}(x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ x_4\ y_4) \\
\textit{circle} ::= \textbf{circle}(x_0\ y_0\ \textit{radius}) \\
\textit{line-statement} ::= \textbf{line}(x_1\ y_1\ x_2\ y_2) \\
\textit{text} ::= \textbf{text}(x_0\ y_0\ \textit{height text})
\end{gathered}$$

All drawing functionality is implented as functions of RepoScript. There are currently five elementary drawing commands in Reposcript: arc, bezier, circle, line and text (see figure 5). They are rudimentary on purpose, but can create complex geometries when combined.

Figure 6
Definition of the control-structures in Extended Backus-Naur form.

$$\begin{gathered}
\textit{import-statement} ::= \textbf{import}\ \textit{script-name}\ [\ \textbf{as}\ \textit{script-alias}\ ] \\
\textit{if-statement} ::= \textbf{if}\ (\ \textit{expression}\ )\ \textit{statement}\ [\ \textbf{else}\ \textit{statement}\ ] \\
\textit{loop-statement} ::= \textbf{repeat}\ [\ \textit{from-value}\ \textbf{to}\ ]\ \textit{to-value}\ [\ \textbf{def}\ \textit{counter-name}\ ]\ \textit{statement}
\end{gathered}$$

The control-structures are implemented to control the procedural flow (figure 6). The loop statement permits a code block to be executed many number of times. It can either be expressed by a simple to-value, where the loop will iterate to number of times, or with an interval, delimited by the from-value and the to-value. The counter-name is an optional variable that binds the value of the loop-counter to be used in the statement inside the loop. The if-statement controls the script-flow based on the comparison operator above. If a condition is found to be true, the first statement is executed, if not, the second is executed. Lastly the import statements load the functional routines defined in other scripts to be re-used in the current scope of the script.

4.2 A drawing library: Git

In RepoScript each user-written code-file is a set of tools that can be imported by other users in other scripts. This modularity is enforced by the use of the git versioning system. All code is versioned, given an URL and stored online for future access. Combined with the modularity of the scripting language, this versioning provides a way to synchronise and merge multiple versions of a script - a requirement for online collaboration. It also lays the foundation for an unprecedented online collection of reusable drawing functionality. Every drawing and drawing tool will be stored and freely accessible for everyone.

4.3 A drawing interface: RepoCad

The drawing library is combined with a real-time interpretation of RepoScript. RepoCad uses JavaScript to evaluate the code and renders the resulting geometries in HTML5. The interface is inspired by the concept of a read-evaluate-print loop, known in programming circles. To quickly test a code snippet, a small interpreter is used to read and evaluate the code and print the result. Contrary to analysing entire software systems this is very efficient to get a proof of how small sections of code behaves. The same feedback loop is applied in the RepoCad interface. An evaluation is triggered every time the code changes, to provide the instantaneous graphical feedback. This allows users to see a visual result of their programming and use it to rewrite the code. Like the library scripts the code for the website is versioned and available online (11).

Figure 7 illustrates the layout of the website: a code-editor is placed in the left pane and the result of the interpretation is immediately visualised on the right. By default the drawings are fitted to an A4 format that automatically scales based on the drawing size. This format may be printed to PDF - either to 1:1 (fabrication) or to scale (blueprints) - using the 'print' button in the bottom, and persisted in Git using the 'save' button.

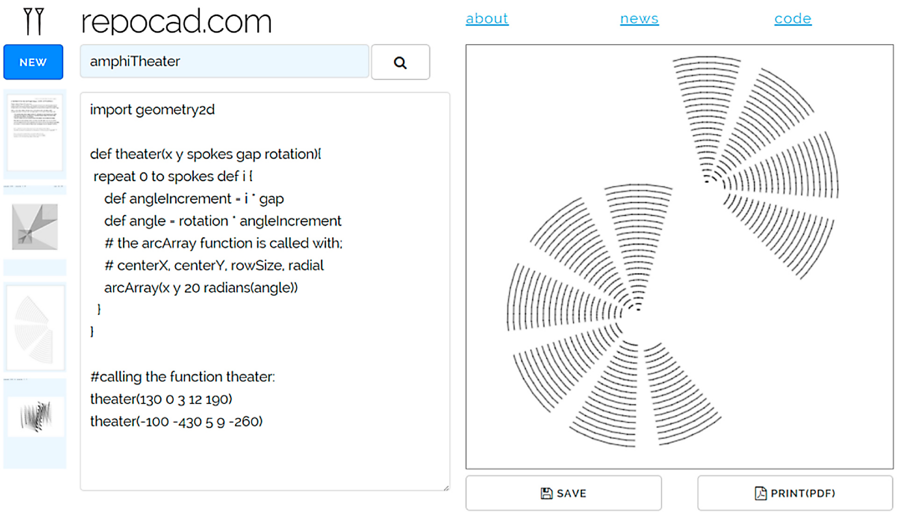
4.4 Diagrammatic design example

The concept for merging "tools" and "drawings" is illustrated in figure 7 by the functional approach to drawing a diagrammatic plan of an amphitheater. The right pane shows a drawing of two amphitheaters, derived from the same "theater" function, in the left panel. The theater function parameterises the position of the arc (x_0 and y_0), the number of seating sections in the theater (spokes), the gap between the arc-cones (gap) and the rotation of the spokes (rotation). This parameterisation allows the amphiTheater drawing to become a tool which users will be able to import and use in other drawings. This is seen in practice on a lower level in the "theater" function. It uses a function called "arcArray", which stems from the imported script "geometry2d". Such drawing hierarchies can be extended ad libitum to derive complex drawing mechanics.

5 CHALLENGES AND FUTURE WORK

5.1 User interface improvements

To improve user friendliness of the interface, icons will be put in place for adding and running functions in a script. Clicking an icon will put the relevant code into the script, as well as display the corresponding artwork on the canvas. This will provide the classic 'manual' CAD style creations, transformations and



```
import geometry2d

def theater(x y spokes gap rotation){
 repeat 0 to spokes def i {
   def angleIncrement = i * gap
   def angle = rotation * angleIncrement
   # the arcArray function is called with:
   # centerX, centerY, rowSize, radial
   arcArray(x y 20 radians(angle))
  }
}

#calling the function theater:
theater(130 0 3 12 190)
theater(-100 -430 5 9 -260)
```

Figure 7
A screenshot of RepoCad where an online script renders two amphitheaters a function called directly from another script. Everything is versioned at github.

editions of drawings, but in a setup where artwork is entirely parametric and code-based. Furthermore, the code editor will include syntax highlighting and code-completion. Syntax highlighting colour-codes the scripts to make them easier to read. It allows the user to filter out irrelevant implementation details. Code-completion helps the user to remember the exact syntax rules and the name of functions from imported scripts. Both initiatives are intended to aid the generation of code. To achieve a better usability of the interface, the user will also be able to modify the output of the scripts by working on the graphics themselves. To avoid inconsistencies between the script and the graphical output, these changes will reflect back on the relevant parameters altered. This technique is already seen in some graphical programming, for instance the antimony project (4).

5.2 Language improvements

A programming language will always have to balance between abstraction level and the degree of control. RepoScript purposefully focuses on abstractions, but is in continuous development to improve its usability. The following roadmap identifies possible hurdles and propose a solution by introducing: 1) type-safety, 2) data-types, 3) dependency locking and 4) event-programming.

1) Type safety will make it easier for users to write scripts that behave as expected, because the types check for any accidental type-violations. It also makes it easier to connect to the functionality of modules, written by other users, because the types reveal which value is expected and allowed.

2) The type-safety will pave the way for the use of data-types. By classifying data into types, the programmer can construct hierarchies of types which can solve very specific tasks. One of these tasks is the use of collections of elements in lists or similar data types, which can be used to simplify the interaction with many values at once.

3) Dependency locking is a need that derives from the shared nature of the scripts. Because scripts are publicly available they are constantly subject to change without warning. However, the versioning of scripts means that every change to a script is retained since its creation. These changes can be seen as unique versions with a unique identifier, or time-stamp. Dependency locking is a feature that uses this time-stamp to ensure the same version of an imported script will always be used regardless if other users produce newer versions.

4) Lastly it is a goal for RepoScript to not only control the graphics, but also the interaction with the graphics. The manipulation of the graphics as described in 5.1, could be programmed in RepoScript and be publicly available like the rest of the scripts. This will mean extending the language to handle user interactions and reflect these interactions back to the script. Such interaction mechanisms would, like the scripts, be available in modules, that could be chosen to fit the task of the user. In the language this will be implemented using reactive programming, which drastically simplifies traditional event-handling (Odersky 2012).

5.3 Platform improvements

The prototypical implementation of RepoCad has shown that 2D designs are feasible, but to widen the applicability of RepoCad to other other platforms, an important next step is to support 3-dimensional designs. Currently the designs can be exported to the web-based library and PDF files, but with 3D this can be extended to design formats that can be used in other platforms and allow the designs to be sent to digital production facilities. Implementing 3D will also open for the use of RepoCad as a plugin for existing software-packages, where their functionality can be used in place of RepoCad. Another relevant improvement for the platform is an implementation of faster JavaScript execution. JavaScript does not perform as well as native desktop applications, which will pose a challenge with larges and more complex scripts. However, this performance gap has been closed by the asm.js project (6), which accelerates JavaScript to a speed that is comparable to desktop applications (ibid.). While the technology is adapted in most newer browsers such as Firefox, Chrome and Opera (5), browsers such as Internet Explorer 9 and Safari do not support this speedup (6).

6 CONCLUSION

The paper has presented the state of the art in computational design tools as well as tools for collaboration and sharing. By drawing from the experience in the field, a novel strategy has been proposed for giving designers the possibility to create designs and design tools with real time graphical feedback and to share their designs and tools globally. The result of this strategy is dubbed the Internet of Digital Design.

The key concepts of the strategy are tested in a piece of experimental software "RepoCad", that illustrates how designs based on parameterised functions can become a part of a global platform. RepoCad designs are written en RepoScript - a prototype for a high-level scripting language with a low learning curve. The language has a large repertoire of functionality and is aimed to enable the construction of drawings and tools with a minimum of programming experience.

In order to widen the applicability, it is concluded that RepoCad should be interfaced with other platforms. This could be done by extending the code to work as a plugin for existing software-packages, thereby giving access to already implemented means for exporting designs to digital production facilities.

REFERENCES

Aish, R 2013 'DesignScript: Scalable Tools for Design Computation', *Proceedings of the 31st eCAADe Conference – Volume 2*, pp. 87-95

Ben-Ari, M 1996, *Understanding programming languages*, Wiley, New York

Chacon, S and Straub, B 2014, *Pro Git*, Apress

Galilee, B 2012, 'Open-source housing', *Domus*, 2012(959), pp. 72-75

Koutsabasis, P, Vosinakis, S, Malisova, K and Paparounas, N 2012, 'On the value of Virtual Worlds for collaborative design', *Design Studies*, 2012(4), pp. 357-390

Lamport, L 1978 'Time, clocks, and the ordering of events in a distributed system', *Communications of the ACM 21 (7)*, p. 558–565

Menges, A 2012, 'Material computation: higher integration in morphogenetic design', *Architectural Design*, 2012-2, pp. 6-143

Odersky, M and Ingo, M 2012 'Deprecating the Observer Pattern with Scala.React', *EPFL report 176887*, pp. 1-20

Pedersen, OE 2011, 'Material Evidence In A Digital Context', *Aarhus Documents*, 14(2), pp. 14-15

Pfenning, F and Elliott, C 1988 'Higher-Order Abstract Syntax', *Proceedings of the ACM SIGPLAN PLDI*, pp. 199-208

Rizal, S 2007 'Managing Collaborative Design', *Buron Academic Publishers*

Rochkind, MJ 1975 'The Source Code Control System', *IEEE Transactions on Software Engineering SE–1 (4)*, p. 364–370

Rocker, I 2008 'Versioning: Architecture as series?', *First International Conference on Critical Digital: What Matters(s)? - 18-19 April 2008, Harvard University Graduate School of Design*, Cambridge, pp. 157-170

Schneider, S, Braunes, J, Thurow, T and Bauhaus, RK 2011 'Design Versioning – Problems and possible solutions for the automatic management of distributed design processes', *Proceedings of the 14th International conference on Computer Aided Architectural Design*

Sestoft, P 2012, *Programming Language Concepts*, Springer

Shelden, DR, Bharwani, R, Mitchell, S, William, J and Williams, J 1995, 'Requirements for virtual design review', *Architectural Research Quarterly*, 1(2), pp. 80-89

[1] http://www.lua.org/

[2] https://www.python.org/

[3] http://www.ecma-international.org/publications/standards/Ecma-262.htm

[4] http://www.mattkeeter.com/projects/antimony/3/

[5] https://blog.mozilla.org/futurereleases/2013/11/26/chrome-and-opera-optimize-for-mozilla-pioneered-asm-js/

[6] http://kripken.github.io/mloc_emscripten_talk/s-loop.html

[7] http://asmjs.org/

[8] http://shadertoy.com/

[9] http://paperjs.org/

[10] http://openjscad.org/

[11] https://github.com/repocad/web

[12] http://www.w3.org/TR/html5/

[13] http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(HTML5)