

# Exploring Instruments for Online Empirical Modelling

by

**Joe Butler**

**Project Report**

Submitted to the University of Warwick  
for a decent mark in

**CS310**

**Department of Computer Science**

May 2014

THE UNIVERSITY OF  
**WARWICK**

This page was intentionally left blank.

## Keywords

**Javascript** - an object-oriented computer programming language commonly used to create interactive effects within web browsers.

**JQuery** - a multi-browser Javascript library designed to “simplify” the client-side scripting of HTML.

**Backend** - denoting a part of a computer program inaccessible by the user, eg the server side code part of an online application.

**EM** - Empirical Modelling (See section 1.1)

**EDEN** - Engine/Evaluator for definitive notations (See section 1.1)

**CS405** - A 4th Year Taught Module in the Department of Computer Science at The University of Warwick.

(see <http://www2.warwick.ac.uk/fac/sci/dcs/teaching/modules/cs405/>)

**JS-EDEN** - A Javascript online environment for Empirical Modelling

(see <http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/js-eden/>)

**TK-EDEN** - A low level implementation of the EDEN environment for Linux, Mac and Windows.

**SCOUT** - A notation for screen layout developed by the Empirical Modelling Research Group at The University of Warwick.

(see <http://www2.warwick.ac.uk/fac/sci/dcs/research/em/notations/scout/>)

**DONALD** - A notation for 2d line drawing developed by the Empirical Modelling Research Group at The University of Warwick.

(see <http://www2.warwick.ac.uk/fac/sci/dcs/research/em/notations/donald/>)

**EDDI** - A notation for database interpretation developed by the Empirical Modelling Research Group at The University of Warwick.

(see <http://www2.warwick.ac.uk/fac/sci/dcs/research/em/notations/eddi/>)

**HTML** - Hypertext Markup Language, a standardised notation for structuring webpage.

**CSS** - Cascading Style Sheets, a standardised notation for describing the presentation semantics of a document written in HTML.

**API** - Application Programming Interface, a specification for how software components should interact with each other.

**DOM** - Document Object Model, A Javascript accessible API for HTML.

**CaS** - Computing at School, An institutional scheme for promoting the teaching of Computer Science at School.

**MOOC** - Massively Open Online Course

## Acknowledgements

Above all others I would like to thank my supervisor Meurig Beynon for his guidance and encouragement throughout this project, without whom this endeavour would have been very lonely and probably have strayed far from the realms of Empirical Modelling. I would also like to thank Jonny Foss for his technical insight and rigorous problem solving passion for web technologies and Jane Sinclair for her invaluable advice and interminable positive demeanor.

## Abstract

JS-EDEN possesses vast potential as an open online instrument to enable constructivist computing. Though the current state of the environment is less than ideal, investigation into how the platform can be re-engineered and extended to attract the enthusiasm of modellers the world over, is undertaken and explored. Three significant developments and many minor improvements have been demonstrated to support the modeller in their activities. The project will lead to further development of the tool as a result of interest from third parties.

# Contents

## 1. Introduction

- 1.0 Overview (9)
- 1.1 Empirical Modelling & EDEN (9)
- 1.2 Current Issues & Motivations for Development (11)

## 2. Project Proposal

- 2.0 Overview (12)
- 2.1 Objective 0 - Environment Investigations (13)
- 2.2 Objective 1 - The State Description Maintainer (14)
- 2.3 Objective 2 - The Dependency Map (14)
- 2.4 Objective 3 - The State Timeline (15)
- 2.5 Continuous Objective - Miscellaneous Engineering (16)
- 2.6 Demonstration (16)
- 2.7 Project Management (16)

## 3. Development Summary

- 3.0 Overview (19)
- 3.1 Objective 0 - Environment Investigations (19)
- 3.2 Objective 1 - The State Description Maintainer (21)
- 3.3 Objective 2 - The Dependency Map (27)
- 3.4 Objective 3 - The State Timeline (31)
- 3.5 Continuous Objective - Miscellaneous Engineering (32)
- 3.6 Demonstration (38)
  - 3.6.0 Overview
  - 3.6.1 The Ratio Model - Demonstrating SLT
  - 3.6.2 The Jugs Model - Demonstrating ST, DM
- 3.7 Project Management (44)

## 4. Further Considerations & Overview

4.0 Overview (47)

4.1 Project Conclusions (47)

4.2 Toward the Future (49)

4.3 Plans in motion (52)

4.4 Overall (55)

## 5. References (56)

## 6. Legal Considerations (58)



# 1. Introduction

## 1.0 Overview

This report details the proposals and outcomes of a Computer Science third year project entitled: Exploring Instruments for Online Empirical Modelling. A demonstration of the tools developed and the power they lend to modellers is exhibited. A summary of the direction that the tools to support Empirical Modelling may take as a result of the work carried out is also detailed.

## 1.1 Empirical Modelling & EDEN

Empirical Modelling (EM) is a novel approach to computer-based modelling that developed from research initiated in the early 1980s by Dr Meurig Beynon [1] from the Department of Computer Science at the University of Warwick. It encourages an open and experiential approach to programming that better accounts for the way that humans naturally experience the world. Where procedural code may be more easily optimised than an object orientated language, or functional approaches may make it easier to deal with problems inherent to machine paradigms such as infinite data; EM concerns itself with issues deeply rooted in areas such as software engineering, agent based approaches, specification, etc. in which the fundamental way of programming / modelling should be reconsidered as to avoid major issues that may occur throughout development e.g. a procedural program once written is incredibly difficult - if not impossible to join to another or integrate with a larger system unless careful consideration is taken long before development has started. An EM approach in contrast could easily allow for multiple models to be seamlessly linked to one and other without any such pre-consideration whatsoever. This consequently provides a potential platform for collaboration and experiential investigation in a way that

traditional approaches to programming do not, many applications in education of computing arise as a result.

Early activity within the EM research group led to the development of a new language called EDEN - an “Engine/Evaluator for DEfinitive Notations.” The first implementation of EDEN was by Edward Yung [2] in 1987 and a number of contributors have been leading the development of this tool ever since, most recently JS-EDEN - a Javascript implementation by Timothy Monks [3] in 2011. JS-EDEN (commonly referred to as EDEN) is the current flagship of the tools constructed to support EM. JS-EDEN has very recently been extended to support the use of plugins in a move to what is referred to as the “master version” by Dr Nicholas Pope [4]. This environment utilises context boxes on a webpage to simulate a graphical user environment much like in Microsoft Windows ©. JS-EDEN is currently an open source development project.

The primary motivation for EDEN as a programming language is to support the conceptual framework of Observables, Dependency and Agency. Observables are variables that represent what the modeller observes from the referent/domain. Dependency is the functional connection between observables. External agents may also exist and have power to change the state of the system or redefine observables.

The EDEN language allows for observables to be “defined” in terms of others, that is their state maintained and their value updated automatically by their definition e.g. ‘a is  $b + c$ ’ would mean that as the value of ‘b’ or ‘c’ are altered in some way, ‘a’ would be changed to meet the mathematical definition of ‘ $b+c$ ’. The power of the language is best realised through visualisation, for example the modeller could define a line in terms of the current position of a mouse over a canvas, to which various other observable constructs may functionally depend. The JS-EDEN canvas is designed specifically to exploit this functionality.

## 1.2 Current Issues & Motivations for Development

Through many decades of research, EM has made significant progress towards developing tools to fundamentally change the way that many computer oriented activities are approached. Although primarily aimed at relieving the traditional issues in software engineering, the methodologies give rise to many educational applications. Educational tools however, must be robust if they are to be taken-up in great number across wide expanses such as the recent MOOC movement [5] made by many universities. The current state of JS-EDEN is significantly less than what would be reasonably required to consider it a robust tool. Many aspects of the environment will need to be improved and in some cases completely re-engineered in order to achieve a satisfactory tool for stable modelling.

As the tool is currently only a prototype, it contains many bugs - some of which are fundamental flaws in design. The tool has also migrated over many versions of JS-EDEN ie: original, emile, master. Due to this, it is most likely that the application backend contains a significant proportion of redundant code. It is difficult for anyone to extend the tool without the source code presented in a way that different programmers can understand ie: code that adheres to high standards in programming practice.

## 2. Project Proposal

### 2.0 Overview

The development requirements of JS-EDEN have been received as a suggestion for a suitable third year computer science project. Under the supervision of Dr Meurig Beynon & Dr Jane Sinclair [13]. The uptaking student will aim to enhance the tool in a variety of ways.

As it will be difficult to quantise the amount of work required to advance the tool to an ideal state given the currently unknown condition of the code, as well as the fact that the tool could be potentially infinitely extended; a finite goal of three development extensions to the current master version of JS-EDEN has been agreed as adequate in terms of scope for a third year project. Parallel to these three extensions of the tool - routine maintenance with the potential requirement to completely re-engineer certain subsections of the application are necessary.

The current master version of JS-EDEN which will be used as a platform for development will incidentally be used as part of CS405 - Introduction to Empirical Modelling. The students registered on the module will be required to undergo a modelling study as part of their coursework. The students may or may not choose to use the current Master version of JS-EDEN for their coursework, although should they choose to do so it may be appropriate to take guidance from their feedback of using the tool, and if possible or necessary re-factor the application in such a way to assist them.

This section outlines the development plans for the project, it also clarifies development issues such as project management methodologies and requirements for

demonstration.

## 2.1 Objective 0 - Environment Investigations

Prior to development of the three finite objectives, it is necessary to evaluate the state of JS-EDEN. Focusing on the quality of the source code and the feasibility of conceptually sound extension and refactorisation, the investigation of some key areas of interest before any design or development work is carried out will likely result in efficiency savings throughout the project.

Some questions to consider are:

- How does the application work? Is it possible to map out the subsystems in a UML like fashion?
- To what extent does the code adhere to high standards of practice in programming? Is it easy to see what files in the backend refer to and contain code for? Is code well commented and appropriately named/structured?
- Has each contributor to the tool approached development in a similar way or are there obvious differences in programming styles? Have a range of different languages been utilised for different, or the same systems within the application?
- Are there significant levels of bugs in the application? What bugs exist? Are they easily addressable? Will they significantly hinder efforts for development in this project?
- Has the environment been constructed and extended in a conceptually sound way, or have workarounds been necessary? Do separate systems within the application communicate well with each other? Is the environment extensible?

The three development extensions require that an internal representation of the state

and ideally some sort of application programmer interface (API) for the tool is found or if necessary sufficiently constructed. Without this, development for the tool is likely to require much more time and result in redundancy.

## **2.2 Objective 1 - The State Description Maintainer**

The first development objective is to achieve a tool that will better represent to the user the effective state of the modelling environment. Currently the symbol list fails to represent much about the internal state of the system such as what symbols are contributing to the definition of other symbols or even what the definition of symbols are - if they are defined. It is very useful for debugging purposes as well as conceptual sanity checks for the modeller to have access to this information.

The tool should give rise to the functionality of being able to export a succinct description of the model. If this is achieved it will enable the user access to a script that will return the environment to the exact state at the time of export. This will consequently provide a basis for recording the internal state of the system in a conceptually sound way - providing it does not already exist.

## **2.3 Objective 2 - The Dependency Map**

The second development objective is to implement a modelling assistance tool which has been previously implemented in various legacy versions of EDEN. The tool in question has been commonly referred to as a "Dependency Modelling Tool." The tool is to be a directed graph which visually represents the internal functional dependency between observables. For example suppose the modelling state comprised three observables 'a', 'b', and 'c' of which 'a' and 'b' were assigned to be the values 5 and 6 respectively and 'c' was defined to be 'a + b': the representing graph of the state would

display three nodes labelled 'a', 'b' and 'c' with two directed edges: one from 'a' to 'c' and the other from 'b' to 'c'. These edges represent that "c is dependent on the values of both a and b," or from a conceptually alternative standpoint, that "a and b both contribute to the value of c."

Previous incarnations of this tool have allowed users to rearrange the layout of the individual nodes as they please. This feature is often considered paramount to the functionality of the tool as automatic layout algorithms do not take into account what the user wants to investigate by using it. Also in many models the complexity of the connections between the nodes is far too great to usefully interpret without rearrangement. This suggests that functionality to remove groups of nodes from the displayed selection may also be useful.

## **2.4 Objective 3 - The State Timeline**

The third development objective emerges from the aspirations of Objective 1: to construct the facility to enable the modeller to reverse the state of the environment to a previously specified configuration. This may help the user to explore various routes of investigation independently without a great deal of manual respecification. For example: in the game 'Sudoku' the player must sometimes make a guess from a subset of states as to which direction to take. If they were able to save the state and recall it, it would alleviate having to undo sequences of changes such as previous moves, manually. If the model is somehow broken, the user would be able to make use of the tool to reverse to a previously functioning state. Providing enough records were made, this tool could significantly help identify bugs or conceptual misunderstandings. In addition, if this tool could be harnessed in some automatic way, it may provide some variation of short term version control functionality to the modeller.

## **2.5 Continuous Objective - Miscellaneous Engineering**

A less well defined objective of the project is to complete general maintenance and re-engineer minor miscellaneous subcomponents within the application environment that do not function in a correct or ideal way. The motivations of this objective are to enhance JS-EDEN for the benefit of its users and future users, so that they are able to develop within the environment efficiently and utilise the tools available without encountering errors.

There is no formal state of completion for an objective of this nature, however there should be sufficient evidence that changes made to the application better support users in their modelling efforts.

## **2.6 Demonstration**

With the exception of Objective 0: the objectives outlined in this section are required to be demonstrated individually through already available models or models constructed specifically to outline the effectiveness of the respective tool. In the case of the Continuous Objective, there should be reasonable evidence presented that the changes made are beneficial to modellers or future developers of the environment, although this is not necessary of every individual change made.

## **2.7 Project Management**

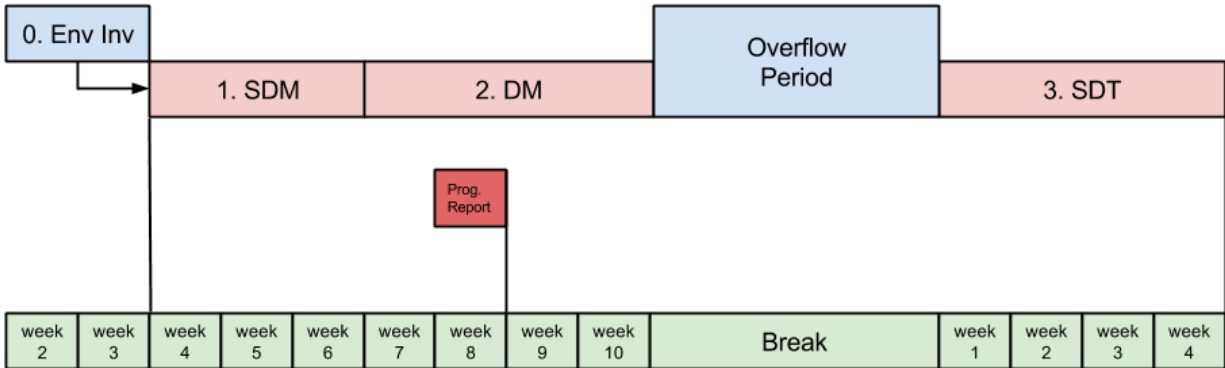
### **2.7.0 Overview**

The following section outlines what management methodologies should be used to



guide the approach to the project.

### 2.7.1 Approach



(Figure 1 - Gantt Chart outlining estimated project deadlines)

The three development objectives for this project will be sequentially completed. Initial investigations have been allocated approximately 2 weeks prior to the development of the first tool - the State Description Maintainer. Following these initial investigations the development strategy for the first objective may be altered to take into account issues discovered. As the first experiences of development for the environment commence it is anticipated that further timetable altering discoveries may be made, setbacks in the condition of the code or lack of required skill to develop may be experienced. If necessary the allocated period of 3 weeks for this first development activity may be extended up to 1 week. The second development objective may commence without completion of the first, should time run short. The overflow period between term 1 and term 2 is to be used to finish any incomplete tasks and bring the first development objective up to a high standard with the extra experience gained from completion of the second objective. As the second term commences, the final development objective should build on the outcomes of the first objective, and draw on experiences gained from development and exploration from all objectives (including the miscellaneous development for the CS405 students) to produce a tool of high

quality.

In general, the guideline timetable should only be used as measure of expected progress as it cannot possibly take into account unexpected discoveries throughout development.

### **2.7.2 Code Management**

Although there is no requirement of version control technology for the source code, intermediate versions will be zipped and uploaded to the department servers for use by the CS405 students and for periodic feedback from the project supervisor. Should anything major happen to the code during development, it will be possible to recall the code from the server. While developments are being made to independent files, copies of the old files should be kept in a nearby directory for reference or reversion if something is discovered to be broken.

### **2.7.3 Periodic Consultation**

The developer should consult with the project supervisor(s) on at least a weekly basis in order to adapt to feedback and if necessary explore paths of miscellaneous development suggested by the project supervisor or CS405 students indirectly.

## 3. Development Summary

### 3.0 Overview

An account of the development for each of the proposals outlined in the previous section are detailed individually below. Issues with development including how they were dealt with are explored where applicable and wider issues such as the suitability to the application of including certain functionality are also noted.

### 3.1 Objective 0 - Environment Investigations

However sound in conceptual design the protocol which maintains the observable update mechanics of the environment are, they are not well implemented. It is very difficult for an onlooking programmer to understand how this subsystem works. Variable names are well chosen and accurately reflect their functionality in most cases, however comments are minimal if existent. There also appears to be many workarounds which are undocumented. The combination of these practices mean that it will be difficult to make alterations to the application through difficulty of understanding, and satisfy the requirement to minimise unintended side effects.

There are considerable quantities of code which appear to be redundant, sometimes relating to legacy functionality which has been replaced entirely. Large blocks of code are commented out, sometime alongside useful comments such as: “//this doesn’t work.” There also appears to be a fair amount of code in an unfinished state: A section relating to the serialisation of state for record and recall was evidently an ambition unrealised.

The server contains many files which are not linked to the application. It also contains many files that are linked, but do not make any contribution to functionality. It also contains files which are related, but exist in different directories. The organisation of the backend requires improvement and refactoring but it is noted that due to the nature of the application, this is not a priority.

The user interface (UI) for the context boxes (The primary extension from original JS-EDEN to the master version) have utilised an imported library which heavily relies on JQuery. As with most imported libraries; heavy redundancy is also acquired - A particular issue for extension and refactorisation. CSS which has been imported in bulk introduces chaos with regards to maintaining conceptual integrity when alterations are desired. The imported library of JQuery CSS is also obfuscated. Individual changes to the user interface, which include the need to remove bugs is therefore infeasible.

Another side effect of utilising JQuery is the conflict it causes with Javascript. From a Javascript perspective: JQuery can be considered a hack. Similarly vice versa. By mixing these approaches an effective situation akin to two autonomous agents attempting to simultaneously maintain an environment using different languages arises. All is fine while the environment is stable and there are no plans to extend the application, but as soon as said extensions are desired, the implementing programmer is posed with a confounding problem: having to make sense of what is going on. Such approaches should surely be actively discouraged in the case of software being “open and experiential” such as this.

Investigations have however proven fruitful in successfully locating a method of accessing and manipulating the database of symbols in the environment - a fundamental requirement for development. The ability to query observable properties through the use of Javascript function ‘root.lookup’ was identified. This will return the Symbol that the Javascript environment uses to maintain the properties and state of

the observable within the system. A curious side effect of using this method to query observable symbols however is that if symbol in questions does not exist, it will be created by invocation of this lookup method - notoriously unanticipated functionality which has resulted in many hours of confusion. If this seemingly innocent function is used to check to see whether if a symbol exists, the programmer will be bewildered until they discover this rogue functionality.

In conclusion: the internal state of the application is detrimental to all efforts of extension and outside understanding. It may be considerably difficult for anyone else to make some of the changes proposed in this project as a result. If the changes are standalone additions however, the bulk of these issues may be not be applicable as there will not be significant interaction with existing code. Bug fixing on the other hand will require copious amounts of said interaction - and by extension difficulty.

### **3.2 Objective 1 - The State Description Maintainer**

The first mandatory development objective was to achieve a tool that will better represent to the user the effective state of the modelling environment, with ambition to be able to export a script which succinctly represents that state to be reinterpreted at the whim of the user.

The user interface for this tool was to be modelled on the symbol list. The symbol list displays the name of all symbols alongside their current value. As the symbols change value in the eden environment, their updates are automatically propagated to the display of this tool. There was originally intended functionality of being able to edit the value / definition of the observable through this symbol list, however it was not realised. Indeed there is no way using this tool to even inspect the definition of symbols.

There has been discussion as to whether the automatic update of the symbols is

desirable functionality. It is sometimes the case that modellers wish to observe state-to-state updates of particular symbols, even with a manual refresh however, the modeller will only be able to observe a start to finish jump in state, discarding any intermediate values. As it is already possible to use EDEN functions to slow the interpreter to achieve this functionality it has been concluded that it is best for the symbol list to update as quickly as possible. The new tool will be presented as a database rather than a list of key value pairs.

```
> root.lookup("z");
▼ Symbol {context: Folder, name: "/z", definition: function, eden_definition: "z is y + x", cached_value: 12..}
  cached_value: 12
  ▶ context: Folder
  ▶ definition: function (context) { return o_y.value() + o_x.value(); }
  ▶ dependencies: Object
    eden_definition: "z is y + x"
    last_modified_by: "unknown"
    name: "/z"
  ▼ observees: Object
    ▶ __proto__: Object
  ▼ observers: Object
    ▶ __proto__: Object
  ▶ subscribers: Object
    up_to_date: true
  ▶ __proto__: Symbol
```

(Figure 2 - The root.lookup() function in a chrome console)


Through the use of the root.lookup function, the programmer has access to all of the information stored through the internal representation of the symbol. Instead of just extracting the name and the value of each symbol as the symbol list does, the new maintainer should take the remainder of the information also, this includes the ‘observees’ and ‘observers’ - which symbols are contributing to the definition of, and which are being defined in terms of the symbol in question. This information is useful to the modeller in many ways, specifically for the motivations behind the next tool.

With the above design objectives in mind, the tool’s implementation was the next target. It was at this point that in-depth understanding of how plugins worked was necessary. This task was non-trivial. The knowledge required to build and integrate a plugin would ideally be taken from a readme prepared by the implementing programmer or drawn upon from observed technique taken from existing plugins. With

the absence of any such documentation, and a confounding scramble of JQuery-Javascript implementation, this approach was infeasible. Instead a trial and error approach with basic understanding of HTML, CSS and Javascript was taken. Conveniently, the implementing programmer included a basic-HTML plugin that provided considerable insight into how the plugin would be assimilated by the environment, without obscuring opportunity for fundamental understanding of functionality with feature dependent code - there is reason to believe this plugin was included specifically for this purpose.

From a blank plugin to something with database like functionality and look, was a journey void of conceptual cohesion; a pure hacking adventure - but one necessary to understand the environment. The first prototype of this tool was unsatisfactory in many aspects, for now the tool had basic desired functionality, however many aspects needed attention. The update required manual intervention, the layout was far less efficient than the original symbol list, the ability to filter observables, actions, functions etc was sufficiently confusing from a user interface point of view etc. Power was added to the overall tool however. One could check the definitions of symbols using a graphical approach instead of hacking the internals, and more understanding of the mechanics of the environment was acquired. Refactoring this tool in the future would be significantly easier with the skills I had gained.

The advent of this plugin - as anticipated - gave rise to the ability to export a complete state script. With some simple logical transformations the code to output this database could be altered to output a script that could be reinterpreted using the input window.



```
Plain HTML View [ExportModelScript]

##Auto-Generated Script of Model by JS-Eden J-version

##Auto calculation is turned off to until the model has been fully loaded
autocalc = 0;

##Observable Assignments:

Error = 0;
base = 200;
capA = 5;
capB = 7;
contentA = 0;
contentB = 0;
jugwidth = 60;
left = 50;
linewidth = 0;
scale = 20;
spacing = 30;
target = 1;
updating = 0;
viscosity = 100;
widthA = 5;
widthB = 5;

##Observable Definitions:

Afull is capA==contentA;
Bfull is capB==contentB;
but1 is Button("but1",menu[1], 50, base+80, valid1);
but2 is Button("but2",menu[2], 50+70, base+80, valid2);
```

(Figure 3 - Script Exporter Viewer generating a script for Jugs)

In order to achieve this functionality, the state of each symbol was recorded, the symbols were then separated into those which existed immediately as the application was launched: the environment symbols, and those which were specified as part of a user oriented modelling activity: the user specified symbols. The environment symbols did not need to be recorded as they would exist on refresh of the application anyway. The state of them however, would be forgotten. This is desirable as the tool has not designed to reproduce the exact layout of the windows etc and other features of the user interface associated with environment symbols, just the information related to the construction undertaken by the modeller. This way, environment dependent bugs such as screen size dependent layout will not occur as a result of the tool. eg: If the tool recorded the position and size of the windows (environment symbols), and the application called the exported script on a machine with a smaller/larger screen the layout would appear cluttered / overstretched. In some cases the windows may be unrecoverable. There are various other reasons as to why the record of environment symbols are undesirable. After identifying the user defined symbols, it was necessary



to organise them into groups of: Observable (with definition), Observable (without definition ie. state only), Function and Procedure. It was important to separate Observables (with state only) from all of the other groups. Once the state only Observables were identified, for each: an interpretable string was generated of the form: “observableName = associatedState;” The Observable symbols which were defined could then be utilised in the same way: “observableName is associatedDefinition;” ignoring the state, as it was unnecessary. The Function and Procedure definitions could be immediately identified and included in the export script in a similar fashion. A visually appealing script could then be generated by arranging these interpretable strings with appropriate explanatory comments where necessary. This procedure could be undertaken at every state the user desired. Automation would be possible, but space requirements are a likely subject of concern before planned implementation.

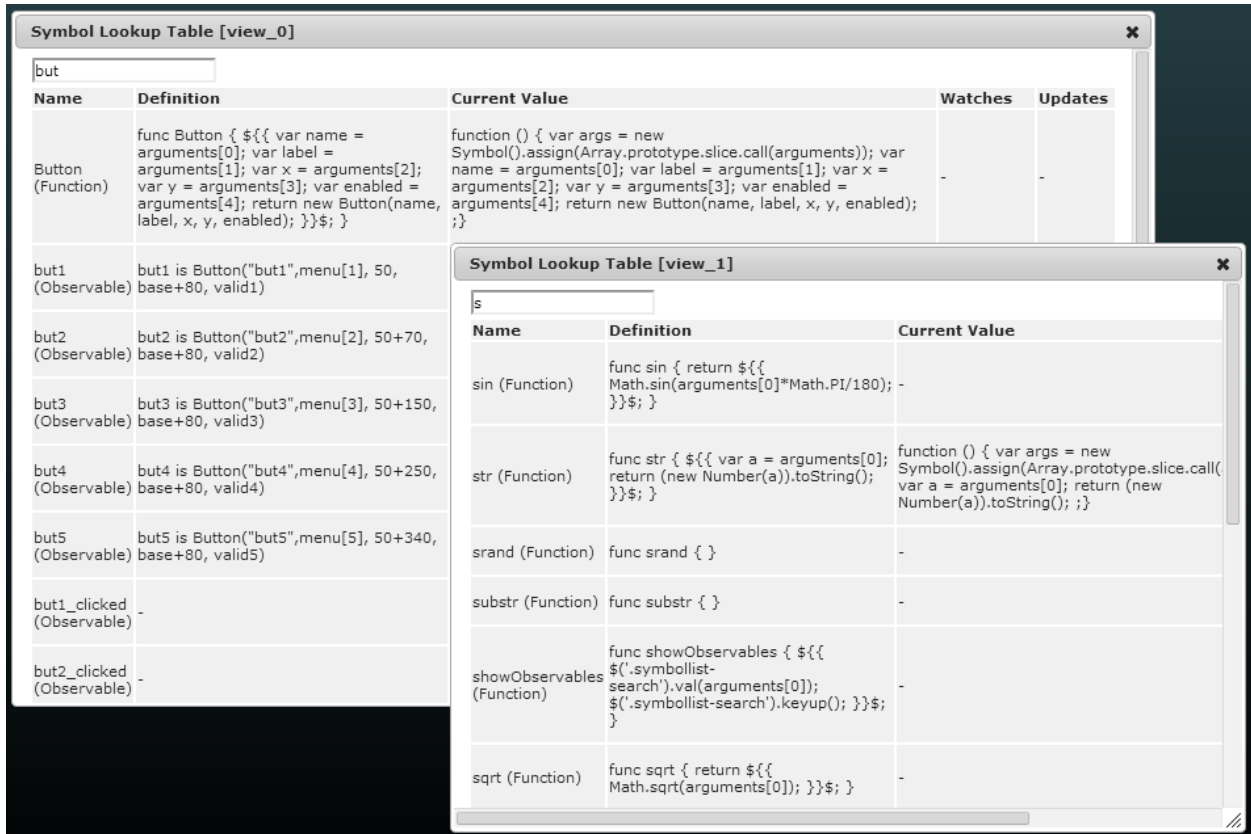
The screenshot shows the State Description Maintainer (SDM) window. It has a title bar 'State Description Maintainer [SDM]' and a close button. Below the title bar are tabs for 'System' and 'Model', with 'Model' selected. Under 'Model', there are sub-tabs for 'Observables', 'Actions', and 'Functions', with 'Observables' selected. To the right of these tabs are a 'Refresh' button, a 'regex' input field, and an 'Export Model Script' button. The main area contains a table with the following columns: Name, Definition, Current Value, Dependant On, and Influences.

Name	Definition	Current Value	Dependant On	Influences
Afull	Afull is capA==contentA	false	capA, contentA	valid1
Bfull	Bfull is capB==contentB	false	capB, contentB	valid2
Error	-	0	-	status
base	-	200	-	jugA_left, jugA_right, jugA_base, jugB_left, jugB_right, jugB_base, jugA_water, jugB_water, statuslabel, but1, but2, but3, but4, but5
but1	but1 is Button("but1",menu[1], 50, base+80, valid1)	Button("but1", "1:Fill A", 50, 280, true)	Button, menu, base, valid1	picture
but1_clicked	-	-	-	-
but2	but2 is Button("but2",menu[2], 50+70, base+80, valid2)	Button("but2", "2:Fill B", 120, 280, true)	Button, menu, base, valid2	picture
but2_clicked	-	-	-	-
but3	but3 is Button("but3",menu[3], 50+150,	Button("but3", "3:Empty A", 200,	Button, menu,	picture

(Figure 4 - The State Description Maintainer and Script Exporter as a Single Plugin)

All of the aforementioned functionality including the database of symbol details was packaged as a single plugin. Although the power of this functionality was to be realised to a much greater extent later with the final development objective, it alone meant that the user no longer needs to worry about maintaining a separate text document with the code to continually resubmit in. It is now stored internally and only has to be recalled.

After gaining extensive experience with the environment and with Javascript through miscellaneous developments under the continuous objective, this tool was refactored into two separate, cleaner tools: The symbol lookup table consisting of the database of symbols, and the script exporter. The newer version of each of these plugins supports multiple independent views, meaning that multiple symbol lookup tables each displaying a separate set of observables may be open and functional separately and simultaneously.



(Figure 5 - Multiple views of the Symbol Lookup Table functioning concurrently)

### 3.3 Objective 2 - The Dependency Map

The goal of this development objective was to realise a JS-EDEN master version of the dependency modelling tool (DMT) - a directed graph which displayed the internal associations between symbols in the EDEN environment.

There have been many previous incarnations of this tool, each with slightly different issues. One version had issues with automatic layout of nodes, others were standalone visualisation packages designed for EDEN, but functionally separate. Due to the nature of the master version, it would be appropriate to include this tool as a plugin with direct access to the observables, therefore would have a dynamically linked nature. The issue of graph layout however will still need to be addressed.

In a model with 100+ observable definitions each with multiple connections to other, it is extremely difficult for the user or even the system to present them in a visually satisfying way. One version of this tool used in CS405 labs would present the user with a graph on a blank canvas, each node representing an observable, somehow equidistantly distributed, and leave the user to manually rearrange them to their satisfaction. There is merit in this - each user may wish to use the graph to investigate different things, thus their respective layout requirements differ; adjusting the layout of nodes to how they internally construe their model is useful to each modeller individually. To initially throw a mess at the modeller however, could be improved.

Another version of this tool [6] presented the user with multiple layout methods. This may be considered beyond the scope of this development objective. To utilise an algorithm to draw the graph in some optimal fashion to begin with and then add functionality to allow the user to rearrange how they saw fit, would be enough to achieve a good balance of effective interpretation and user control.

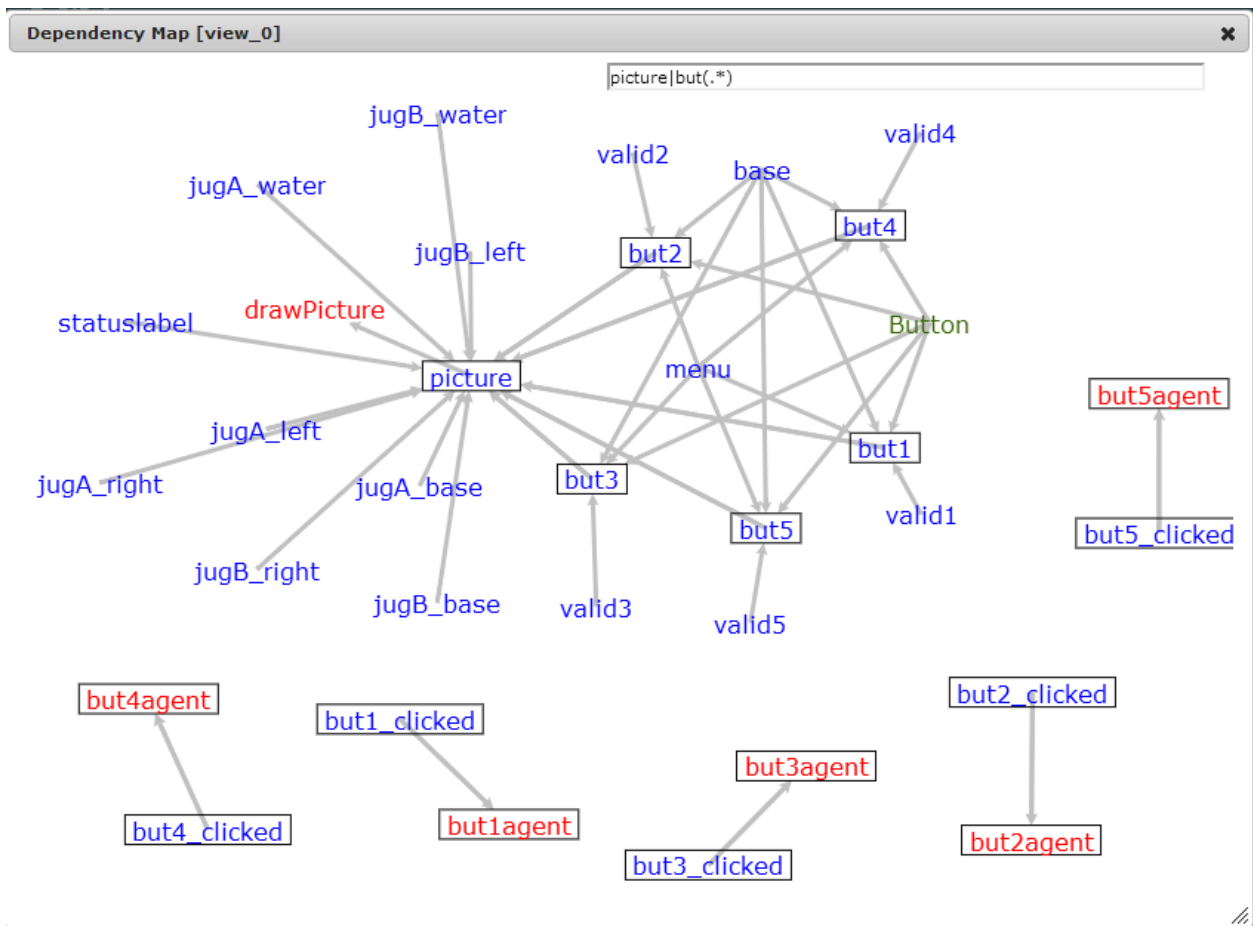
It was appropriate to investigate what graph layout algorithms existed. A particular algorithm was quickly discovered; 'Force Directed' graph drawing was described by wikipedia as a class of algorithms for drawing graphs in an aesthetically pleasing ways. After much deliberation and investigation of various other solutions, the most appropriate method was deemed to be this 'Force Directed Layout'. In the midst of researching examples of equations for that would emulate such functionality, a public Javascript library for exactly that was discovered: Springey.js. All that was required now, was to link this library to JS-EDEN in such a way that the internal representation of state (conveniently discovered in previous requirements of this project) could be directly represented in the graph. This has been achieved with fantastic results. The smaller forests are pushed to the edge of the graph due to their connections having a smaller combined weight than the larger more complex forest. Leaf nodes are pushed

toward the edge of the complex forest for the same reason. Functionality has been manually added in order to be able to move the nodes from their stable state to a user defined position, once the node has been moved, it will not respond to the force directed mechanics which define the initial layout, thus there is initial Force Directed Layout to which the system will eventually rest, then after user defined redirected layout of specific nodes, automatic adjustment to which the system will again rest. If all nodes have been individually respecified following the initial layout, force directed adjustment will no longer take place until the graph changes, which will reset the system.

Manually defining the mechanics of the force directed layout would have removed dependency on an external library, however would likely have resulted in an implementation of lower quality. The code for the library is completely un-obfuscated, well commented and partially rewritable if necessary, It was designed specifically for this purpose and the author has published basic tutorials detailing how to customise the library to suit various applications' individual requirements.

A regular expression (regex) search box has been included in order to allow the modeller to restrict the display to exactly the connections that they are interested in. With nothing in the search box, no graph will appear, as the user begins to type, the graph is updated with a different graph with every keystroke - as long as a change to the structure of the graph is made. This ensures responsiveness from the application. Only the symbols captured by the regex or those which have direct connection via an inward or outward edge will be displayed on the canvas.

In addition to the force layout, visual customisation has been added to the graph as to distinguish the separate symbol types to the user. The blue correspond to the observables, red to the agent procedures, and green to the functions. Also each symbol captured by the regex search functionality is highlighted with a box.



(Figure 6 - Dependency Map)

In the example above we can see the buttons but1, but2 etc each call Button() in their definition, so an edge is directed from the button observables to the function Button. The regex used to display this graph is: `picture|(but.*)` meaning: picture or any word beginning with “but”. As we can see with the boxes around certain nodes “but1” has been captured by the regex, but “valid1” one has been displayed because it has a direct edge to “but1”. Although “but1clicked” and “but1agent” have both been captured by the regex, there are no edges aside from the one between them, as no functional relations in the application aside from this single edge exist.

This plugin has overall been an incredible success. It has some minor non-ideal

features such as the refresh of the visualisation requiring a “mouse-over” the canvas, but overall a result of a historically significant feature of EDEN implemented to a high standard in the master version of JS-EDEN.

Both the Symbol Lookup Table and the Dependency Map failed to meet their requirement for dynamic update. For this to be achieved, it is necessary to include a call to update deep within the application at a suitable location where changes to the state of the environment can be detected and verified. The lack of recognisable structure within the code prevented this. It would be possible to initiate the update of the plugin from within each plugin as the canvas currently does. This solution however is unscalable from a performance aspect.

### **3.4 Objective 3 - The State Timeline**

The goal of this development objective was to build on the aspirations of the first objectives outcomes. The first objective lead to the creation of a script-exporting plugin that allowed to user to extract a succinct representation of the internal model for resubmission. Now that this is possible, an extra level of automation can be added to enrich the functionality from the point of view of the user.

The plugin developed for this objective is named “The State Timeline”. It not only allows the users to record the state of the model at the touch of a button and restore the state recorded at the touch of another, but maintains a list of the states recorded allowing the modeller to jump between any of them at their whim.

The tool utilises a minimal HTML context and appends a list item to the context each time the modeller records a state. The list item contains a restore and delete button allowing the user to recall the state of the modelling environment stored by that entry and delete the entry from the list respectively.

With the additional functionality of multiple plugin support, the modeller is able to make separate timelines over multiple views of this tool. If the modeller was exploring which states the model was broken for, they may want to record it to inspect later, or, if they have broken something without realising they may need to compare the state of the environment with a working state in order to locate the problem - it may be convenient to separate these into different views. For similar reasons the ability to name the states upon recording was included. This allows the modeller to clearly distinguish which state is which, and assure themselves of the reasons that recorded it in the first place.

Having gained so much experience from the continuous development objective and the previous plugins, there were very few technical issues encountered during the development of this tool, those that were encountered were minor indeed and very briefly overcome.

### **3.5 Continuous Objective - Miscellaneous Engineering**

The application uses a combination of HTML5 canvas primitives such as circle, rectangle, line etc. and native HTML elements such as button, text etc. to draw items to the canvas. While it is true that many of the HTML elements could be implemented using the draw methods of canvas, it was seen as a 'shortcut' to simply make use the functionality that already existed in browsers, indeed to implement button using rectangles and touch events is not as trivial as appending an HTML button to the div containing the canvas. Unfortunately, there are a great many unforeseen issues that arise and plague the development of the application when any such shortcuts are preferred to conceptually sound approaches.

The first issue with appending HTML elements to a canvas, is that HTML elements by

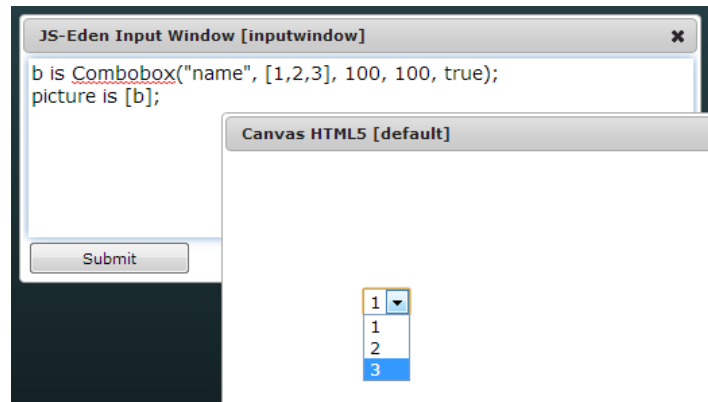


nature are not 'drawn' in the same way that canvas primitives are, they are appended. The technique to draw primitives is to call a draw function on the object representation of said primitive which will utilise the internal coordinates stored in the object: ie if I adjusted my rectangle using EDEN by submitting: "myRectangX1 = 15;" each time the draw method is called it will find this value and draw the rectangle with the updated coordinates. HTML elements however are appended to the div, and not continually redrawn, so a separate process entirely is required to maintain these constructs - check that they are updated and if not, delete the element and create a new one with the updated parameters then re append. By adopting this approach we have added a layer of complexity to our application and created twice as much work in order to maintain both render loops, more so for a unfamiliar developer such as myself to then attempt to understand. Many users have required / desired functionality that the application does not yet implement such as an input box. Due to the level of understanding required of the HTML render loop, rather than implement the construct in Javascript, it has been common practice for users to hack into HTML and write it themselves. Even developers who contribute to the application - competent in HTML, struggle to understand the methods for maintaining programmed constructs using this loop.

With respect to the original implementer, it is not trivial to solve the problems highlighted above, and due to the time constraints of the project, reimplementing of all constructs using primitives and events is not within the scope of this project. Further discussion regarding steps to resolve this issue in the future is detailed in section 4.2. For now, implementing many of the widely "hacked" constructs such as input box, radio buttons etc using the appending approach would have to suffice.

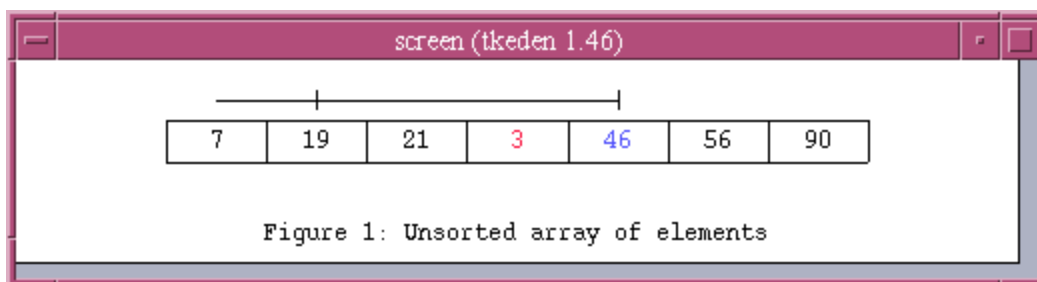
Most of the HTML elements were found to be suffering from the picture removal bug: after specifying an observable which uses an HTML element ie: "myButton is Button("hello", 100, 100, true)" and adding the observable to the picture ie: "picture is

[myButton];” the element did not reappear when resubmitted following removal ie: “picture is []; picture is [myButton]”. This was identified and solved by correcting a line of code that appended the element to the canvas.



(Figure 7 - Combobox on a canvas)

Many of the HTML elements were also found to be suffering from a different bug where the state of the element is lost following a resize of the canvas. Resizing the canvas forces a redraw, as the elements state is destroyed with the element, and re-drawing elements requires that they be destroyed and recreated, they were re-appended as new elements with a default state. ie: if a combobox’s final option was selected, then the canvas resized, it would be redrawn with its default option selected. Fortunately the state selected was remembered by the observable representing its state, so the element merely had to be forced back into the correct state following a redraw.



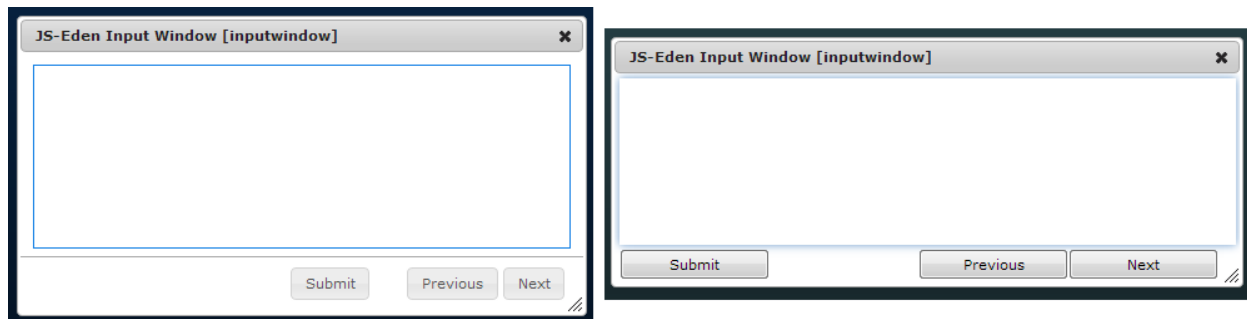
(Figure 8 - A 1998 version of the Beynon Bubblesort in tkeden)

The Bubble-Sort model by Meurig Beynon (2007) [7], when ported to the master version of JS-EDEN (originally written in traditional EDEN) highlighted some functional bugs by demonstrating some interesting and non-intended behaviour. The particular problem of skipping an element in the list when starting the next iteration (pass through the list it was bubble sorting) was highlighted. The solution was found following rigorous inspection of the Div Element drawing code, which was seen to be assigning a value to an observable two times unnecessarily, triggering an agent procedure twice instead of the intended once. The simple removal of the duplicated line solved the problem.

The user interface for the master version of JS-EDEN has been identified as causing a wide range of undesirable functionality. eg: the windows that are draggable around the screen cannot be partially dragged off of any edge including the top and bottom, meaning that if the user had more than a few windows open, any marginally larger window would prove obstructive to the smaller windows, this has on many occasions lead to the input window being lost behind the canvas, requiring either a close and open, or a resize and a re-resize once the hiding window had been recovered. The windows are also unable to be closed. Once you are finished with a view, you are unable to remove it from the application without hacking the Javascript. You are able to minimise windows, but this functionality is much more like closing them. ie: if you were to click the [x] in the top right corner of the window, the window would be minimised, but to the menu where you originally created it from - removing it from ease of recall and sufficiently confusing the modeller into believing that they had removed it entirely, as is often the observed consequence that a new view is created to replace a mistakenly minimised one.

The ambition to resolve these issues was entirely quashed upon discovering obfuscated JQuery and JQuery generated CSS. Altering parts of the CSS which appeared to be responsible for various undesirable features often resulted in no change at all as multiple class assignments to each element overwrote single

redefinitions multiple times, it was concluded therefore infeasible to alter altogether. The user interface imported from this JQuery library became a particularly serious issue when attempt to implement an EDEN function to target elements within the user interface was made. The “showObservables” function is used to make a particular regular expression appear in the search box of various windows, this is useful to the modeller in guiding another user to a particular feature. Although this function was successfully implemented, a similar feature for the input window was desired by a “copyToInput” function. The input window in particular makes use of JQuery input methods which appear to contradict any attempt to penetrate using traditional programming methods such as that made by the DOM. The behaviour exhibited by the input window appeared to be destroying and recreating some part of the input field with each keystroke. When targeted using standard DOM methods to inject a string into the field ie: `element.value = mytext`, the function appeared to work, but then reverted as soon as other user intervention such as the mouse clicking on a section of the copied text was made. This behaviour could not be reasonably attributed to any known standard programming behaviour. It was attributed therefore to unknown JQuery methods.



(Figure 9 - The Old Input Window (Left) vs the New Input Window (Right))

Approaches were taken to redesign the input window in the same way that the development objectives made plugins from blank plugin templates. Without making use of the imported JQuery and CSS libraries. The input window was redesigned from

scratch using standard HTML and CSS approaches. The result was a conceptually sound input window that was able to be manipulated using standard approaches enabled by the Document Object Model (DOM) such as `element.value = mytext`. This time the function worked exactly as intended, although the window itself still suffered from the same user interface issues as all of the other containing context boxes as it still had to utilise the imported libraries for the basic plugin outline. This development was to highlight the importance of standard approaches to programming over JQuery methods - Quick controls were added to the input window for the users benefit: Ctrl+Enter for submit, Ctrl+left arrow for previous entry etc.

On inspection of the code which is responsible for drawing to the canvas, it was noted that the “set interval” approach was being utilised - `setInterval` is a general method in Javascript for recalling a function after a specified amount of time has passed. The code in this case was calling the render method for canvas every so many milliseconds. This approach to canvas drawing has long since been deprecated and replaced by `requestAnimationFrame()`. Reasons for this include that each machine will take different amounts of time to draw items to the canvas depending on its graphics rendering capabilities, if the amount of time specified is fixed then some browsers will render “jumpy” frames and others will call the render again before the previous call has been completed. `requestAnimationFrame` ensures that each browser calls the method synchronously as fast as it possibly can without causing garbage. In order to ensure that JS-EDEN functions smoothly this dependency on `setInterval` should be removed. After refactoring the canvas code to achieve this, the speedup was so significant the HTML elements were rendered unusable. It was impossible to interact with them as the code to redraw / check whether they should be removed or updated was locking the mouse input quicker than any reasonable interaction could be made. Thus due to the HTML elements this advance is not possible.

A number of other minor contributions have been made to the application. Pixel has

been implemented at the request of one of the CS405 students, textarea has also been implemented at the request of another. Radio buttons have been implemented at the request of Meurig, Sliders have been reimplemented using standard HTML instead of JQuery UI. Numerous alterations to the symbol list including text highlighting depending on whether an observable is defined or assigned have also been made. HTML interpretation has been suppressed in the symbol list, symbol lookup table and interpreter history, this was to prevent tables, buttons and input fields etc appearing where simply the text should have appeared eg: `<input>...etc`. The input box was implemented with the assistance of Hui Zhu [8], who had a previous implementation with some bugs. The bugs were mostly due to failings copied from other HTML element implementations.

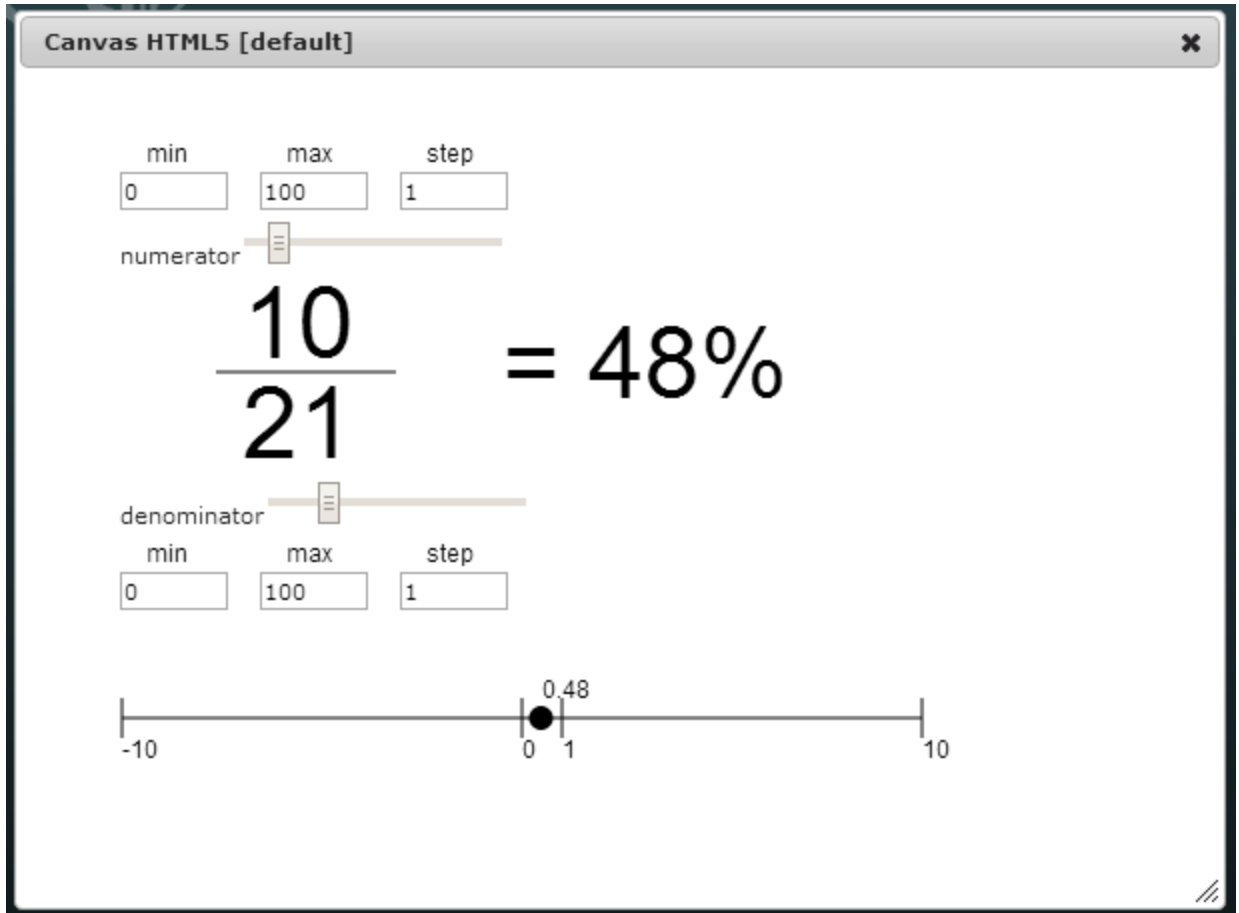
## **3.6 Demonstration**

### **3.6.0 Overview**

This section will demonstrate how the changes made by this project have improved the state of the JS-EDEN environment.

#### **3.6.1 The Ratio Model - Demonstrating the Symbol Lookup Table**

The first model to consider is the ratio model. The ratio model was designed by myself during term 1 as part of an educational module. It facilitates the exploration, using some sliders and input boxes, how fractions, percentages and ratio are linked. Particular development / re-engineering pertaining to or inspired by individual models has been included.



(Figure 10 - The Ratio Model)

The user is directed to shift two slider bars which control the value of the numerator and denominator. By default, the values that the slider bars range over are both 0 to 100 in steps of 1, although the user can change this to further experiment with the model via the input boxes respectively labelled min, max, step, for each numerator and denominator. As the user adjusts the values using the slider bars, the value illustrated by the larger text alters appropriately, as does the calculated percentage. A number line also exists at the bottom, which illustrates the overall value of the fraction, the user will hopefully refer to this in order to understand the connection between fractions and overall values through experimentation with the slider bars.

The slider bars, as part of the reimplementation by this project, now conform to the

HTML standard. The previous slider bars were implemented using JQuery UI which meant that they were visually larger and inflexible to the programmer. They also suffered from the bugs highlighted in an earlier section on miscellaneous development. As they were not implemented using standard Javascript, the discovery of a JQuery solution to a problem that was solved in other cases using Javascript was required. Instead of attempting to patch what I considered to be an already hacked solution, I decided to implement the sliders using standard Javascript. As a result of this change, we have acquired historical functionality from standard HTML elements, and eased the maintenance task for future generations.

The text boxes have been re-engineered to exist separately of the residing text eg. previously “min” would sit to the left of its respective input box inflexible to alteration or otherwise specification. In order to place the text above the box as displayed in this model, the modeller would have had to hack the HTML generated by the EDEN code, this would have a number of side effects detrimental to the motivations of the environment. Properties that text contains such as font, colour, size, etc. were also previously unspecifiable. Thus separating the text from the box altogether has enabled greater flexibility, and removed redundancy. The same would ideally be true of the slider bars, alas this functionality has not yet been implemented.

Text has been given a ‘size’ argument, which has historically confounded developers such as Matt Cranham who failed to produce an explanation as to why it did not work, despite being such rudimentary functionality. In order to alter the size of text drawn to the canvas, a font family name must be provided. This discovery was made only recently. A change has now been made to allow the desired size of the font to be specified by the modeller without requirement of a font family, by appending a permanent font family. The text which denotes the numerator and denominator, as well as the calculated percentage in the model constitutes demonstration of this alteration.



The modeller may observe that the precision of the percentage output is less than desirable. In fact it is misleading to a learner.  $10 / 21$  is not 48% - a decimal place needs to be included to re-assure any learners that this is indeed incorrect. In order to edit the text, the modeller first requires the name of the the observable symbol that represents the displayed text on the canvas. They may wish to go about this in any number of ways. If the model contained reasonably few symbols such as this one, a quick skim down the symbol list would reveal 'percentageText = Text(230, 160, '= 48%', "black")' - although not immediately obvious, this is indeed the symbol which requires alteration. Now that we know the symbol which must be altered, we must find its definition. This is impossible in the symbol list, so the use of either the Symbol Lookup Table, or the Script Generator (both products of the first development task) is necessary. Either tool will provide the modeller with the following effective definition: percentageText is Text("= " + str(roundPrecision(percentage, 0)) + "%", 230, 160, "black"); The function call roundPrecision will be the culprit. A similar lookup, or intuitive guess will point you to replacing the 0 in the call to the amount of decimal places you want to round the percentage to. Job done. Without a user friendly way of accessing the definition of symbols, this redefinition would not have been feasible. Paramount functionality to the environment has been added with the development of this tool.

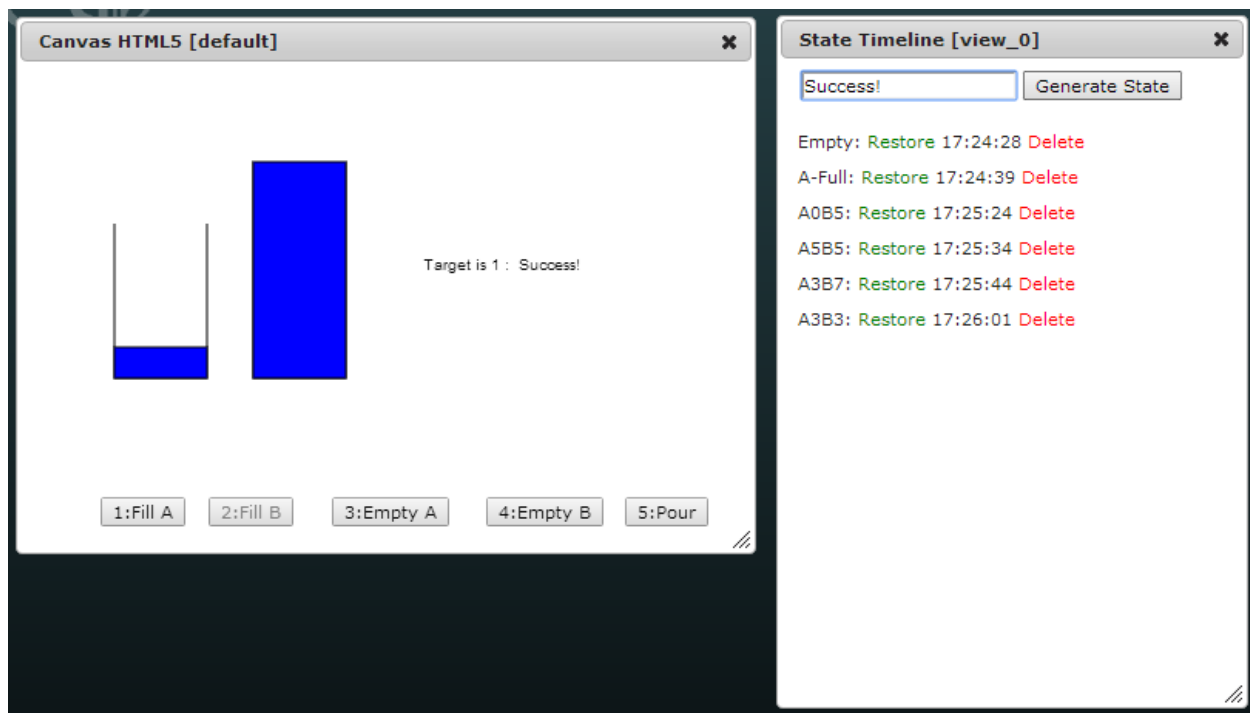
### 3.6.2 The Jugs Model - Demonstrating the State Timeline and the Dependency Map

The second model to consider is Jugs. Jugs has presence as a demonstration model in practically every EDEN implementation. The model provides the user with a series of buttons. The user may negotiate applicable buttons to fill, empty and pour water between jugs in aide of achieving the target volume in either of the jugs, specified to the right of the jugs - the default is 1.

The task of achieving the correct volume in either jug is obvious. The modeller may

wish to map out the different routes to the solution by testing exhaustively - A tool which would be great use to the modeller in this situation is the State Timeline. (The 3rd development objective)

The State Timeline allows the modeller to record snapshots of the state of the model and label them as they see fit. By labelling each state with the volume of jugA and jugB the modeller can more easily inspect and skip between previously recorded states.



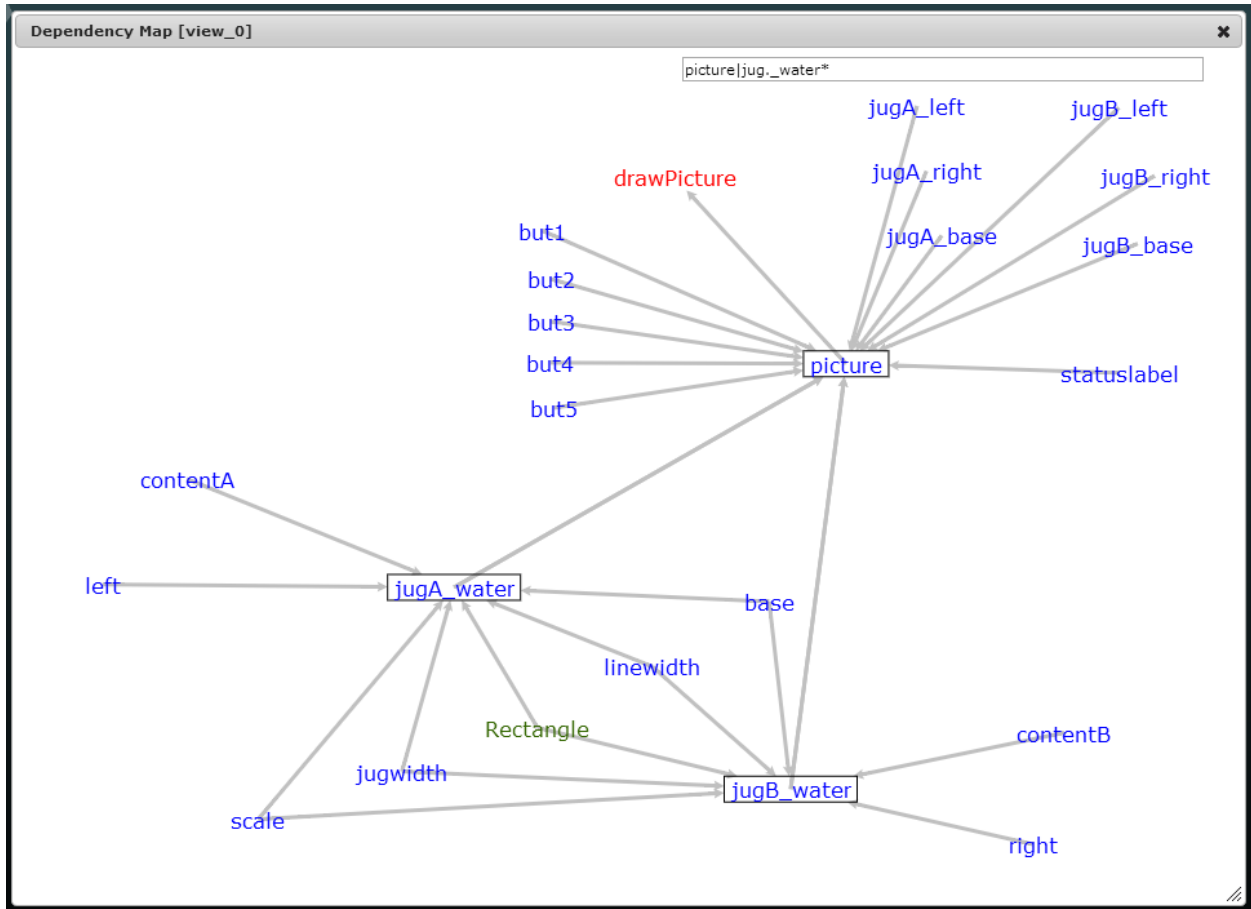
(Figure 11 - Jugs Model with State Timeline Plugin)

If the modeller wished to develop from state 06; that is to say the state in which left jug has volume 0 and the right jug has volume 6, they need only click 'restore' beside the label '06' - they can then go on to record further states after exploring various paths of exchange, or return to the state that they came from - provided they recorded it. This constitutes demonstration of the State Timeline.

The modeller may wish to inspect the structure of the model. They may desire in the

case of jugs, to understand the construction of the model surrounding the liquid. They could use the Symbol Lookup Table to individually inspect the dependencies of each observable symbol, but that would be tedious and difficult to visualise; they would have to manually search for keywords that they know are related to the liquid and follow a string of dependencies. A tool that would greatly aid their struggle is the Dependency Map.

The dependency map facilitates the visualisation of functional dependency, and therefore structure within the model. A good place to start is to instruct the map to display all of the observables associated with what is drawn to the picture. This can be achieved by simple typing “picture” into the regex box at the top of the dependency map tool. The picture observable symbol will appear as a boxed node in the graph, and all nodes functionally dependent on or to it, will appear as unboxed nodes with directed edges. From the nodes that have appeared, the modeller will be able to see two particular nodes labelled “jugA\_water” and “jugB\_water” they may then extend the regular expression to enlarge capture to those observables in order to reveal their dependent parts. The new regular expression will be “picture|jug.\_water”



(Figure 12 - Dependency Map illustrating JUGS)

The nodes can then be arranged manually to make better sense of the structure within the model such as in the picture above. The modeller is able to clearly see that both jugs' liquid representation are made structurally identically, they each rely on Rectangle, they each have a linewidth, jugwidth, base and scale; contentA and contentB are obviously very similar, as are left and right, though it is not clear what these observables represent from the diagram. This information and its conclusions, along with further potential for exploration would not be easy without this tool. This constitutes the demonstration of the Dependency map.

### 3.7 Project Management

This section outlines the results of project management activities.

Development within an environment that was completely under my own control was possible due to Vagrant [9] and VirtualBox [12]. The program installed a virtual server on a private machine that allowed a shared location on the harddrive to be broadcast on a specified port. The location was selected to be within a dropbox directory, thus allowing development to be made from a remote location, this functionality was especially useful during the CS405 labs, when the assistance of others with superior knowledge and experience with web technologies were available for assistance with some of the more confounding problems.

The initial investigations did not take as long as anticipated. After approximately 3 days it was concluded that the bulk of the environmental familiarity would come with development, thus, after locating the the functions that would allow the vital extraction of information from the database of observables, development of the first objective was started.

The first development objective took the longest to complete. With lack of environmental awareness and inexperience with JQuery, HTML, CSS etc. The simplest design activities took far longer than they would have by the end of the project. After 4-5 weeks of development and incidental investigation the first tool was in a functional but undesirable state. The aesthetics would have to wait for the overflow period between terms 1 and 2.

The second development objective was achieved in roughly 3 weeks, less than forecast. During this time significant miscellaneous development also took place. The swiftness of this was due to the discovery of the Springy Javascript library which removed the need to implement a force directed algorithm from scratch.

During the overflow period the refactoring of the first tool into two separate plugins, along with the promised advances in aesthetics were completed. In addition, the third development objective was also completed, all in a period of 4 weeks.

During the interval that the final tool was set to be completed in, further refinements were made, and possible opportunities for further extension were discussed with the project supervisor.

Multiple submissions of the tool as a work in progress were submitted. The improvements made to the UI and the interpreter were evident from the outset, and useful to the CS405 students using the environment as part of their coursework.

Meeting with the project supervisor was routine, averaging roughly 3-4 hours per week. Subjects of discussion most often featured miscellaneous development tasks, although when a submission was ready to be made, feedback and constructive criticism was often received on demonstration and addressed before submission to the department servers. A particular submission which rendered a significant proportion of the tool useless was identified and addressed within minutes of discovery.

## 4 Further Considerations & Overview

### 4.0 Overview

This section concludes the project. The outcomes are summarised, and a detailed recommendation of future directions to take are discussed.

### 4.1 Project Conclusions

The project has improved the state of the environment in a variety of ways. Many improvements have been made. Four high level plugins have been developed to assist the user in their modelling endeavours. Each provide the functionality of system state customisation and recall, which can enable the modeller to explore many paths of configuration previously tedious to manually attempt. They provide insight into the structure of models, this can be of great use to any modeller attempting to understand the structure of their, or anyone elses model. They also provide more information about the system environment to the user, without which technical reassurance in times of confusion may be difficult if not impossible to obtain without hacking.

Many existing structures have been improved, some entirely re-written to support the philosophies of Empirical Modelling - the input window in particular. These improvements will ultimately mean that future developments require less effort in these particular areas that have been re-engineered. Compatibility issues with other plugins are also easier as a result of the JQuery removal, which has been replaced with pure Javascript wherever possible.

Many minor aesthetic changes have been made to the tool for the benefit of the user.

Text highlighting to display to the user which symbols are defined for example. These changes increase the level of feedback that the user receives, which is key to any integrated development environment. The higher frequency the feedback presented to the modeller, the less work they have to do to solve problems. With high quality feedback the gap between the modeler's understanding and their implementation is ever stronger bridged.

A plethora of existing HTML drawable components have been bug fixed, their code has been greatly simplified in some cases and a number of new drawables have been added. Improvements suggested by the CS405 students were immediately addressed, this included the implementation of many new drawables such as Pixel and Textarea. Bugs have for a long time plagued the use of this tool, and that is now significantly eased as a result of the efforts from this project. Elusive errors such as HTML interpretation in code submission have been suppressed. The environment now produces relatively few terminal errors on submission of code. The removal of roadblocks such as these will undoubtedly have a great effect on those using the tool. It will help to prevent detracting people who are interested in the ideas of Empirical Modelling, but frustrated by practical inadequacies.

A variety of skills have been gained as a result of the tasks carried out in this project. Starting out with only a basic understanding of HTML, CSS, and a more developed understanding of Javascript, the knowledge required to bridge these three technologies have been acquired. Critical evaluation of the use of JQuery and its relation to the conventional approach to web based programming using the Document Object Model (DOM) with Javascript has been exercised and the use of alternative notations in language design have been explored.

An invaluable experience with software development in practice has been obtained, particularly with reference to development across generations by multiple developers.



Critical evaluation of approaches including recommendations for improvements have been central to the project. System design skills have also been honed, and knowledge of the philosophies relating to Empirical Modelling have been strengthened.

## 4.2 Toward The Future

As a consequence of investigations and development with JS-EDEN, sufficient insight into the direction the tool should take as a result of future development has been made. A critical recommendation as to how the environment should evolve is an appropriate way to conclude the project.

As highlighted throughout the report, the use of HTML constructs as a shortcut to achieve interactive elements that the user can manipulate to guide the construction and exploration of their model, should be entirely deprecated. These constructs would afford the user far more control over their display and functionality if they were to be implemented directly within the canvas. The modeller would also be able to specify such constraints much easier within EDEN rather than the minimal control HTML provides. The already deprecated 'setInterval' method to refresh the canvas would also be able to be replaced by the entirely more appropriate requestAnimationFrame as encouraged by the online community [10]. This is currently not possible as the elements lock up when the canvas refreshes too quickly, it would refresh much quicker providing the user with a smoother update feel with the recommended function due to the efficiencies afforded. The backend of the code would also be significantly easier to maintain, as developers would no longer have to concern themselves with maintaining multiple render loops. The constructs would actually all be drawables, rather than drawables and appendables. This alteration will take time to implement from scratch but the efficiency, maintainability, extensibility and integrity afforded are in the legacy and future interest of Empirical Modelling.

The removal of JQuery and other heavy imported libraries are paramount to the malleability of the application environment. As experienced throughout development, JQuery methods pervert the standard Javascript approaches to programming. In some cases overwriting pure implemented code when one least expects it. The two approaches cannot be used together with conceptual stability, thus one must go. Since JQuery is essentially an extension of Javascript, written in Javascript: it is weaker than Javascript. Eventually, a developer will have to revert to Javascript in order to achieve something that they cannot in JQuery, since JQuery is designed for popular operations to do with webpage layout configuration or information retrieval, a highly complex experimental modelling environment such as JS-EDEN will inevitably command approaches that JQuery will be unable to satisfy. Indeed this is why a mix of Javascript-JQuery is observed throughout the backend. All of the JQuery that this application relies on in the future should be removed and replaced with pure Javascript.

The imported libraries such as the ever outdated JQuery library that must be downloaded in order to make use of the functional shortcuts afforded by the language, is always heavily obfuscated. Obfuscation deliberately skews the code so that developers cannot easily alter it. The removal of non-ideal features within user interfaces such as some unhelpful configurations within CSS, become infeasible as a result of this practice. Unless it is absolutely necessary, obfuscated libraries should not be present anywhere within the application. The existing JQuery UI with CSS needs to be removed, and alternative CSS specification should be made.

Standards of practice for development need to greatly improve in both the front and back end. Code should be well commented wherever it is not completely obvious what the implementer has done. A preamble for each section should explain to the reader what the code for the section does, and how it fits within the overall application. The user, upon loading the environment should have the option for more information about

how to use the application, including basic tutorials and examples.

There should be an ability to consolidate symbols into groups. Currently if two models were to be loaded into the application, the symbols contained in the models would be forever inseparable without individual manual respecification. Additionally there is no way of replicating or extending constructs without first considering them as a whole. This is a huge limitation to an environment that is founded on constructionism. We are forever forced to consider a brick as sand, clay and water. Effort has been made to address this issue by adopting various object oriented approaches such as the dot notation in a recent WEB-EM submission: "ODIN: A conceptual framework for an Object Extended EDEN." Without addressing grouping of symbols in one way or another there is no hope of merging models using distributed modelling techniques without serious conflicts.

The EDEN language from the perspective of the application environment JS-EDEN should be simplified. First conceived in 1987, the language contains non standard programming notation commonly referred to as "a dogs dinner" by Prof. Stephen Jarvis [11]. Symbolic notation that would not be familiar to a programmer with experience of other well known programming languages and does not fulfill some language specific function, or, notation that is not a widely accepted shortcut of a traditionally known function such as '++' should be deprecated. Traditional EDEN makes use several such functions e.g. '/' for list concatenation or '#name' for list magnitude/cardinality. To an unknowing onlooker, the functionality of such notation will be completely unknown, whereas 'name.length' or 'size(name)' would be available to programmers of a variety of backgrounds without the need for extraneous reference. It would also reinforce the ability of learners who go on to use other languages. For legacy reasons it may be desirable to maintain these features. But in order to improve the tool for the benefit of the masses who do not immediately share the passion for Empirical Modelling, it needs to be made as accessible as possible.

As with the input window, the JS-EDEN environment should be clearly accessible from within the interface. EDEN like notations, however many may exist within the tool in the future should be able to manipulate both the surrounding environment and the internal symbols using a clean, consistent and well documented API. Functions that retrieve a Javascript copy of an internal symbol, while simultaneously creating it if it doesn't exist such as the 'root.lookup' currently implemented, does not constitute a clean, consistent and well documented API. This will help to facilitate future notation development within the environment, or legacy invented notations such as SCOUT, DONALD, EDDI etc.

Due to the current condition of the backend, reengineering separate sections of the application individually is infeasible. A developer would not be able to tackle the user interface without removing the JQuery and CSS. This would cause other subsections of the application to fail. In order for the application to realise its full potential, it needs to be redesigned from scratch, preferably with each section individually and incrementally added layer by layer.

### **4.3 Plans in motion**

Application for funding during the summer of 2014 has been made to IATL (The Institute of Advanced Teaching and Learning) in order to re-engineer JS-EDEN with the alterations outlined above. Once completed, the environment will be tested with Javascript proficient teachers via Dr Colin Price, (Head of Computing at the University of Worcester) and Computer Science Master teachers working under the CaS (Computing at School) scheme.

The first steps toward major re-engineering as an extension to this project have been made. Completely reconstruction and design of the user interface dialogue has been

completed in its majority.



(Figure 13 - The new JS-EMPEROR interface)

Multiple issues with the interface as raised in this report are corrected in this construction.

Firstly, the 'minimise, maximise, close' trio of buttons in the top right hand corner of a standard dialogue box has been adopted to mirror a familiar interface in many popular operating systems. Unlike the previous implementation, each respective button does exactly what one would expect. A system tray has been included at the bottom of the page to reinforce exactly which plugins are open, minimised and closed. Minimised windows can be recalled by a simple click on the associated tab in the tray, again like many familiar operating systems. It has been designed so that any one window can be maximised at any one time, appearing behind all other windows. Excluding maximised windows: a single click on any part of a dialogue box will bring it to the front of all partially obstructing windows. Dialogue boxes can be dragged using the top bar and their size can be changed by dragging from any corner or side excluding the top right. Users will not experience any surprises or unhelpful/obstructive functionality from this implementation of dialogue windows. A primary crux of the previous design.

Secondly, all CSS and other code relating to the styling and functionality of the interface has been specified completely using standard Javascript. It is well structured and commented. Any intermediate level programmer would be able to alter the interface without issue.

A plan for how to proceed with migrating from the old JS-EDEN to the new interface has been set out, once basic parsing has been redesigned to support more ambitious live feedback plugins, the Observable, Dependency, Agency sub-environment will be implemented and legacy functionality will be reinvented in the new environment. A platform for distance collaboration will be established and the efforts to revitalise the tool will be trialled.

#### **4.4 Overall**

Overall I consider the project to be a complete success, with a clear direction for the future and means to achieve it, it is entirely appropriate to consider the instruments for online Empirical Modelling explored.

## 5. References

- [1] Meurig Beynon: Project Supervisor, Reader Emeritus, Department of Computer Science, University of Warwick,  
[http://www2.warwick.ac.uk/fac/sci/dcs/people/Meurig\\_Beynon](http://www2.warwick.ac.uk/fac/sci/dcs/people/Meurig_Beynon)
- [2] Edward Yung `*EDEN: An Engine for Definitive Notations*. MSc thesis, Department of Computer Science, University of Warwick, UK (September 1989),  
<http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/mscbyresearch/eyung/files/>
- [3] Timothy Monks `*A Definitive System for the Browser*. MSc dissertation, Department of Computer Science, University of Warwick, UK (September 2011),  
[http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/mscprojects/timmonks/trmonks\\_dissertation\\_report.pdf](http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/mscprojects/timmonks/trmonks_dissertation_report.pdf)
- [4] Nicolas Pope: *Supporting the Migration from Construal to Program: Rethinking Software Development*. PhD thesis, Department of Computer Science, University of Warwick, UK (December 2011),  
<http://www2.warwick.ac.uk/fac/sci/dcs/research/em/publications/phd/nick/files>
- [5] *Its not a MOOC its a Movement*, Inside HigherED, December 2013,  
<http://www.insidehighered.com/blogs/higher-ed-beta/its-not-mooc-its-movement>
- [6] *Dependency Modelling Tool*, Antony Harfield, Phd Thesis, 2006,  
<http://empublic.dcs.warwick.ac.uk/projects/dmtHarfield2006/>
- [7] *Bubble Sort*, Meurig Beynon, 1998,  
<http://empublic.dcs.warwick.ac.uk/projects/bubblesortBeynon1998/>
- [8] Hui Zhu, MSc Research Student, Department of Computer Science, University of Warwick.
- [9] Vagrant, Open Source Development Environment, MishiCorp,  
<http://www.vagrantup.com/>
- [10] *setInterval for game draw loop not running consistently*, Stack Overflow,  
<http://stackoverflow.com/questions/11268885/javascript-setinterval-for-game-draw-lo>



[op-not-running-consistently](#)

[11] Stephen Jarvis, Professor, Department of Computer Science, University of Warwick.

[http://www2.warwick.ac.uk/fac/sci/dcs/people/stephen\\_jarvis/](http://www2.warwick.ac.uk/fac/sci/dcs/people/stephen_jarvis/)

[12] VirtualBox, General Purpose Full Virtualiser, Oracle,

<https://www.virtualbox.org/>

[13] Jane Sinclair: Project Supervisor, Associate Professor, Department of Computer Science, University of Warwick

[http://www2.warwick.ac.uk/fac/sci/dcs/people/Jane\\_Sinclair](http://www2.warwick.ac.uk/fac/sci/dcs/people/Jane_Sinclair)

## 6. Legal Considerations

The copyright of the original developers of the JS-EDEN source code and 3rd party software must be respected and upheld.

Vagrant is released under the MIT License.

VirtualBox is released under GPL General Product License V2.

Springy.js is released under the MIT License.