ELS – An Eden Based Digital Logic Simulator

0315420

Abstract

This paper will explore the possibilities and problems associated with modelling digital logic circuits in a definitive language. This will be achieved by attempting to model a digital circuit (the logic, its visual representation and an editor) using a definitive notation. It is hoped that wires in a circuit can be modelled as dependencies in Eden - the output from the wire (and therefore input to the next component) *is* the output of the previous component. The methods used by the model (and possibly the efficiency of a dependency based simulation) will then be compared and contrasted with the methods used in the Java based OLS solution.

Weighting : 40% Paper, 60% Model

1 Introduction

This paper was inspired by the authors work on Open Logic Sim 2 (OLSv2) - an open source digital logic simulator written in Java and previously used in the "Computer Organisation and Architecture" module of the 1st year computer science degree at The University Of Warwick. The author made many improvements to OLSv1.36 in his second and third years so has an in depth understanding of OLS's implementation. The authors initial aims were to create a tool that could be used for modelling digital circuits in Eden based on 'is' dependencies in Eden. Then, the model could be extended for use as a teaching aid (see '5 Future Directions'). There was also the possibility of developing a new notation (using the AOP) for creating and interacting with the circuits. As the model was written, it became clear this was not all possible (partly due to time constraints, partly due to the constraints of TkEden itself).

2 The Model

With the model, the user can construct a simulation of a digital circuit. There are two main sections of the model – the circuit editor and the logic to actually run the circuit. The editor allows the user to add and remove components and wires, save and load ELS files and gives access to rudimentary undo/redo functionality.

Before the model was implemented, it became clear the idea of modelling wires as dependencies would not work as any sort of feedback network would never terminate. Therefore, the system was implemented using a clock. The Eden clock is set to tick every 10ms. In practise, each tick usually takes more than 10ms to run and therefore the clock just ticks continuously whilst allowing other procedures to be run between ticks e.g. refreshing the screen.

There are several pieces of information that must be kept about each component. In a classical programming language (e.g. Java) this would have been implemented as a data structure/class. However, since Eden does not support user defined data structures, all the information about a component had to be held in lists of a specified structure. This made programming difficult at times as instead of using the dot notation (position.x) one must access all the information just using its index (position[1]). For the component data structure this became quite confusing at times, as some parts of the structure are 5 lists deep.

The visual representation of the circuit (and the ability to edit that representation) was based on the DMT graph although the model has been heavily modified to reach this stage. The DMT used macro() as a way to emulate classes. Macro allows code represented as a string to be modified by replacing ?1 (etc.) with a given argument. This means a particular piece of code can be applied to multiple 'objects' without writing it out by hand. The author learnt about macro (and execute) from the DMT graph model and these functions became essential to the construction of this model. It allows, for instance, the code for drawing an and component to be used

as many times as necessary (once for each and component in the component array).

The visual representation of the model attempts to represent the circuit in a human 'readable' way – allowing the user to recognise what a component is and showing how the components are interconnected and (for some) what the components current state is i.e. switches and LEDs. Although not really considered as components, 'wires' also show their current state by changing colour depending on whether the output they are connected to is 'on' or 'off'.

3 Issues

A discussion follows of several issues encountered when constructing the model.

3.1 The Concept

One of the key inspirations for this project was the possibility of modelling wires in the circuit as dependencies in the model. Unfortunately, this was quickly discounted due to the problems that would arise from self-referential circuits e.g. the not-SR latch. This uses 2 nand gates to store 1 bit of information by feeding the outputs of each nand gate back to one of the inputs of the other nand gate (see figures 1 and 2 in 4.2 Comparison As A User). When trying to enter this circuit into TkEden, if the model was made using dependencies, an error would have been generated. Therefore, the model was implemented using a clock so the model would perform 1 step each clock tick. However, this would also have lead to a problem as the order in which components were ticked would have mattered – components may have used some data from the previous step and some from this step. Therefore, a notion of 'previous state' had to be used so all outputs could be calculated from the components input state when the step was started.

3.2 The Implementation

There were two areas where problems arose in the implementation of this project – problems with me and problems with the model. I had 2 problems – firstly, my knowledge of TkEden and its languages is very basic. Having only had limited exposure to the use of Eden before this project, writing any model in TkEden would have been a struggle since it takes time working out how to perform relatively basic tasks. Secondly, having to use execute and macro to emulate classes was difficult to understand (and debug) sometimes since any functions used with macro had to be entered as a string.

Now, the more important problems are those concerning the model itself. They will be presented here in order of significance (least significant first).

1. The model will not run on TkEden 1.68. This seems to be a bug in TkEden (as opposed to in the model) as the model works on 1.66.

2. There seems to be a memory leak although I cannot tell if this is something the model is doing wrong or a fault of TkEden. I tried to debug this using several commands found in the Eden documentation (e.g. symboltable and symbols) but nothing was of any use. I do not know how to debug this problem and find where the fault lies.

3. The most important problem - the model slows down dramatically as more components are added. The laptop used for writing and testing the model is an AthlonM 3000+, approximately 2 years old. When tested, the model slowed down noticeably after adding approximately 25 switches (simple to draw components) or 15 nands (more complex components). This was with no interconnections having been made, just adding lots of components. Again, I am unsure as to whether this is due to a poorly written model, bugs in TkEden or simply that Eden is not an appropriate language to write this type of model in. The latter possibility occurs because Eden is an interpreted language and will therefore be comparatively slow anyway (which will only be noticeable when workload is high).

3.3 Eden and TkEden

The final set of issues is issues with TkEden itself. Firstly, the documentation of Eden and the other languages is difficult to use in the text form and difficult to find in the web form. The amount of documentation is comparatively tiny – if a programmer needs to find out how to do something in, for example, Java, they will usually be able to find an answer through any search engine. However, in Eden one either must read through all the documentation hoping to find something relevant or try to guess what the functions may be called and search for it. Now, maybe this is an unfair comparison (since Java's user-base is rather larger than that of Eden) but it does not negate the fact that the vital resource of community help is almost non-existent.

The more important issues are those concerning the Eden language itself (and, indeed, perhaps the very basis of Empirical Modelling). Coming form a classical computer science background, there are some

features which I would have found extremely useful when making this model. Firstly, as has been mentioned previously, data types would have been much easier to work with. Data types help the programmer to more easily understand what they're writing by actually using names for the data types elements, not just list indices. In the same way, they help ensure semantic correctness of programs – getting 2 list indices mixed up (e.g. data[1][2]) would not throw a 'compile time' error (or 'interpret-time' error in Eden's case) and would not necessarily cause a 'run time' error. Whereas, getting 2 elements of different data types would probably throw an error (e.g. data.input.connection).

Another feature that would be very useful is classes. This would simplify several of the processes involved in the circuit model – most importantly, the process of defining types of components. Since all the components share the same types of data and use the same types of functions, it would be appropriate to use classes for representing the components (as is done in OLS).

4 Comparison With OLS

One of the aims of the project was to compare OLS to ELS. The comparison follows.

4.1 Comparison As a Programmer

As a programmer, OLS is much easier to program. This is due to two factors – the author's familiarity (lack of) with the Java (Eden) programming language and the suitability of modelling a digital circuit simulator in Eden. In a digital circuit there are many components – almost all having the same information and needing the same functions. This means the components are well placed for using classes – firstly so all the components can be interfaced with in the same way and secondly so many instances of a component can be easily created. Whilst this can be done in Eden (using the macro function as discussed previously) it is difficult to write and to comprehend.

4.2 Comparison As a User

I will try to evaluate the experience of a user comparing the 2 programs. However, I have a unique point of view having done a lot of work with both of them.

The first difference a user would see is that OLS has a much more 'polished' interface. The example figures below demonstrate how different the interface are but, at the same time, how they can both be used to display a similar visual representation of a circuit.

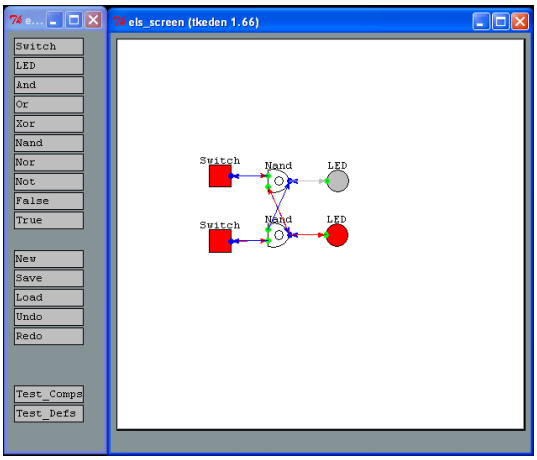

Figure 1: A not-SR latch implemented in ELS.

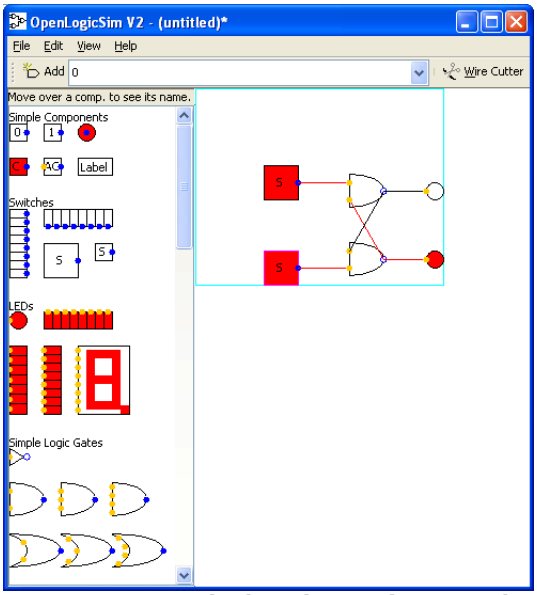

Figure 2: A not-SR latch implemented in OLS. The light blue box around the components is the edge of the (expandable) circuit.

The ELS interface could be improved (probably to a similar standard of OLS) given time and effort.

More importantly (depending on the users point of view), a comparison must be made between speeds of execution. Eden is slow compared to Java. There is no way to avoid this fact – probably because Eden is an interpreted language and Eden has to maintain dependencies. These issues are exacerbated by the fact that many 'built in' functions are themselves written in Eden (see 'lib-tkeden*.eden' in the TkEden release, especially eden.eden) when it would almost certainly be faster to run these in

whatever language TkEden is written in (the language is irrelevant – it would remove a layer of interpretation whatever the language was).

Another point is memory usage. Since TkEden is not typed, I would assume it dynamically allocates memory depending on what type it believes observables to be. Strong typing should help to reduce memory usage since only the necessary amount of memory would need to be allocated. This is also applicable to lists – if a data structure (like the component data structure) is implemented as a list, the interpreter has no concept of what should be in the data structure, how big it should be etc. leading to degradation of memory performance (and probably increasing the time of execution).

However, TkEden (and Empirical Modelling in general) does have 1 major advantage – its adaptability. The ease with which one can redefine anything at will is amazingly useful – in the appropriate context. If a function is not doing what it is supposed, it can be changed anywhere, anytime. This has been very useful for debugging purposes and whilst writing the model but would not be so useful once the model was finished. However, is a model ever finished? This is one of the fundamental points of Empirical Modelling – each individual user must decide if the model is finished for their purposes. There may be bugs in it but if a user never encounters that bug, is the program finished? Since I am (and will remain) firmly rooted in classical computer programming, I would say no! However, in Empirical Modelling, since the decision is left open to the user and the user does not encounter the bug then (I believe) to them that aspect of the model would be finished.

5 Future Directions

There are many directions this model could be taken in the future depending on what the modeller was developing the model for. If it were for the same purpose as OLS (part of teaching a course) then it would probably help if the user interface were improved – if one is trying to learn about digital logic circuits, one should spend as little time as possible learning about the program used for teaching.

For lower levels of education, the model could be developed (as had been hoped for at the start) to have a way for 2 (or more) students to interact with the same model using DTkEden. Firstly, this would facilitate groups to work together on building one circuit but not have to be fighting over one keyboard and mouse (they could just fight virtually through DTkEden). It could then be further extended to allow two or more students to create circuits, swap them with each other so that another student can 'break' some of the components of the circuit, and then give it back to the creator so they can find out what is broken in the circuit. This would require quite a lot of work on the model – for instance being able to set different modes for how a component breaks would be quite a challenge.

A less important extension is simply to add more components – ELS currently has 10 components, OLS has 47. However, problems would arise adding components with lots of inputs/outputs due to the way the script has been written. Currently, all inputs are on the left edge of the components and outputs on the right edge. They are all evenly spaced along their edge. Also, currently all the components have to be the same width and height. Whilst these features could be modified, the lack of classes makes this more difficult than it should be.

6 Conclusion

This model was a very interesting challenge to program. However, the OLS implementation of a digital logic circuit simulator is much better (in my view of the purpose of OLS/ELS) than the ELS solution. Since my approach to ELS was very much based on what I learnt from programming OLS, there may have been a better way to implement ELS that I didn't see because I was blinkered by classical programming methods. I suspect, however, it is merely that Eden is not suited to this type of model/program.

From my brief foray into the world of Empirical Modelling it seems this way of thinking and programming has few useful applications. Though I have not seen that many models, I believe I have yet to find a situation where writing an Eden model is more 'beneficial' (however that is measured) than writing a classical program.

Acknowledgements

The model was adapted from the Eden version of the DMT graph, given to us in lab 5 (I could not find a projects directory for the model).

References

None