Industrial Project – Digital Design Automation Software

Jonathan Beaumont – 2013/14

A report submitted for the Industrial Project of MEng Electronics and Computer Engineering

<u>**Summary**</u>

Digital systems can be very complex to design as they contain many different features comprising of many logic gates, some of which can be part of multiples of features. Rather than design a whole system at once, it is easier to design each feature, function or subsystem separately, and then combine them once each separate design has reached a point of completion.

For this project, I used the algebra of Parameterised Graphs as a method of mathematically defining each subsystem and created software which used equations representing these, expanded them, and used them for further equations, such as the combining of many of these graphs for example, to find the equation of a whole system.

This report will contain my experiences when writing software for the purpose of automating digital design using my knowledge of programming and the design of digital systems. This success of this project, and how it could be used in the future will also be discussed.

Contents

Introduction

1.1 Project Description

This software will be written in the Java programming language to run on a PC using a Java Virtual Machine. A function of a digital system will be described, using Parameterised Graphs, by a set of vertices and their edges, and this can then be represented algebraically in an equation. Each function will be described using the same method, and these equations can be referred to in another equation using a label, for example:

$$Graph1 = A + B \qquad Graph2 = C + Graph1 = C + (A + B)$$

Figure 1 – Graphs can be referenced within other graphs

After all functions have been described in this way, a list of all the functions equations will be obtained.

The labels for the function equations can then be used to combine together to generate further equations containing one or more of the functions, and this acts as a method of combining functions together in an attempt to build an equation of the overall system.

Boolean variables can be applied to vertices or equations, and their values set, to determine whether or not certain parts of functions or even entire functions will appear in an equation. For example, in an equation made up of multiple equations representing features of the system, if one of these features is only to appear if a Boolean variable is set to true, then its feature equation, or its label when referenced, can have a Boolean variable applied to it, which will cause this functions equation to appear only if this variable is set to true.

When a the equations have all been described, a text file is created listing equations representing the functions, as well as equations containing labels for function equations which are to be expanded and calculated to find an equation for multiple functions. This file also lists the Boolean variable values, either ‘1’ for “True” or ‘0’ for “False”, however a Boolean variable does not have to be set. An unset variable will use its label to represent it in the equations.

Once this file is created, the software can take all the functions from the file in turn, expand them by removing labels wherever possible, and calculate any new edges in the graphs. It will then apply Boolean variables and their values to the equations, and return the newly calculated equation to a specified output text file.

This output file will list all equations listed in the input file, but with each containing the vertices and edges of all equations referenced by label, and with Booleans resolved to remove any necessary equations. Each equation has duplicates removed, and necessary steps are taken so each equation applies the properties of Parameterised Graphs.

1.2 Reasons for Undertaking the Project

Digital systems are becoming more and more powerful as time passes, as well as smaller. For example; computers, smart phones and tablets are some of the more commonly used devices which are in high demand, and required to do more than word processing, play music or make a phone call.

Designing these systems has become more complex as a result, and it is difficult to ensure that the system functions correctly. If it is possible to model each function of the system as a set of possible states the digital system could enter, and the ways they can enter other states, then it is possible to test they work and check for similarities between the functions[1].

Parameterised Graphs is a method of doing this. Each group of states, or function, can be visually modelled as a graph, similar to a state transition graph. However, these graphs can then be represented as algebraic equations. In one system, there can be many equations representing functions, and these can be combined in ways to check for similarities.

The number of possible states can be huge in modern systems, so the process of checking all equations can be lengthy and complex if carried out manually, and this can make the outcome subject to errors. A better way of carrying out the necessary calculations for this method is automatically, using software.

An application specifically designed to take large numbers of these equations, expand them and combine them as specified can reduce the likelihood of errors and the time in which it takes to complete the process. Further work can then be done with the results to change the design of the system. This can help reduce the design time of the system.

1.3 Aims and Objectives

Now that the reasoning behind this project is known, it is possible to define what can be done to improve the system, and some deliverables for the end of the project. The aims and objectives for this project are:

- Learn more about Parameterised Graphs, and the design of digital and embedded systems.

  As an Electronic and Computer Engineering student, I am interested in digital and embedded systems, and this project will help to me to understand the process of digital design. This will be useful for my future career.

- Create an application that can take several graph equations, and expand them, perform calculations with them and output results, all while keeping with the rules defined by Parameterised Graphs.

  This is the major function of the software, and will be the major component when determining the projects success.
- Add to the application the ability to use Boolean variables as part of the calculation process.

  Part of parameterised graphs is the ability to use Boolean variables as conditions during calculations, to change which equations or vertices are used. This is another important part of the software.
- Add additional functionality for the purposes of making the usage of the software simpler.
  - Features such as validation for the equations to ensure that the results are not incorrect due to some form of input error, adding this software into the existing application, Workcraft, which is used within the university.

As of writing this, the software correctly takes in a text file with a list of graph equations and Boolean values, calculates graphs and includes Booleans when calculating and outputs them to a text file. Some simple validation is included for Booleans, but no major validation for the whole equations.

1.4 Layout of the Report

Project Background

This will feature a review into Parameterised Graphs, and to the necessary features of it which need to be included in my software. I will also explain more about the software already used by researchers, Workcraft. This software has features for creating graphs and viewing them, and I will discuss how this project could work with this.

Project Specification

I will give a detailed description of what my application will contain and how I intend to write it. The specification is important to know what the outcomes of the project should be, and it should detail what aspects of the program take priority, useful to know when looking at the final result of the project.

Design Flow

In this section I will aim to discuss the process of creating the software, including how I wrote the program, the problems I faced, and the aspects of the specification I completed and didn’t, and reasoning as to why these we’re left uncompleted at the end of the project.

Results and Discussion

The software has to produce results which are compliant with some properties of Parameterised Graphs, and in this section I will try to prove that the software works with these, and discuss why the properties are important, as well as how the software corrects for these. There will also be some tests to prove that the software actually works with full graphs, including several properties and Boolean variables.

Project Review

The project was undertaken to create software for a purpose, I will discuss how the project will be used in this section, and how it could be advanced for use in the future.

Critical Analysis of the Project

The project, while completed, may not be perfect. As the author of the software, I will discuss how I feel the project is in its current state, and how I would, if I feel it is required, change it.

Conclusion

In the final part of the report, I will discuss the value of the project for me, for the end user, and the overall success of the project, with reference to the aims and objectives, to prove how well the version of the software available at the end of the project reflects my original thoughts on the project at the start.

Project Background

It is important to understand, to some degree, the state of the work around the project you are doing before the project is started, in order to ensure that you understand it and can make improvements with the work you can do. In this section I will explain my understanding of what is being used in this method of digital design. This includes two aspects; Parameterised Graphs, the graphs and algebra used in the design, and WorkCraft, existing software which is used with Parameterised Graphs. I will explain how each of the works, in order to help understand how the application created for this project will work.

2.1 Parameterised Graphs

In simple terms, Parameterised Graphs are a type of directed graph, containing vertices and edges, and similar to a state graph, each vertex acts as a possible state, and the edges act as single direction transitions between states. The graphs provide a visual method of viewing how a function could work in a system, however Parameterised Graphs can also be represented in algebraic form.

The algebra of these graphs describes the vertices and their edges, but not every vertex is necessarily connected to another. In order to describe both which vertices are in a graph, and which edges connect which vertices, there needs to be operators which describe these. In the algebra of parameterised graphs, there are two operators: Overlay, represented by '+' or ' ' (space), and Sequence, represented by "->".

An overlay can be used to express the inclusion of a vertex in a graph, for example:

$$Graph1 = A + B + C + D + E$$

Figure 2 – Algebra of a Parameterised Graph featuring overlays only.

This graph has 5 vertices; A, B, C, D and E. Each one is in the graph named Graph1, but there are no edges. The visual Parameterised Graph of this is as follows:

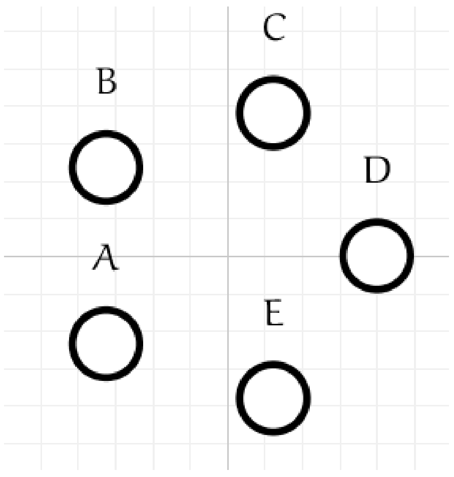


Figure 3 – The Parameterised Graph of Graph1 from figure 2

Figure 3 shows all vertices, but no edges.

Sequences are used to show edges connecting vertices; they are used to connect one vertex to another, in a single direction. Vertices can be connected to any other vertex, including itself. For example:

$$Graph2 = P \rightarrow Q$$

Figure 4 – Algebra of a Parameterised Graph featuring sequences only.

In this example, Graph2 describes an edge from vertex P to vertex Q. The visual Parameterised Graph of Graph2 is:

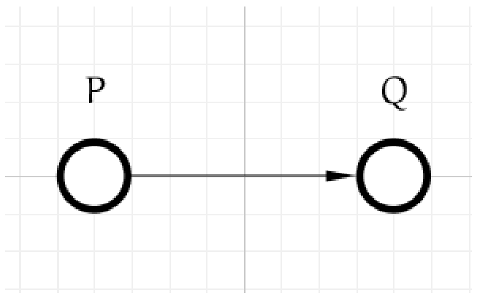


Figure 5 - The Parameterised Graph of Graph2 from figure 4.

A sequence does not have to be used to only express the edge between two vertices. Multiple vertices can be connected in sequence, and this will describe that every vertex will connect to every vertex following it in a sequence. For example:

$$Graph3 = M \rightarrow N \rightarrow X \rightarrow M$$

Figure 6 – Algebra of a Paramerterised Graph of a long sequence.

Figure 6 contains a longer sequence which, as in Figure 4, contains only sequences. Like in figure 4, M has an edge connecting it to N, N connects to X, and X connects to M. However M also connects to X and M (itself), and N also connect to M. The graphical representation of Graph3 is:

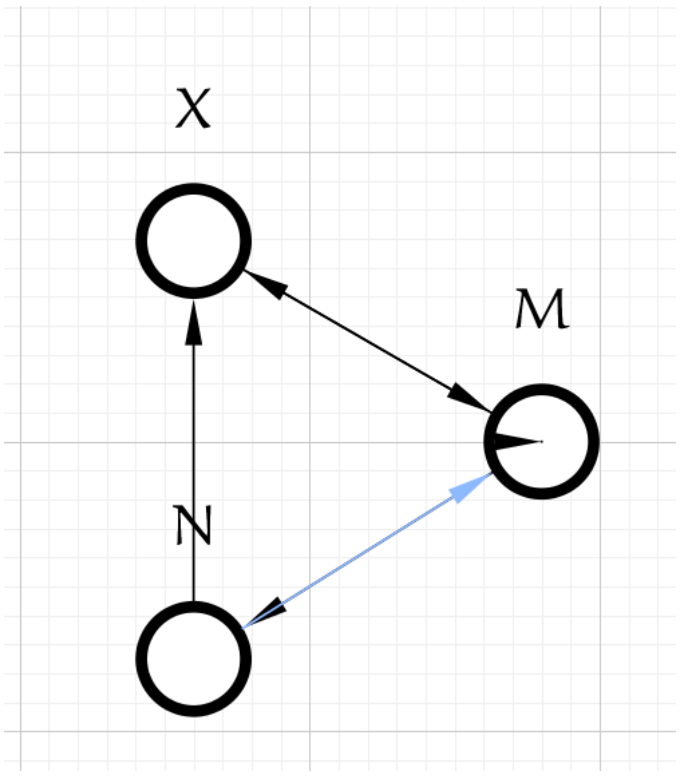


Figure 7 – The Parameterised Graph of Graph3 from figure 6.

This graph, as well as the graphs in figures 3 and 5, were created using WorkCraft, software already used in the design of Parameterised Graphs. WorkCraft has functionality to create vertices and then connect them. In this example, the graph has been created to represent the equation Graph3, however, the edges appear to be two-way. In figure 7, I have selected the edge from N to M to show that each edge is separate, and two-way edges have not been created. The black segment in the circle representing the vertex M shows that it has an edge connecting it to itself. Figure 7 shows all the necessary connections that the Graph3 describes.

Of course, both these operators can be combined in one equation. For example:

$$Graph4 = A \rightarrow B \rightarrow C + P \rightarrow Q$$

Figure 8 – Algebra of a Graph containing both overlays and sequences.

Both A->B->C and P->Q are parts of this graph, but no vertices in the first part, A, B or C, connect to the vertices in the second part, P or Q. Thus, the overlay ('+') will ensure these are both in Graph4, but no connections between them will occur. The visual representation is:

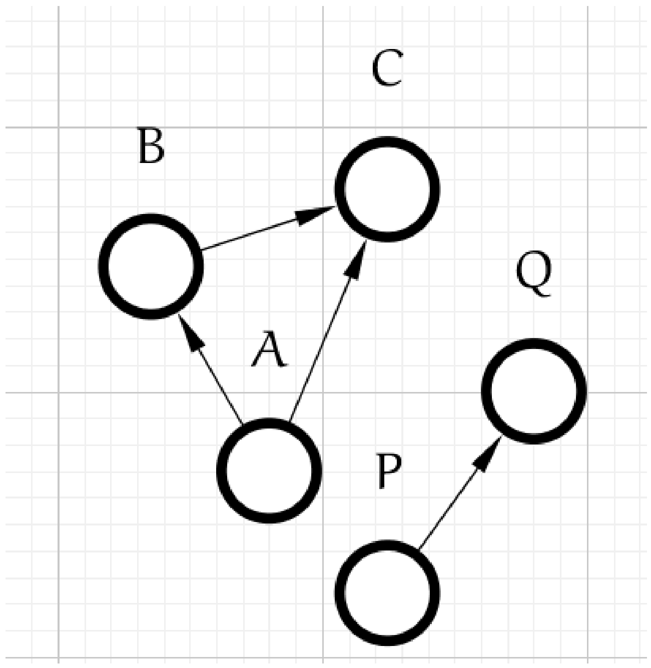


Figure 9 – The Parameterised Graph of Graph 4 from figure 8

Figure 9 shows two parts to the graph, where A connects to B and C, and B connects to C. Also, with no connections to the vertices of A,B or C, P connects to Q. This is expected from the equation shown in figure 8, Graph4.

Of course, with algebraic operations there are some properties and equivalences which arise in order to set some rules by to ensure operations are completed correctly in every situation, and to help in simplification. The properties were useful when writing the application, as it helped me to have some structure by which to calculate the combinations of several graphs, however simplification was not overly included in the final application, so the equivalences were mostly unused. I will not discuss the properties in this section, but they can be found in the appendix in figure 33, taken from the Documentation on Parameterised Graphs. I will use them in testing the software to prove it correctly operates by these properties.

Graphs themselves can also be combined using sequences and overlays. They have similar properties to the equations, however instead of referring to the actual graph equation of each graph, you can use its label. For example:

$$G1 = A \rightarrow (X + Y)$$

$$G2 = B \rightarrow (P + Q)$$

$$G3 = G1 + G2$$

Figure 10 – A set of Parameterised Graph equations, featuring some containing graphs themselves.

When calculating this, G1 and G2 would be replaced by their equations. This is useful, as one single part of a graph could appear in several different graphs, so if this part is reduced to a label, it can reduce the length of the equation during calculation. This feature is used in the application. A break down of the equation G3 would be as follows:

$$G3 = \big(A \rightarrow (X + Y)\big) + \big(B \rightarrow (P + Q)\big)$$

$$G3 = (A \rightarrow X + A \rightarrow Y) + (B \rightarrow P + B \rightarrow Q)$$

$$G3 = A \rightarrow X + A \rightarrow Y + B \rightarrow P + B \rightarrow Q$$

Figure 11 – The break down of calculating G3 from G1 and G2.

The properties for using Graphs with overlay and sequence are shown in the appendix, in figure 34. It can be seen that these are some what similar to the properties in figure 33. However, it is necessary to apply these properties to the graphs when calculating using them, before replacing the graph label with the equation it represents.

The final feature of Parameterised Graphs that needed to be included in the software for this project, and the final feature I learned about, was conditions. Boolean variables equations can be included, between '[' and ']'. These are used to determine whether or not a vertex, edge or even entire graph will appear in the final equation. In essence this can represent a flag in a digital system which, depending on whether or not it is set, will govern which states the system could move into.

In this application of Boolean equations, there are two operators which are used; '^' for an AND operation, and 'v' for an OR operation. There is also a '!' character which is used to show negation. These expressions need to be validated to check for tautologies and such, and resolved to find their final value, to be true or false. Only a true value will cause the vertex, edge, or graph to appear in the final calculation. A false will not.

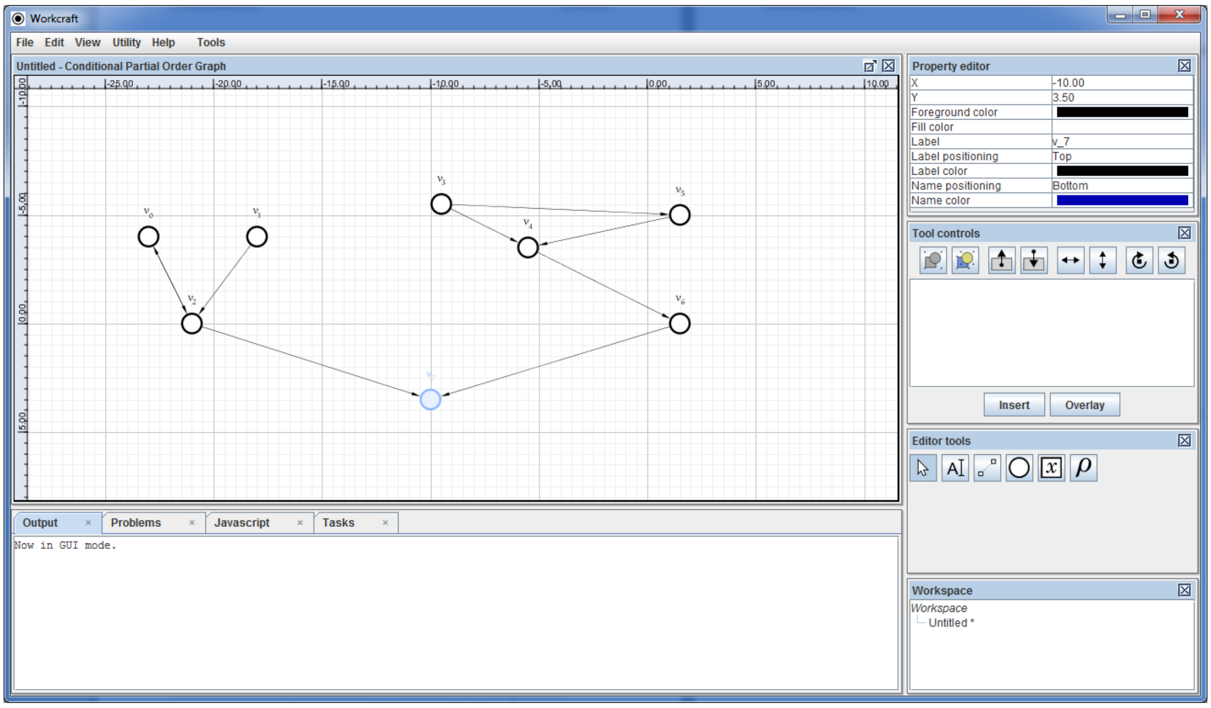
These operations, '^', 'v' and '!', have their own properties which are used in the validation process. I used these when validating and resolving Boolean expressions from the equations, in order to find a value. The properties I used are shown in 35, in the appendix.

Some properties when used with Paramertised Graphs are shown in33 and 34. These must be included for calculations to be carried out correctly when Booleans are present in graph equations. Again, I will not go into too much detail when discussing them here, but I will use the Boolean properties when testing the program, and discuss the importance in the discussion section.

Parameterised Graphs is an interesting and effective method for designing and testing digital systems, and this discussion of them has only scratched the surface of their use and potential. In the future I aim to learn about them more thoroughly, and use them when designing systems, but for this project, the amount I have learned is more than enough to create software which can successfully perform calculations with these graphs, and hopefully make the future of using Parameterised Graphs easier. [2, 3]

2.2 Workcraft

Workcraft is open source software, available at http://workcraft.org, which is already used in the aid of digital system design. Simply put, this software provides tools to visually design a digital system using various different graph types. This includes Conditional Partial Order Graphs, which is what Parameterised Graphs are based on, and this is the graph type used when designing parameterised graphs with Workcraft. This is the type I will focus on while discussing the software.



Property editor	
X	-10.00
Y	3.50
Foreground color	■
Fill color	
Label	v_7
Label positioning	Top
Label color	■
Name positioning	Bottom
Name color	🟦

```
Now in GUI mode.
```

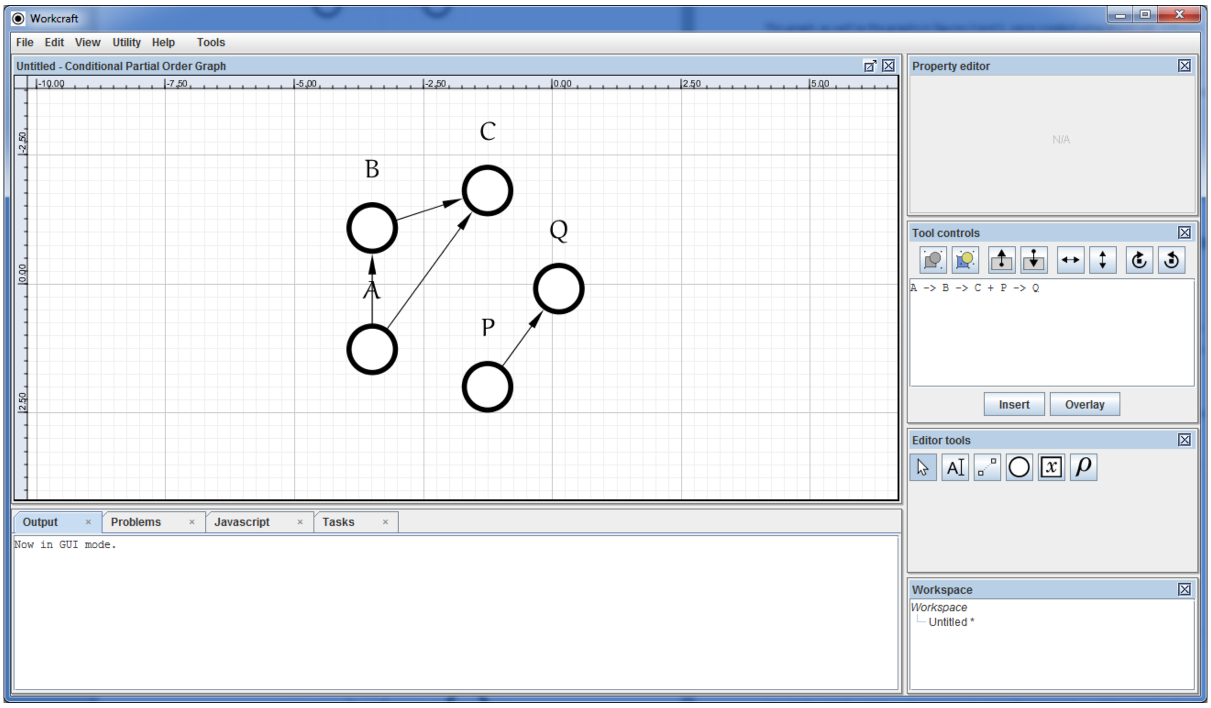
Figure 12 – A screen shot of Workcraft. A graph is displayed in its graph display

As figure 12 shows, Workcraft allows you to draw graphs. Using the "Editor Tools" on the tool bar to the right of the window, you can draw vertices, connect them, move them around the screen and add

Boolean and Rho clauses to them. This method makes drawing graphs much easier, and it makes them easier to store as well.

The property editor allows a user, when a vertex or edge is selected, to edit the properties of the selected element, such as its name, positioning in the graph, colour etc. All this is in aid of creating graphs which when designed are easier to use, such as with the ability to colour coordinate a graph for an easier understanding of how it works.

Tool Controls, from the tool bar on the right of the window, allows the user to have further options when using a control. For Conditional Partial Order Graphs the only tool with further controls is the “Select” tool, with the icon of a cursor. This has some tools to change the positioning of a part of, or all of, a graph, but it also provides a text box. This text box allows a user to enter the equation of a graph, and the software will create this graph.



```
A -> B -> C + P -> Q
```

```
Now in GUI mode.
```

Figure 13 – A screen shot of Workcraft, working with graph equation input.

In figure 13, I have entered the graph equation “Graph4” from figure 8. On pressing the “Insert” button, the software creates this graph. This is an important part of the software, as it allows for graph equations, such as those for Parameterised Graphs, to be entered into the software. Ideally, the project I am working on could use this text box, or another text box which works similarly, and allows several equations to be input, and calculated together.

As Workcraft is already used regularly for digital design, it would be useful to have a function such as the automation of the calculation of combining Parameterised Graphs added to this software, and for this reason, part of the objectives is to possibly add the software produced by this project to Workcraft. Thus, further useful functions that are used when designing a system are gathered in the same software package, removing the need for multiple software packages, which may not be compatible with one another. [4]

Technical Section

In this section, I will discuss the how the project was undertaken, from the initial workings, to its state at the end of the placement. The technical section will have several sub-sections:

- Project Specification
    - A detailed description of what is required of the software
- Design Flow
    - The process of how I created the software
- Results and Discussion
    - Proof of the software working, with reference to the properties of Parameterised Graphs.

3.1 Project Specification

Once I had learned about Paramterised Graphs, from how they work to how they are used, I was able to come up with a specification for the project I was to undertake, to give a guideline as to what should be expected of the software by the end of the placement.

There are several major features of the software which should exist for this project to be considered successful. These are:

1. Correctly take in parameterised graph equations, which may reference other equations, and expand them, performing the necessary operations to ensure all vertices and edges from the equation are accounted for.
2. Remove duplicate vertices and edges from the final equation, to reduce the length and complexity of output equations
3. Allow the use of Booleans, either undefined or defined as "True" or "False", using defined Booleans to find the correct value of Boolean equations in order to determine whether elements of equations are included in the final equation.
4. Perform all calculations on equations according to the properties defined for Parameterised Graphs.

All of these features are testable, which is useful for proof that they work correctly. A project which has all of these features I would consider to correctly perform the tasks required for the project, making it a successful project.

There are also some minor features, not critical to the operations of the project, which would exist to make the use of the software easier. These include:

5. Perform validation on input equations, including Boolean equations, and inform the user of any errors, in order to avoid output errors.
6. Provide a user interface which can display all graphs, vertices etc. to allow the user to view and create new graphs or vertices using the current ones, and set or unset Boolean variables.
7. Include the software as part of the currently used Workcraft software, to allow the use of its features in conjunction of the features of this software, such as the input of several graphs into Workcraft, which can then be used in calculations to create new graphs using the software produced by this project, and these can be viewed in a visual format by Workcraft.

The version of the project at the end of the placement included all of the major features of the specification to some extend, and in the following section, the design flow, I will aim to explain how I achieved this. There is a rather large lack of the extra features was mainly due to time constraints, following from the complexities of covering the major features. In the section “Critical Analysis of the Project”, I will discuss how these features and how I could have implemented them.

3.2 Design Flow

Creating software is a lengthy process, and a single feature is not usually implemented in a few lines of code in a single procedure. The features for this software needed to be broken down into smaller parts to be implemented and tested, usually including the previously implemented parts to help with the testing of the part currently being implemented.

This software I wrote in Java, as it is the programming language I have learned the most about during my time at University. It is an open source language, with lots of third party libraries, and help available on the internet to assist with any problem that you may have. For this project, I used the standard Java libraries, for which there is online documentation available from the company which develops Java, Oracle.

For the design flow, I will aim to describe broadly the way I wrote this software in the order of the methods, or group of methods, I wrote in order to complete this task. I will discuss their workings as in the version of the program at the end of the placement time. To start with, I aimed to create stand alone software, to allow it to be tested separately, and then when it worked as expected, I could add it to Workcraft to work with its features.

3.2.1 Finding Elements

Inputs to the software are to be in String form, with vertices being a set of alpha numeric characters, separated by the operators of either overlay ('+', ' '), or sequence ("->"). Brackets, '(' and ')' could also be included in this, so elements within these parenthesise need to be included when finding the elements.

I created a method called *findElements()*, with the purpose of taking in the equation of a string, and returning a list of the elements. The parameters are an *AbstractCollection* of the type string to list the elements, a string to hold the equation and a Boolean variable, used by some methods to include sequence operators in the list for their use. I will ignore the Boolean variable and its use in this function for now as it is used for a method I will discuss later, and is crucial to that methods performance. *findElements()* also returns an *AbstractCollection*

AbstractCollection is used in this case as it is the parent class from which two important list classes used in this software inherit. These are *HashSet* and *ArrayList*. Each is used according to the calling methods requirements from a list of the elements, and neither has a limit to its size.

A *HashSet* is a list of type *Set*. Sets do not allow multiples of the same element in a list, and I used *HashSet* as it is an implementable type of *Set*. A *HashSet* is used when only a list of the vertices is required, and the order is not needed to be known. [5]

An *ArrayList* is used when the order of the elements appear in an equation is required, and thus duplicates vertices are important to keep the order, as a vertex can appear in an equation more than once. [6]

Both work correctly in this method, as when adding a vertex to a list, the *add()* method is used, which both list types feature. When adding an element to an *ArrayList*, it is added at the next index with a null value, keeping the order they are added. *HashSet*s will attempt to add the element to the list, but if it already contains the same element, determined by its hash code, it will not add it, thus not adding a duplicate, but not affecting the list as it stands. This means that both list types can be used for their purposes using the same code, reducing the amount of code required.

The simplified operation of *findElements()* is as follows, starting from index 0 of the string:

1. Check the next character. If it is a bracket, go to 6.
2. Find the next operator ('+', ' ', "->", '('). If there are no more operators, go to 7.
3. Vertex is a substring of the equation, which is all characters since the previous operator, or index 0 if there is no previous operator.
4. Add this vertex to the list of elements. Get the character after this operator.

5. Go to 1.
6. Find the end of the brackets (')'). Run *findElements()* on the string between these brackets. Add the vertices from this bracket to the list. Find the next character after the brackets end. Go to 1.
7. No more operators means the last vertex has been reached. This is a substring from the last operator to the end of the string. Add this to the vertex list. Return the list.

The operation is designed simply to just find the elements, by finding the characters or list of characters between the operators. When a bracket is found, whatever is between the brackets, be it a single vertex, or multiple edges and operands, if typed out correctly will act as a separate equation, thus calling *findElements()* recursively will find all the elements that are between the brackets, and these can then be added the list, and will keep the order if an *ArrayList* is used.

This method finds all the vertices in an equation. I felt this was an important start, as either an overlay operation requires the elements from one equation to either be included with the elements of another, and a sequence needs edges to connect the vertices of the first equation with the vertices of the second.

3.2.2 Calculations

The next method written was in to some extent the most important part, the method which actually performs the overlay and sequence operations. This method was called *calculate()*. It has 3 parameters, 2 graph equations, which were a class type created to encompass the name and equation of a graph, these are labelled g1 and g2, and the operator. The operator holds either '+' or "->" to represent overlay or sequence, and this is used to see which calculation to operate.

For this method, I used my knowledge of parameterised graphs to determine how these could be calculated. The graph equation input parameters would be needed to calculate the output, but how they are used is what changes. For an overlay, from the properties in figures 33 and 34 show that the two equations are just combined, so in an instance where the operator is a '+', the resulting value is simply the equation of the first graph, a '+' sign and then the string of the second equation. A simple way of performing the operation, however it holds with the properties.

Sequences however were slightly more complicated. Simply put, the properties state that in a sequence, all vertices in the first graph must have edges connecting them to all the vertices in the second graph, as well as the original equations themselves. This is performed using nested loops. All elements are found for both equations, using a *HashSet* as this helps to reduce duplicate vertices, and then a string is created, like with an overlay, with both of the original equations with a '+' between them to show them both being included in the final equation. Then, the outer loop gets the first vertex in the set of elements for the first equation, and adds to the result string this sequenced with every vertex in the second set of elements for the second equation, which the inner loop iterates through, by writing the vertex from equation one, a "->" sign and then the vertex from sequence two. This outer loop repeats this for all elements in the first equation set, and this ensures all vertices in the first equation are sequences with all the elements in the second equation. Each of these sequences is separated in the result string with a '+'.

The result string is then returned to the calling method for use in further calculation or storage.

3.2.3 Removing Duplicates

The *calculate()* method works well, yet the output it produces can be lengthy and full of duplicate sequences and vertices. This makes the output hard to read, longer to perform further calculations with and a larger waste of resources, so it is necessary to clean up the result by producing an equation with one of each sequence, and if a vertex does not appear in a sequence, then it must be overlayed in the equation to ensure it is included. This is perfomed by the *removeDuplicates()* method.

This is a matter of finding which sequences exist in the equation, and including these, and which ever vertices are not included in a sequence are also added. To do this, the another *HashSet* of the vertices in the equation is created using *findElements()* and then cloned so two lists of elements exist. More nested loops are used, iterating through all vertices in one list and checking the equation to see if there is a sequence to any vertices in the second list.

If the equation is found to contain at least one of these sequences, then it is added to a string to be returned at the end of the method. The vertices featuring in a sequence which is found in the equation are added to a separate *HashSet* which is used after all possible sequences have been checked, by iterating through the original list of vertices and comparing them to this list. Any which are in the list of vertices in the equation, but not in the list of vertices which feature in sequences are added to the string with overlays.

3.2.4 Booleans

At this point, without Booleans, the software could use overlay or sequence to calculate a number of parameterised graph algebraic equations, and simplify it to have each vertex appear at least once, being featured in the resulting equations multiple times only if they appear in more than one sequence. This was good progress, but a major part of parameterised graphs is Booleans.

At this point I started adding Boolean variables and equations to equations. They are surrounded by '[' and ']', and appear before a vertex. The software would recognise this as just part of the vertex, so further programming would be required to work with them. Boolean variables and equations would not be known to be different at this point, however a Boolean equation could feature three possible operators; '!' to represent a negation, '&' to represent an AND operation and '|' to represent an OR operation.

3.2.5 Condition Regularisation

$$[b_1]p \rightarrow [b_2]q = [b_1]p + [b_2]q + [b_1 \wedge b_2](p \rightarrow q)$$

Figure14 – The property of Condition Regularisation. '^' is an AND operation.

This is an important property of Booleans in parameterised graphs, as it determines under all conditions, whether or not a sequence or the vertices in the sequence are featured in the final equation. An issue with Booleans equations is that, in a sequence, if one of the Boolean equations is resolved to be false, the vertex cannot be removed, as one side of the sequence cannot just be an 'empty' vertex, the properties deem this to be unsuitable. Instead, it must be considered that both vertices will appear if their individual Boolean conditions are true, but the sequence will only appear if both these conditions are true. If both conditions are true, then the two separate vertices with their conditions can be removed from the final equation. This is only required of the conditions for each vertex are different.

This is a special property which I felt requires its own method, *removeDuplicates().* This takes an equation, and finds sequences in it. If a sequence contain vertices with Booleans, they are checked to see if they are the same. If so they are left as this will not be affected, but if these Boolean equations are different, then the sequence is replaced by the necessary equation, featuring the separate vertices and their Boolean equations, as well as the sequence, with the two Boolean equations ANDed, so that this sequence appears if both these Boolean equations are resolved to be true.

3.2.6 Expanding Equations

Next, I felt it necessary to allow for equations to be input which have brackets, to allow for simplified inputs. At this point, only *findElements()* can deal with '(' and ')' brackets in an equation, but what if an equation is set out as follows:

$$Gph1 = [b1](X \rightarrow (P + Q))$$

Figure 15 – An example equation featuring brackets.

The properties tell us that this would expand to:

$$Gph1 = [b1]X \rightarrow [b1]P + [b1]X \rightarrow [b1]Q$$

Figure 16 – The expanded version of the equation from figure 15

It may be less time consuming for a user to write several input equations similar to as in figure 15, or maybe this is just how they determined the equation of a graph. Either way, if a user inputs this type of equation, the software should be able to deal with it, maybe not in its current form, but providing it can convert it into a format that represents the exact same graph, and is useable by the software then it can use this equation.

expand() is the method I created to do this. It takes in an equation string and expands it in three stages, returning the result. First, it applies expands brackets with Booleans surrounding them, then it expands brackets which are part of a sequence, then it expands all other brackets, such as those with overlays, as this does not affect the equation inside the brackets.

When expanding brackets with Booleans, it finds the Boolean variable or equation, and the equation inside the bracket, and passes these into a separate method, *expandBooleans()*. This applies the Boolean equation to each vertex inside the bracket, making it a form that the software can use for further calculation.

In the case that a vertex is already subject to a Boolean equation, it applies the principle that the Boolean from outside the must be true in order for the equation inside the brackets to exist at all, and it does this by applying the Boolean to any vertex subject to a condition by ANDing the external Boolean with all internal Booleans. For example:

$$Gph2 = [b2](A + [b3]C)$$
$$Gph2 = [b2]A + [b3 \,\&\, b2]C$$

Figure 17 – An equation to show how *expand()* applies Booleans to vertices already subject to conditions

In figure 17, you can see that if "b2" is false, neither vertex A nor C would appear.

Next, *expand()* expands all sequences. It finds a "->" operator in an equation, and then finds all "->" operators following it, before the next '+'. Several sequences in a row need to be calculated in order to turn the equation into a useable form. It then finds the operands of the sequences, which can be a single vertex or a bracketed equation. Once it has all of these it can then calculate, starting with the left most operand, the whole sequence, using the *calculate()* method. As this only takes in 2 equations at a time, it loops, starting with the first operand, and sequences it to the second, and this result is then sequenced with the third operand, and so on until the entire sequence is calculated. The sequence is then replaced in the original equation with the result of this calculation.

Because of the workings of this method, only brackets which are surrounded by overlay operators will remain. In this case, the brackets can be removed, as overlays do not affect data within a bracket. For example:

$$Gph3 = (O \rightarrow P) + (Q + R + S)$$
$$Gph3 = O \rightarrow P + Q + R + S$$

Figure 18 – An equation to show how overlay operations work with brackets.

3.2.7 Class Hierarchy

At this point, several classes had been created in order to contain the necessary information for different parts of an equation. To this point, for the ease of writing the software, all methods had been created in the *Main* class, so it was time to implement the hierarchy in order for a class to encompass the necessary methods for manipulating its own data. This diagram shows the class hierarchy of this software:

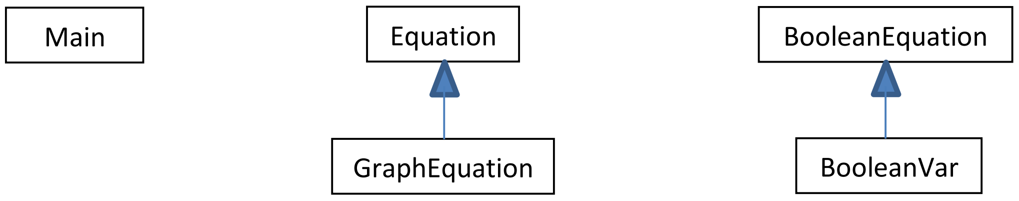


Figure 19 – Class Hierarchy of the software

This hierarchy is fairly simple, however for the application in its current state, it works correctly. It was originally intended to feature a “Vertex” class which would be used to identify vertices, for the purpose of listing all equations, vertices, Boolean equations and Boolean variables in a Graphical User Interface. However, it has a different interface and this is no longer required.

The *Main* class is used to start the program, and contains features to read and write the equations from and to files, as well as create and use objects from all other classes. It serves simply to run the main method sequentially from the execution of the application, until it has performed all calculations on all data in the input file, when it ends.

Equation contains the name of an equation, and the actual equation. It also features methods useful to convert an input equation into a version useable by the software. Originally this was intended to be a parent class to both *GraphEquation* and a *Vertex* class, with the idea that these are both types of equation, a graph equation containing multiple vertices, and a vertex being part of an equation, but as stated before, vertex was not needed in this implementation.

GraphEquation inherits from *Equation* and as such can use all methods featured in *Equation*. Initially, this was going to act as a class so that it was distinguishable as an equation and not a single vertex, useful for listing in a Graphical Interface, but as the implantation no longer requires this, it simply holds further methods for removing labels for other graph equations and simplifying Boolean equations and variables to make resolving them easier.

BooleanEquation holds the name and equation of a Boolean equation. It contains methods which are used when trying to resolve Boolean equations, such as validating them to check for tautologies and such, and reducing the number of variables in the equation.

BooleanVar inherits from *BooleanEquation,* as a Boolean variable is part of a Boolean equation. While *BooleanVar* does inherit the methods from *BooleanEquation*, it doesn't work in the same way. *BooleanVar* stores the label of a variable as the name and equation, but also has two Boolean variables used to determine whether or not the variable has a value, whether it is "set" or not, and if it is set, it holds the value, either '1' for "True" or '0' for "False".

This hierarchy could be adapted to make the code be less complex, such as removing *GraphEquation* and incorporating its methods into *Equation*. If there was a need to be able to distinguish between parts of the equation in a list, then this class could be more useful.

3.2.8 Removing Graph Labels

At this point, when the application runs, the user inputs the address of a text file containing multiple parameterised graph equations, and performs necessary calculations on these, and writes the results into another user defined text file. This implementation allows graphs which have been calculated be referenced in other graphs by their label, or name. In this case, at some point during the calculations these labels need to be removed and replace by the equations they represent instead of being treated as vertices.

When an equation has been fully expanded and all calculations have been performed on it, it is added to a list, in *Main* which contains all equations which have been calculated. Before any major calculations start on an equation, it is expanded including the graph equation labels, and then all *findElements()* is run to attain a list of the elements. Each item of this list is checked against the name of calculated equations, and if one is found to be the same, it needs to be removed and replaced by the equation of this label.

GraphEquation containts a method which takes in two strings, the first is the label of the equation which needs to be replaced, the second is the equation to replace it with. This replaces all graph labels, and then it is expanded further and calculations are performed on this new equation.

3.2.9 Resolving Booleans

Finally, as part of the requirements of the application, Boolean equations needed to be simplified as much as possible, and their value found if variables are set. Thus, the application was changed to allow Boolean variable declarations, as ‘1’ or ‘0’ for ‘True’ or ‘False’ respectively, in the input file.

I wrote a method in *GraphEquation*, called *simplifyBoolean()* to simplify Booleans equations as much as possible, to make resolving them easier. This equation will find the instances of a vertex in a graph equation, and gather all its Booleans together, and create one Boolean equation containing all the variables the vertex is conditional of.

Next, a method from *BooleanEquation* is called to resolve all Boolean equations in a graph equation and replace them with their value, or remove the vertex if their value is [0]. This is called *testBools()*. It starts by listing all the variables in an *ArrayList* to keep their order, as well as keeping the operators which are being used, as this is important to resolving them.

Each variable is then checked against the list of Booleans that were declared in the input file to see if they have a value. If they do, their name is removed from the list and replaced with their value. Once this is done, the list is resolved from the left for all of the same operator, until the operator changes. For example:

$$[A|B|C\&D|E|F]$$

Figure 20 – An example Boolean equation as would be used in this software

From figure 20, the software would resolve “A|B|C” to either a value, or as few variables as possible, then it would resolve the result of this with “&D” and the result of this with “|E|F”. This is not the most accurate method of resolving Booleans, but in the time I had when writing this part of the software, this was the best method I could achieve. Ideally, with more time, I could research some form of third party libraries which designed to resolve Booleans, or methods to write it myself.

Then the Booleans are resolved using *validateBools()*. This uses the rules shown in figure 35 to either find the value of the Boolean equation, or reduce it to as few variables as possible.

When *validateBools()* has finished its operations, the values are returned to *testBools()* which then, if the final value is ‘0’ will take steps to remove the vertex from the equation, if the value is ‘1’, will keep the vertex but remove the Boolean as it is not required to show the vertex appearance in the equation, or if the value is one or more undeclared Boolean variables in an equation it will replace the unresolved Boolean equation with the resolved one.

3.2.10 Full Operation

All of the above explains how each different requirement of the software works, but it needs to operate all at once in order to work on a file containing multiple Equations and Boolean variables. This is performed by the *main()* method in the *Main* class.

This method starts by acquiring the address of the text file to with the input graph equations and the address of the file to output the results to. It then reads in all the equations, without performing any calculations, creating objects of *GraphEquation* with each one and stores these in a list in the order they are written in the file. It then reads in all the Boolean variables, and creates *BooleanVar* objects with their values, and stores all these in another list.

Next, it loops, and performs the same for all graph equations in the list. First, it expands an equation, to remove brackets and apply all Boolean equations to vertices and graph labels if they exits. It then finds all the vertices in the equation and checks them agains the list of previously calculated graphs, and replaces any matching graph labels with their equations.

Once this is finished, it then expands again, as this performs the necessary calculations, and will apply Booleans which were applied to graph labels to their equations as well. Next the equation is checked for condition regularisation to ensure the output is correct according to the properties. Duplicates are then removed to reduce the length of the equation and make it easier for the next step, which is simplifying the Booleans. The variables are then replaced by their values if available and the Boolean equations are resolved.

The result is then fully calculated and written to the output file, as well as stored in the list of calculated graphs for use in further calcuations on other equations. When all equations have been calculated, the program closes the input and output files and terminates.

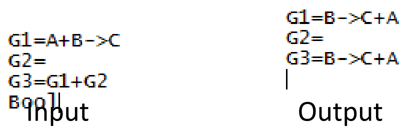
The code for this software follows this report, in the appendix, figure 38.

3.3 Results and Discussion

Software like this has many possibilities for error, as it is designed to calculate multiple equations, using other equations, which can include Booleans both declared and undeclared, two different operators, and all this has to produce results which apply to some properties in order for the output to be correct. For the testing of this program, I decided to test the properties that are available for it (shown in figures 33 and 34) while cumulatively adding each property test to the same input file, in order to prove that the software handles many equations. I will point out the important data in the screen shots from the output text file, and discuss what it means for the program.

3.3.1 Test 1 – Overlay Identity

This is designed to ensure that when an empty graph is overlayed with a none empty graph, then the none empty graph is still included in the final equation. This could occur when one graph has Boolean equations defined which mean all vertices in its graph are removed.



```
G1=A+B->C
G2=
G3=G1+G2
Bool
```

Input

```
G1=B->C+A
G2=
G3=B->C+A
```

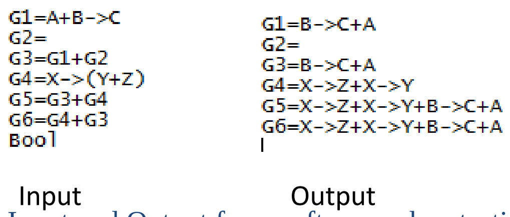
Output

Figure 21 – Input and Output from software when testing overlay identity

As you can see in Figure 21, G1 is a non empty graph and G2 is an empty graph, signified by having no vertices. When G1 is overlayed with G2 in G3, the output shows just the equation of G1, the order has changed to how it was written in the input file, however it is equivalent, this is due to using *HashSets*. This agrees with the overlay identity property.

3.3.2 Test 2 – Overlay Commutativity

The order of overlaying graphs is not important, and I will aim to prove this with this test



```
G1=A+B->C
G2=
G3=G1+G2
G4=X->(Y+Z)
G5=G3+G4
G6=G4+G3
Bool
```

Input

```
G1=B->C+A
G2=
G3=B->C+A
G4=X->Z+X->Y
G5=X->Z+X->Y+B->C+A
G6=X->Z+X->Y+B->C+A
```

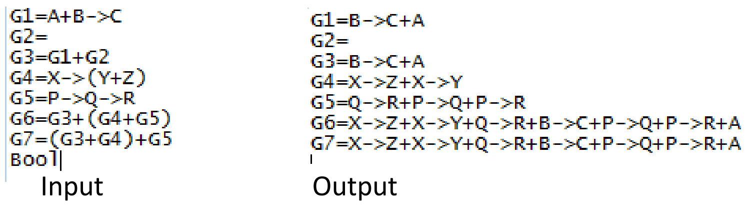
Output

Figure 22 – Input and Output from software when testing overlay commutativity

The input, shown in figure 22, uses previous equations to prove that the software can cope with multiple graph references. G5 and G6 overlay G3 and G4 in different orders, as you can see in the output, G5 and G6 are the same, proving that the order of overlaying vertices or graphs does not matter, and this software complies with this.

3.3.3 Test 3 – Overlay Associativity

Similar to the property of commutativity, when introducing brackets to an equation containing overlays, the location of the brackets in the equation doesn't matter.



```
G1=A+B->C
G2=
G3=G1+G2
G4=X->(Y+Z)
G5=P->Q->R
G6=G3+(G4+G5)
G7=(G3+G4)+G5
Bool
```

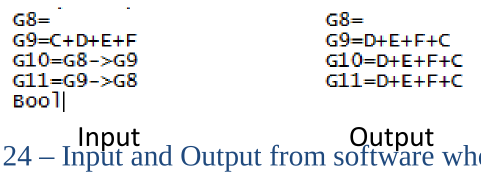
```
G1=B->C+A
G2=
G3=B->C+A
G4=X->Z+X->Y
G5=Q->R+P->Q+P->R
G6=X->Z+X->Y+Q->R+B->C+P->Q+P->R+A
G7=X->Z+X->Y+Q->R+B->C+P->Q+P->R+A
```

Figure 23 – Input and Output from software when testing overlay associativity

In figure 23, you can see that G6 and G7 both overlay three graphs, G3, G4 and G5, but G6 overlays G4 and G5 in brackets, and G7 overlays G3 and G4 in brackets. The right hand screen capture, the output, shows that G6 and G7 produce the same equations, and this is what is expected. The software therefore works with the property of overlay associativity.

3.3.4 Test 4 – Sequence Identities

Sequences have their own properties, and these need their own testing. For these tests, I will keep the same equations as the previous test in the input file, but use new equations to test sequence properties.



```
G8=
G9=C+D+E+F
G10=G8->G9
G11=G9->G8
Bool
```

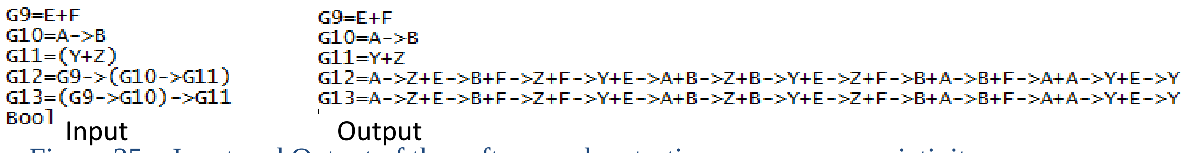
```
G8=
G9=D+E+F+C
G10=D+E+F+C
G11=D+E+F+C
```

Figure 24 – Input and Output from software when testing sequence identities

Figure 24 shows the input and output for this test. G8 is an empty equation, G9 is not. G10 and G11 sequence these graph equations in different order, as shown in the input, and the output shows that G10 and G11 produce the same equation. This proves that the software complies with the property of sequence identities.

3.3.5 Test 5 – Sequence Associativity

Similar to overlay associativity, this property exists to prove that bracket placement in sequencing multiple graphs or vertices do not change the order of the output. This software will break down the sequences into each individual sequence and overlay them all, but the output will not be affected by the bracket placement and should be the same.



```
G9=E+F
G10=A->B
G11=(Y+Z)
G12=G9->(G10->G11)
G13=(G9->G10)->G11
Bool
```

```
G9=E+F
G10=A->B
G11=Y+Z
G12=A->Z+E->B+F->Z+F->Y+E->A+B->Z+B->Y+E->Z+F->B+A->B+F->A+A->Y+E->Y
G13=A->Z+E->B+F->Z+F->Y+E->A+B->Z+B->Y+E->Z+F->B+A->B+F->A+A->Y+E->Y
```

Figure 25 – Input and Output of the software when testing sequence associativity

Figure 25 has the evidence for this test, and proves that this test does comply with the property of sequence associativity. G12 and G13 sequence G9, G10 and G11 in the same order, but G12 has brackets surround the sequence of G10 and G11, and G13 has the sequence of G9 and G10 surrounded by brackets. In the output, G12 and G13 both show the same equation, which is both the correct output for the sequences, and the same as each other, as expected from the software for it to be following this property.

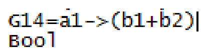
3.3.6 Test 6 – Left Distributivity

When both sequences and overlays are involved, the placement of brackets becomes important. In these next tests, I will use single equations to test the property, as multiple equations can make it complicated to explain and prove. I will also explain the expectation from the equation before the test.

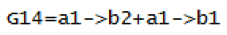
$$G14 = a1 \rightarrow (b1 + b2)$$
$$G14 = a1 \rightarrow b1 + a1 \rightarrow b2$$

Figure 26 – The expected expansion of G14 to test left distributivity.

According to this property, the sequence suggests that a1 will have an edge connecting it to each vertex within the bracket, in this case a1 will connect to b1 and b2.



```
G14=a1->(b1+b2)|
Bool
```



```
G14=a1->b2+a1->b1
```

Input Output

Figure 27 – The input and output of the software when testing left distributivity.

As shown in figure 27, the software expands the equation as expected, albeit in a different order to that shown in figure 26, yet these are equivalent.

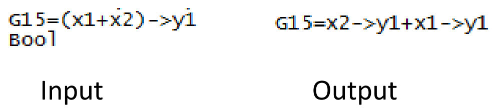
3.3.7 Test 7 – Right Distributivity

This test is similar to left distributive, yet the brackets are before the sequence in this property.

$$G15 = (x1 + x2) \rightarrow y1$$
$$G15 = x1 \rightarrow y1 + x2 \rightarrow y1$$

Figure 28 – The expected and expansion of G15 to test right distributivity

In this figure, the brackets being before the sequence means that all vertices in the bracket, in this case x1 and x2, will have an edge connecting them to y1, the vertex after the sequence.



```
G15=(x1+x2)->y1
Bool
```

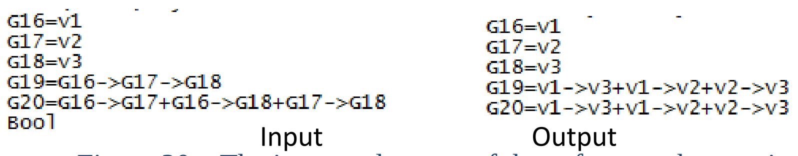
```
G15=x2->y1+x1->y1
```

Figure 29 – The input and output of the software when testing right distributivity

Again, the output shown in Figure 29 is in a different order to the expected output from Figure 28, but it is still equivalent, and this proves that the software works correctly with left and right distributivity.

3.3.8 Test 8 – Decomposition

This property proves that sequences can be written in different ways and be equivalent.



```
G16=v1
G17=v2
G18=v3
G19=G16->G17->G18
G20=G16->G17+G16->G18+G17->G18
Bool
```

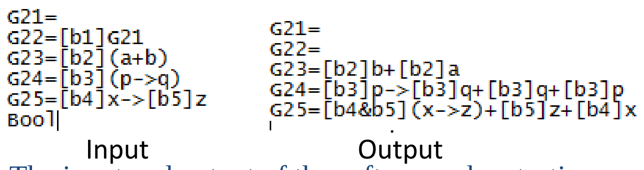
```
G16=v1
G17=v2
G18=v3
G19=v1->v3+v1->v2+v2->v3
G20=v1->v3+v1->v2+v2->v3
```

Figure 30 – The input and output of the software when testing decomposition

The input for this test shows three graphs G16, G17 and G18, each containing one vertex which are sequenced in G19 in order, and G20 contains the broken down version, with G16 having an edge to both G17 and G18, and G17 connecting to just G18. The output proves that both G19 and G20 are exactly the same output, and that the software correctly works with the decomposition property.

3.3.9 Test 9 – Conditional Properties

These properties can be easily tested at once. The Boolean variables I will leave undefined in order to show where the variables and equations are applied. I will test 3 properties; conditions with empty graphs, conditions with whole equations or brackets, and conditional regularisation.



```
G21=
G22=[b1]G21
G23=[b2](a+b)
G24=[b3](p->q)
G25=[b4]x->[b5]z
Bool
```

```
G21=
G22=
G23=[b2]b+[b2]a
G24=[b3]p->[b3]q+[b3]q+[b3]p
G25=[b4&b5](x->z)+[b5]z+[b4]x
```

Figure 31 – The input and output of the software when testing some condition properties

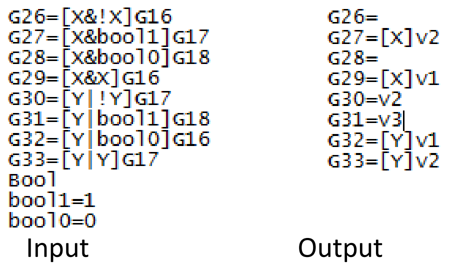
In figure 31, G21 and G22 are used to test conditions with empty graphs. A Boolean variable being applied to the empty equation G21 causes an empty graph equation to be output, as expected according to the property.

G23 has the Boolean variable b2 applied to a sequence bracket, and this correctly expands to apply to each vertex within the bracket. This also applies to G24 with a sequence. However the while the output does apply to both the vertices, some issue within the software also causes both the vertices to be added as overlays at the end of the output equation, both still both the Boolean variable b3 being dependent of them. While this is correct, the overlays are unnecessary and is part of some validation of output.

G25 is designed to check for condition regularisation. As I have stated before, condition regularisation is necessary to ensure that all possibilities of conditions is accounted for, and this is proven to be working in this software, by the sequence being conditional of both b4 and b5 being true, and having the vertices overlayed with their own Boolean variables, b5 with z and b4 with x, in case either of these variables are false, meaning the sequence will not appear in the final equation.

<u>3.3.10 Test 10 – Boolean Equation validation properties</u>

It is also required that I test the Boolean properties I used, shown in figure 35 for some simple Boolean validation. Again, it is possible to test together, and this will feature some declarations of variables. This also will test whether a vertex is output or not depending on how a Boolean equation is resolved.



```
G26=[X&!X]G16          G26=
G27=[X&bool1]G17       G27=[X]v2
G28=[X&bool0]G18       G28=
G29=[X&X]G16           G29=[X]v1
G30=[Y|!Y]G17          G30=v2
G31=[Y|bool1]G18       G31=v3
G32=[Y|bool0]G16       G32=[Y]v1
G33=[Y|Y]G17           G33=[Y]v2
Bool
bool1=1
bool0=0
  Input                  Output
```

Figure 32 – Input and Output from the software when testing the Boolean equation properties

In figure 32, the Boolean declarations follow the word "Bool", and there are 2 which are used to represent either 1 or 0 when testing the properties.

Comparing the Boolean equations from the input in figure 32 and properties in figure 35, the first property is proved by G26. ANDing a variable and its negation makes it impossible for the output to be true, resolving this to false, and thus in the output G26 is empty, as expected. G27 is the AND of a 1 and X, which resolves to just X and this is reflected in the output. ANDing any variable with a 0 resolves to 0, or false, and thus this should cause the vertex to not appear in the output, which it does not. G29 also proves to agree with the property, as performing an AND operation with the same variable is just the same as just using the same variable, and this is reflected in the output by just having the vertex be conditional on the Boolean variable x.

The or properties also work correctly. An OR operation involving a variable and its negation always resolves to 1, as in G30, thus just the vertex appears in the output with no Boolean variable or equation applied to it. G31 has a Boolean equation involving a variable defined as 1, or true. If an OR operation has a true value, the output is true, thus, just as before, just the vertex appears in the output. G32 involves a Boolean variable defined as 0, and thus, this can be removed from the equation, meaning the equation relies on Y to determine whether the vertex appears in the output, and this is reflected. Finally, G33 involves an OR operation with the same variable twice, which resolves to just one of this variable, in this case Y, and so the output shows the vertex being conditional on Y.

These tests show that the software in general applies these properties during its calculations, and this is important for correct calculations. In one case the output, while correct, has extra terms, and this shows an issue in simplification of equations, and can be corrected without causing incorrect output. In multiple cases, the output is in a strange order, due to *HashSet* lists ordering it in order of each objects hash code.

However, testing the properties of parameterised graphs does not reflect on how well the software could work when being used by an end user who wishes to use it to calculate multiple long and complex equations is yet to be tested. Figures 36 and 37 in the appendix are the final input and output text files used and produced by the software for these tests.

Project Review

The point of the industrial project is to work on an actual project which will be used in its relevant field in the future, whether this is an incremental upgrade to existing methods, or a large overhaul requiring a whole new design, implementation and testing. The final project may not have to be complete, but it must be in a state from which the work on it can be carried on by other members of the company.

In this case, I created software to automate the calculations of multiple parameterised graphs, in an attempt to reduce the time required when designing and testing a digital system. However, whether the work I handed in at the end of the placement time was effective enough to be used or not needs to be discussed.

The final software seems to work quite well. In testing, I have not found any major problems, but further use of the software by its end users may prove to find issues of which I am not aware. As I am fairly new to using parameterised graphs, I have never used them in any form of actual application; I have simply used them to understand the algebra in order to create software to work with them. Thus, I have not used any real world data on the software to test its workings for a real application.

My supervisor, Dr Andrey Mokhov, have met many times over the placement in order to discuss the algebra, and demonstrate the current version of the software and how well it works. I have used these in order to find issues the software may have during the design time and correct them, as well as discuss what the next feature to add to the software should me.

Dr Mokhov has received and tested the version of the software available at the end of the placement, and we have discussed how well it works, however I am not sure as to how complex the data he has tested is. He has however stated there are some Minor issues, such as all duplicates not being removed, and some extra overlay operators, '+', being left at the end of output equations, and has expressed a wish for the equations to be ordered better, such as alphabetically, in order to make the reading of the output equations simpler.

I believe that as a base for further development, the software I have created is very good. From its state at the end of placement, it works well, and bugs can be fixed and features can be added in the future, and I am happy to help Dr Mokhov do this. Once he deems that it works well enough, it can be incorporated into Workcraft. This is a slightly more challenging task, but with the code to perform certain operations there, it can be changed to deal with the input validation that exists already in Workcraft.

As a newcomer to a subject such as parameterised graphs, I think the work is good, and whether or not it is used by Dr Mokhov and colleagues in his field, I think it proves that software can be written to automate this process and gives hope for such software to be readily used in the future, hopefully with my work as a stepping stone to better, more easily used software.

Critical Analysis of the Project

This section is used to discuss what my thoughts on the work I did were, regardless of how well it performs for its intended application. In order to create better software in the future, I aim to analyse the software and discuss what I could improve.

The biggest weakness of the software was the way in which equations were handled. Each equation was input, stored and edited as a string. This I feel caused issues as string manipulation can be tricky, as it required adding, removing and editing substrings of equations. This meant that indexes of the start and ends of substrings were needed to be stored, and changed when the data changed. In some cases, where possible, I aimed to change this by using lists of vertices and parts of the equations, as it was easier to perform operations on smaller strings than larger ones.

A list of vertices and their operators could have been easier to work with, yet there is still the issue of indexes becoming an issue. But for example, a list could be created listing all the vertices between overlay, '+', operations. This means that some list elements would be just vertices, and some would be sequences. Since sequences require more calculation, such as decomposition, a sequence could be used in calculations, and then the index of the list the original sequence was stored in could be replaced by the decomposed sequences from that index, and the end index of this would then be the original index plus the number of additions to the list.

The string version of this operation would require first finding this sequence in the string, both start and end index, and then after the calculation was done, removing this part of the string, and replacing it with the decomposed version, and then finding the end of this new substring in order to continue for the rest of the equation.

Another weakness I found is that I didn't use the available resources. Workcraft is open source software, and already features validation of text input to convert equations into graphs. This feature could have been used, albeit with some tweaks, to ensure validation of inputs for this software, as well as kept the formatting of equations used by both my software and Workcraft the same, to make integration at a later date much easier.

Finally, another large problem I feel could have been avoided is Boolean validation. The method I used is good enough to work with simple Boolean equations, but more complex ones may be a problem. The point I started working on this in the placement was quite late on. If I had though ahead earlier on I could have started researching methods to perform validation on more complex equations such as third party libraries, and adapted them to this software. The time I had when I started working on this would have meant that doing this would have caused a more uncompleted version of the software to be left at the end of the placement.

These weaknesses in the software I feel, with more time, I could correct. Considering that I was new to parameterised graphs, and then I wrote software to calculate parameterised graph equations to some accuracy, even with its issues, makes me proud of the work I have done. I aim to help my supervisor fix any issues he find, and I would be happy to work on this software in the future to make it work as well as possible, and have it become a standard feature of Workcraft, for users to benefit from the automated calculation of parameterised graphs.

Conclusion

Now that I have discussed how the software was written, proved that it works, discussed how well it works for its intended application and how I as the author of the software feel the project turned out, we can now discuss how successful the project was in comparison to what was expected at the start of the project, with reference to the aims and objectives, defined in the introduction to this report.

- Learn more about Parameterised Graphs, and the design of digital and embedded systems.

In the process of writing this software, I learned about parameterised graphs in order to write code which uses the algebra of parameterised graphs correctly, so this part was successful. I was, however, not able to use it in an application of actually designing a digital system as I had no real data to test the software with, so this part was unsuccessful. If I had more time, I'm sure I could learn how to use parameterised graphs to design a system, and apply my software to this application, but as I have not done this, this part of this aim was unfortunately unsuccessful.

- Create an application that can take several graph equations, and expand them, perform calculations with them and output results, all while keeping with the rules defined by Parameterised Graphs.

As has been proven with the data shown in the testing section 3.3, the software can correctly calculate the output of several equations, using either vertices or previously calculated graphs, and the output correctly conforms to the properties defined by the algebra of parameterised graphs. This is a major objective of the project, and it has been successfully carried out.

- Add to the application the ability to use Boolean variables as part of the calculation process.

Again, testing has proven that Boolean variables and equations can be included in the input file, and these will be resolved and used to determine the final output of an equation, for both defined and undefined variables. Thus, this second major objective has been successful, to some extent. As I have stated previously, this only applies to simple Boolean equations, as the validation used when resolving Boolean variables is only simple. I count this objective therefore as successful, as the major point it to have Booleans be part of the calculation process, which they are, but further work is to be done in order to allow any form of Boolean equation be used.

- Add additional functionality for the purposes of making the usage of the software simpler.

In terms of additional functionality, the software has none. The additional functionality I expressed a wish to add, such as validation of whole equations, and including this software in Workcraft had not been considered by the end of the placement. The main objectives had taken enough of my time to make work correctly, or as well as I could in the time. As I have stated before in this report however, that if I had used the Workcraft code in the first place, then validation could have been included from the beginning, but this could have also been a large use of my time adapting it to work for my purposes, it is uncertain. This objective therefore can be considered unsuccessful.

In general, I would consider the project to be a success. The major objectives, writing the software and including Booleans that calculates equations correctly according to the properties of parameterised graphs, have been completed and tested. While learning more about the digital design process would have been interesting for me as an electronic and computer engineer, it was not required for the project, and neither was the extra functionality.

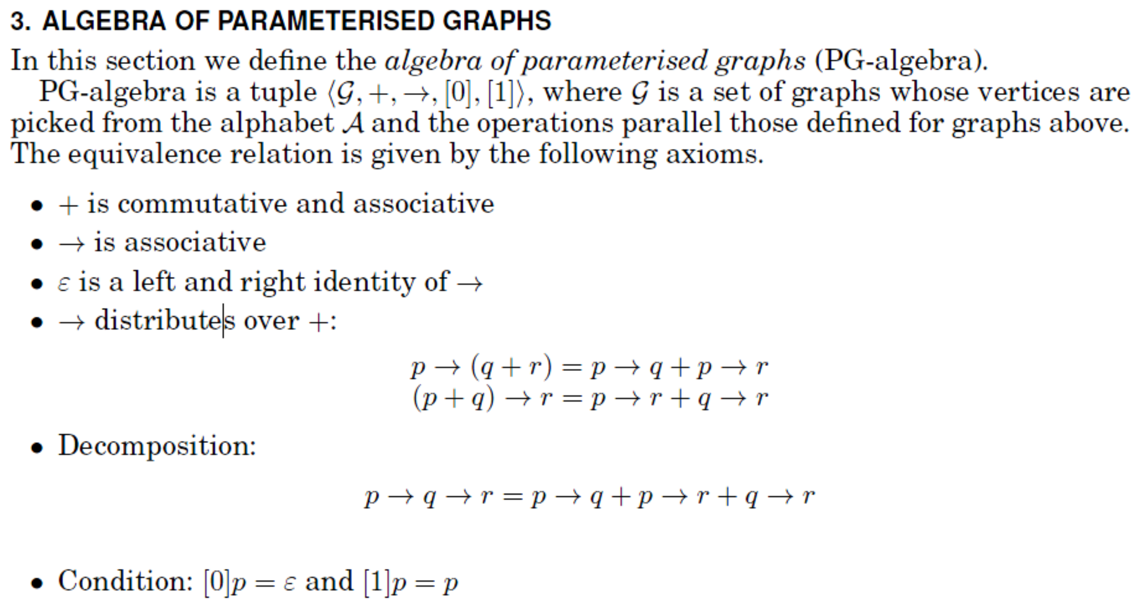
What remains now is the software to be tested by people who work with parameterised graphs regularly, and for them to use it in an attempt to make their time working with them easier. The outcome of this real world application testing will really determine the success of the project. Any bugs found with the code I am happy to fix, and I am happy to either work on adding this to Workcraft, or in either case I am happy to work with others aiming to make my software better.

Bibliography

[1] V. K. Andrey Mokhov, “Algebra of Parameterised Graphs,” pp. 1, 2013.
[2] V. K. Andrey Mokhov, “Algebra of Parameterised Graphs,” pp. 1 - 22, 2013.
[3] V. K. Andrey Mokhov, “Algebra of Paramterised Graphs,” pp. 1 - 10, 2012.
[4] "start Workcraft," 2014; http://workcraft.org.
[5] "Set (Java Platform SE 7)," 2014; http://docs.oracle.com/javase/7/docs/api/java/util/Set.html.
[6] "ArrayList (Java Platform SE7)," 2014; http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html.
[7] V. K. Andrey Mokhov, “Algebra of Parameterised Graphs,” pp. 8 - 9, 2013.

Appendix

Figure 33 – Properties of the algebra Parameterised Graphs. [7]



3. ALGEBRA OF PARAMETERISED GRAPHS

In this section we define the *algebra of parameterised graphs* (PG-algebra).

PG-algebra is a tuple $\langle \mathcal{G}, +, \rightarrow, [0], [1] \rangle$, where $\mathcal{G}$ is a set of graphs whose vertices are picked from the alphabet $\mathcal{A}$ and the operations parallel those defined for graphs above. The equivalence relation is given by the following axioms.

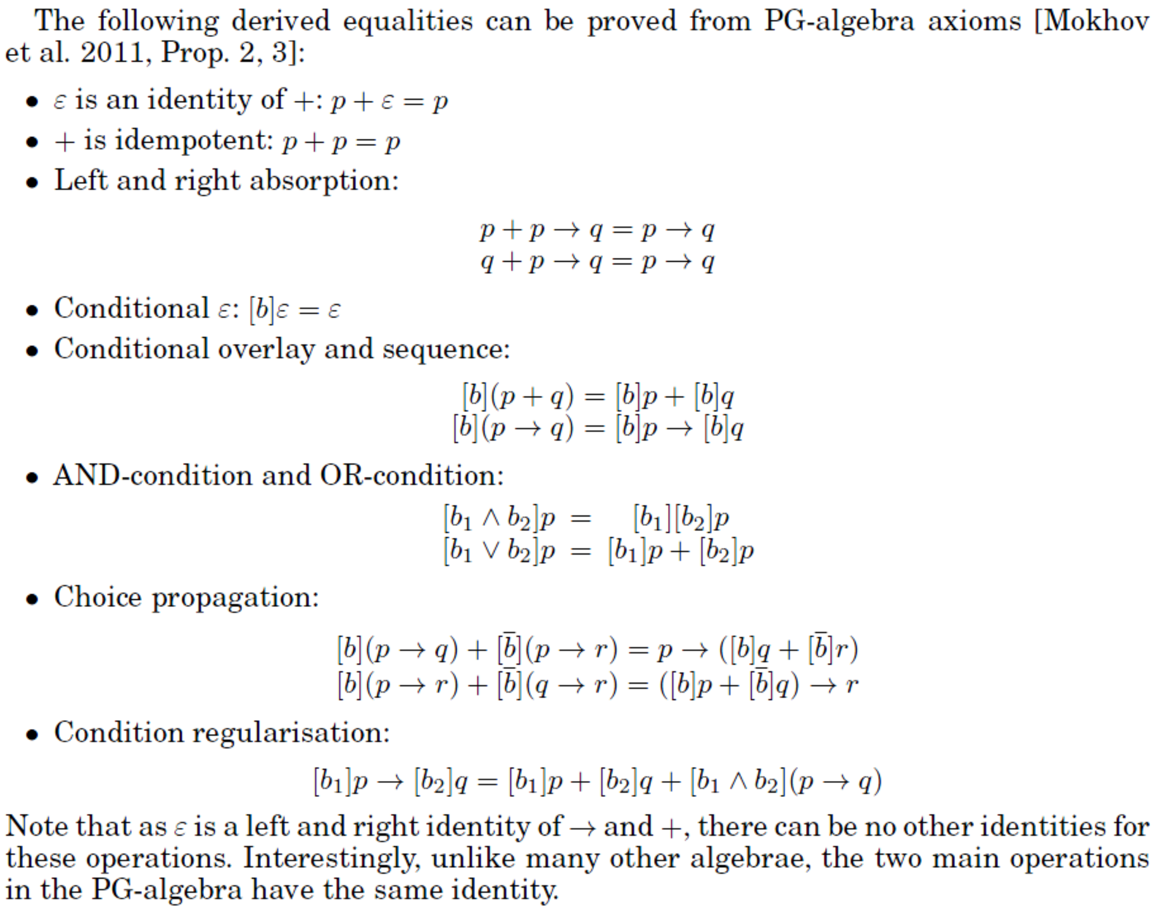
- $+$ is commutative and associative
- $\rightarrow$ is associative
- ε is a left and right identity of $\rightarrow$
- $\rightarrow$ distributes over $+$:

$$p \rightarrow (q + r) = p \rightarrow q + p \rightarrow r$$
$$(p + q) \rightarrow r = p \rightarrow r + q \rightarrow r$$

- Decomposition:

$$p \rightarrow q \rightarrow r = p \rightarrow q + p \rightarrow r + q \rightarrow r$$

- Condition: $[0]p = \varepsilon$ and $[1]p = p$



The following derived equalities can be proved from PG-algebra axioms [Mokhov et al. 2011, Prop. 2, 3]:

- ε is an identity of $+$: $p + \varepsilon = p$
- $+$ is idempotent: $p + p = p$
- Left and right absorption:

$$p + p \rightarrow q = p \rightarrow q$$
$$q + p \rightarrow q = p \rightarrow q$$

- Conditional ε: $[b]\varepsilon = \varepsilon$
- Conditional overlay and sequence:

$$[b](p + q) = [b]p + [b]q$$
$$[b](p \rightarrow q) = [b]p \rightarrow [b]q$$

- AND-condition and OR-condition:

$$\begin{aligned} [b_1 \wedge b_2]p &= [b_1][b_2]p \\ [b_1 \vee b_2]p &= [b_1]p + [b_2]p \end{aligned}$$

- Choice propagation:

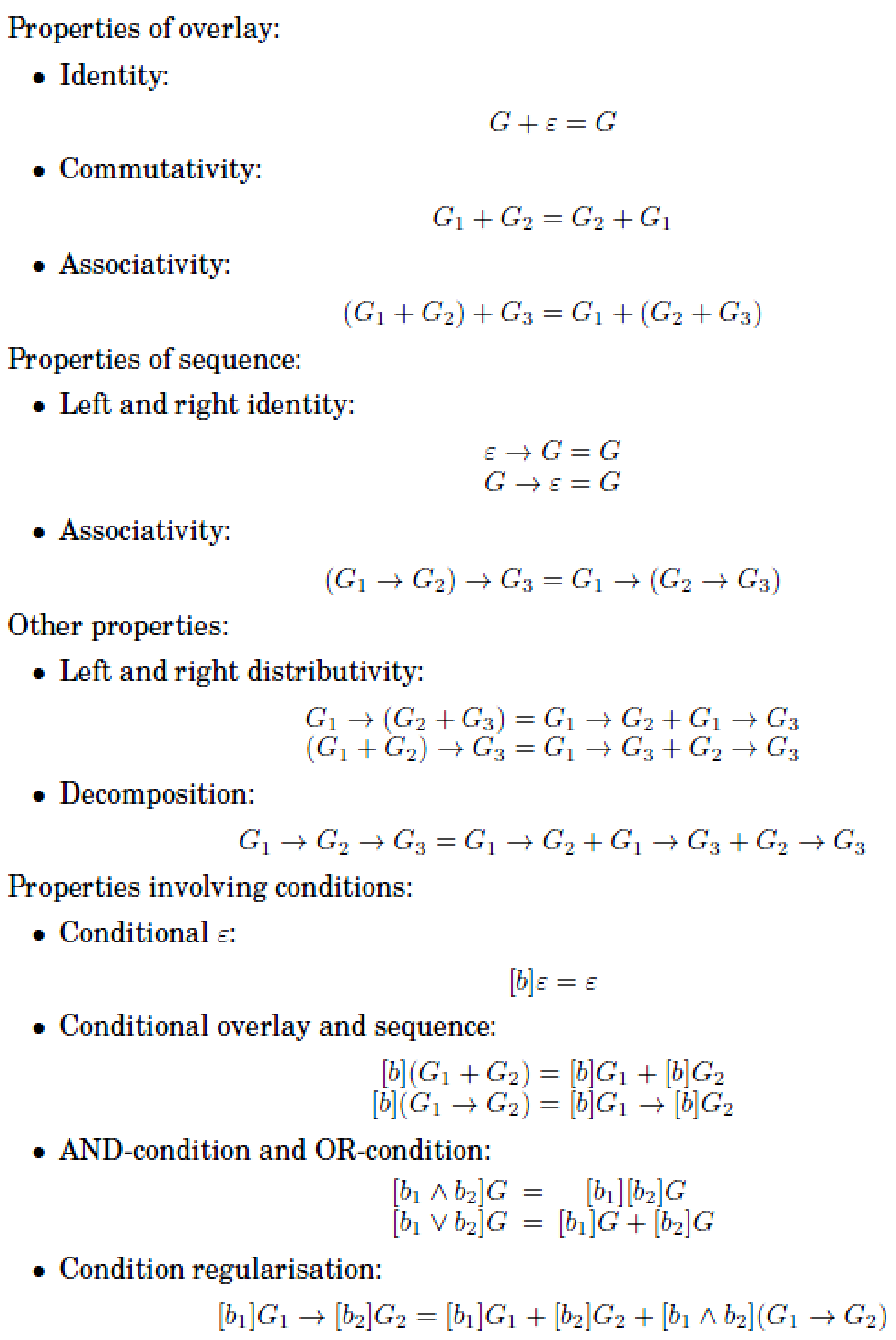
$$[b](p \rightarrow q) + [\overline{b}](p \rightarrow r) = p \rightarrow ([b]q + [\overline{b}]r)$$
$$[b](p \rightarrow r) + [\overline{b}](q \rightarrow r) = ([b]p + [\overline{b}]q) \rightarrow r$$

- Condition regularisation:

$$[b_1]p \rightarrow [b_2]q = [b_1]p + [b_2]q + [b_1 \wedge b_2](p \rightarrow q)$$

Note that as ε is a left and right identity of $\rightarrow$ and $+$, there can be no other identities for these operations. Interestingly, unlike many other algebrae, the two main operations in the PG-algebra have the same identity.

Figure 34 – Properties of Parameterised Graphs, when calculating using graphs, instead of just vertices



Properties of overlay:

- Identity:

$$G + \varepsilon = G$$

- Commutativity:

$$G_1 + G_2 = G_2 + G_1$$

- Associativity:

$$(G_1 + G_2) + G_3 = G_1 + (G_2 + G_3)$$

Properties of sequence:

- Left and right identity:

$$\varepsilon \rightarrow G = G$$
$$G \rightarrow \varepsilon = G$$

- Associativity:

$$(G_1 \rightarrow G_2) \rightarrow G_3 = G_1 \rightarrow (G_2 \rightarrow G_3)$$

Other properties:

- Left and right distributivity:

$$G_1 \rightarrow (G_2 + G_3) = G_1 \rightarrow G_2 + G_1 \rightarrow G_3$$
$$(G_1 + G_2) \rightarrow G_3 = G_1 \rightarrow G_3 + G_2 \rightarrow G_3$$

- Decomposition:

$$G_1 \rightarrow G_2 \rightarrow G_3 = G_1 \rightarrow G_2 + G_1 \rightarrow G_3 + G_2 \rightarrow G_3$$

Properties involving conditions:

- Conditional ε:

$$[b]\varepsilon = \varepsilon$$

- Conditional overlay and sequence:

$$[b](G_1 + G_2) = [b]G_1 + [b]G_2$$
$$[b](G_1 \rightarrow G_2) = [b]G_1 \rightarrow [b]G_2$$

- AND-condition and OR-condition:

$$\begin{aligned} [b_1 \wedge b_2]G &= [b_1][b_2]G \\ [b_1 \vee b_2]G &= [b_1]G + [b_2]G \end{aligned}$$

- Condition regularisation:

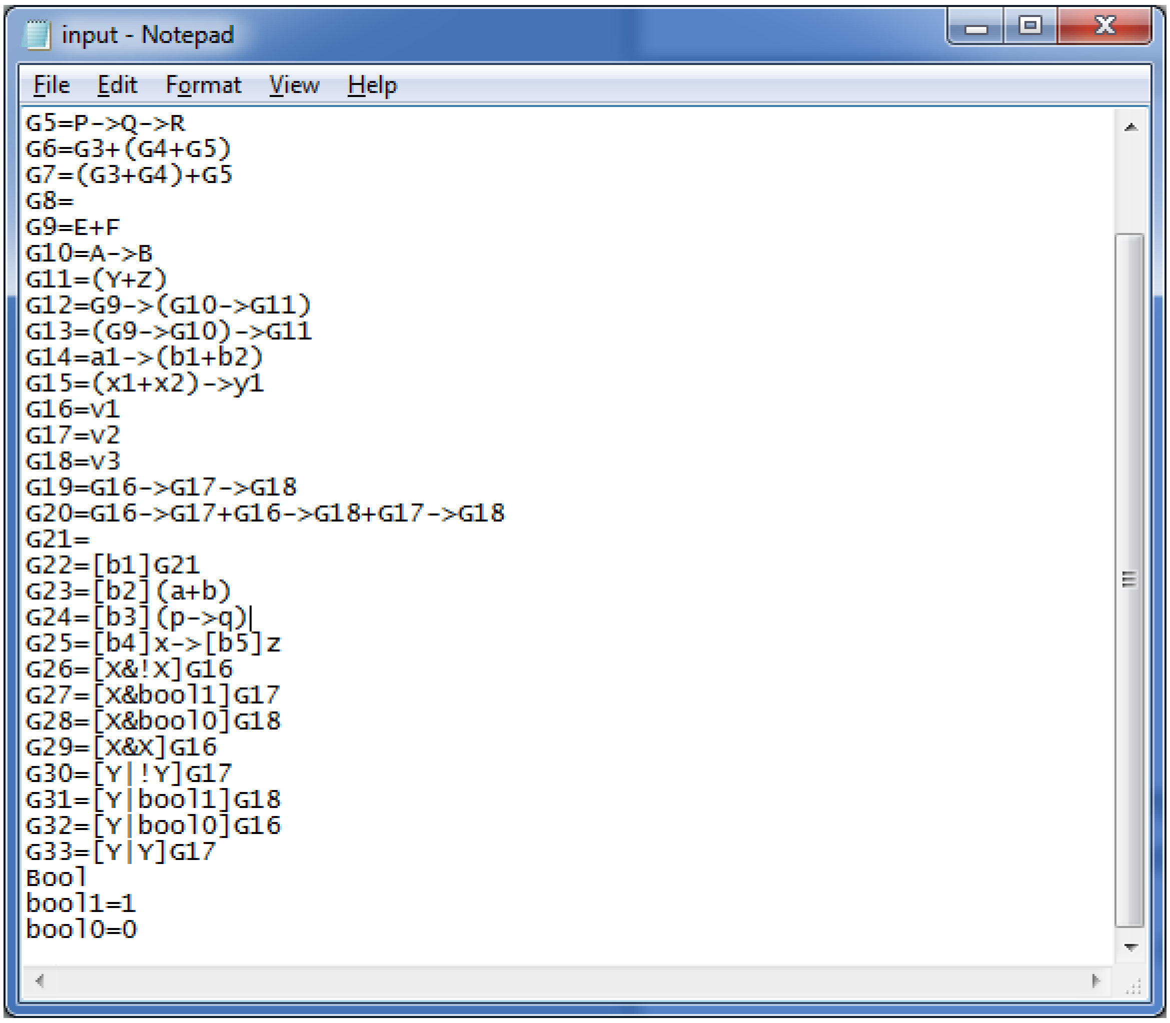
$$[b_1]G_1 \rightarrow [b_2]G_2 = [b_1]G_1 + [b_2]G_2 + [b_1 \wedge b_2](G_1 \rightarrow G_2)$$

Figure 35 – Properties of AND, OR including Negation for Boolean expressions

$$X \,\&\, !X = 0$$
$$X \,\&\, 1 = X$$
$$X \,\&\, 0 = 0$$
$$X \,\&\, X = X$$
$$Y \mid !Y = 1$$
$$Y \mid 1 = 1$$
$$Y \mid 0 = Y$$
$$Y \mid Y = Y$$

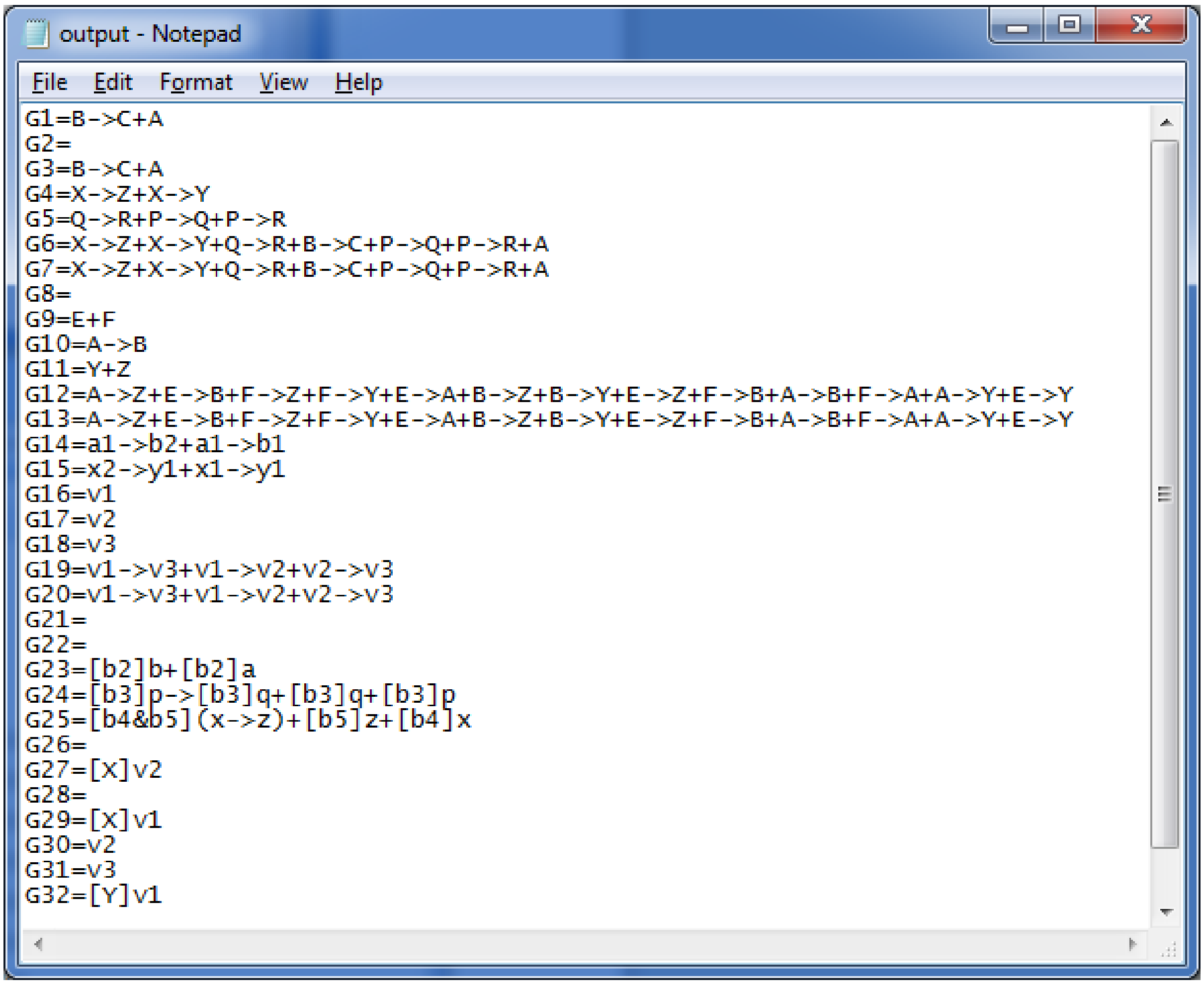
Here, X and Y represent Boolean variables, ‘&’ is an AND operation, ‘|’ is an OR operation and ‘!’ represents a negation of a variable value.

Figure 36 – Final input file used by the software in results and discussion



```
G5=P->Q->R
G6=G3+(G4+G5)
G7=(G3+G4)+G5
G8=
G9=E+F
G10=A->B
G11=(Y+Z)
G12=G9->(G10->G11)
G13=(G9->G10)->G11
G14=a1->(b1+b2)
G15=(x1+x2)->y1
G16=v1
G17=v2
G18=v3
G19=G16->G17->G18
G20=G16->G17+G16->G18+G17->G18
G21=
G22=[b1]G21
G23=[b2](a+b)
G24=[b3](p->q)
G25=[b4]x->[b5]z
G26=[X&!X]G16
G27=[X&bool1]G17
G28=[X&bool0]G18
G29=[X&X]G16
G30=[Y|!Y]G17
G31=[Y|bool1]G18
G32=[Y|bool0]G16
G33=[Y|Y]G17
Bool
bool1=1
bool0=0
```

Figure 37 – Final output file produced by the software in results and discussion



```
G1=B->C+A
G2=
G3=B->C+A
G4=X->Z+X->Y
G5=Q->R+P->Q+P->R
G6=X->Z+X->Y+Q->R+B->C+P->Q+P->R+A
G7=X->Z+X->Y+Q->R+B->C+P->Q+P->R+A
G8=
G9=E+F
G10=A->B
G11=Y+Z
G12=A->Z+E->B+F->Z+F->Y+E->A+B->Z+B->Y+E->Z+F->B+A->B+F->A+A->Y+E->Y
G13=A->Z+E->B+F->Z+F->Y+E->A+B->Z+B->Y+E->Z+F->B+A->B+F->A+A->Y+E->Y
G14=a1->b2+a1->b1
G15=x2->y1+x1->y1
G16=v1
G17=v2
G18=v3
G19=v1->v3+v1->v2+v2->v3
G20=v1->v3+v1->v2+v2->v3
G21=
G22=
G23=[b2]b+[b2]a
G24=[b3]p->[b3]q+[b3]q+[b3]p
G25=[b4&b5](x->z)+[b5]z+[b4]x
G26=
G27=[X]v2
G28=
G29=[X]v1
G30=v2
G31=v3
G32=[Y]v1
```