Department of Computer Science

Submitted in part fulfilment for the degree of BSc.

A Graphical Editor For GP 2

Samuel Hand

January 2019

Supervisor: Detlef Plump

For my parents - Russell and Rachael Hand

Acknowledgements

I would like to thank Detlef Plump, my project supervisor, for his patience and guidance over the course of this project. Thanks also to my parents, for their continued support and advice throughout both the project and my degree.

Contents

List of Figures

List of Tables

Executive Summary

This project presents a graphical editor for the graph programming language GP 2. GP 2 models computation by transforming an input, or "host" graph. A GP 2 program consists of a set of "rules" - descriptions of when and how to transform the host graph. The language also provides some simple control structures to manipulate the application of these rules.

The primary aim of the project was to produce an editor that provides an accessible, intuitive and robust user experience, specifically improving on some of the usability and technical problems with the existing editor, GP Developer. Having such an interface in the GP 2 development ecosystem would be valuable, as it would both aid newcomers in learning the language, and help existing GP 2 programmers work more effectively.

Additionally it was desired to produce an editor capable of converting to and from a common format for graphs, allowing users to integrate external tools more tightly with their workflow whilst working with GP 2.

Development of the tool took a user centered approach. Before any design phase began, stakeholders were identified and usage scenarios were constructed. These were then used to inform the design, ensuring that the produced editor actually met the user's needs.

For reasons of portability, it was decided to target the tool at the web. As robustness was a concern, the functional "compile-to-javascript" language elm was used for implementation, which promises to have "no runtime exceptions in practice".

The final outcome of the project consisted of two separate tools: a GP 2 rule editor and a GP 2 graph editor. These allowed setting all node and edge attributes currently available in GP 2. Sadly, there was not enough time to integrate these two components into a fully fledged IDE capable of producing and executing GP 2 programs.

Testing of the editors showed that they are significantly more robust than the existing GP Developer. They handle invalid inputs appropriately, and user feedback included no reports of any crashes or unexpected behaviour. In general, the feedback from users was positive, but identified several problems with the user interface that could be improved upon in the future.

Overall the project contributes a good first step to improving the provided

development environments for GP 2. However, more work remains to be done before the editor achieves all of the original aims of the project, and is a fully integrated environment capable of editing and executing GP 2 programs. Once this work is complete the project could be extended by first testing its existing visualisations, with comparisons to similar work, and then by implementing further, more dynamic visualisations, with the aim of allowing the user to gain a deeper understanding of the GP 2 system.

Statement of Ethics

The overall outcome of this project has no ethical implications. It is impossible to foresee any situation in which the editor could be used, intentionally or otherwise, in a harmful manner.

However, human subjects were used to test the application. All of these people were adults of sound mind, and provided informed consent to their feedback being included in the report. None of the feedback presented in the report identifies any individual or group of people.

1 Introduction

GP 2 [1] is a programming language developed at the University of York that performs computation with graph transformation. Programs are formed from a collection of rules: descriptions of how to modify certain parts of the structure of an input, or "host", graph.

1.1 Aims

Over the course of this project a graphical editor for the GP 2 language will be designed, implemented, and tested. This editor will be designed with a user centered approach, and should aim to provide a high quality user experience for the creation, modification, and execution of GP 2 programs and inputs.

Additionally the editor should be capable of working with graph formats non-native to GP 2. A user should be able to both export an existing rule or graph to a format for use in other applications, and import a graph produced using an external tool into the editor.

1.2 Motivation

There is already a graphical editor for GP 2, but this is old, unstable, and suffers from a range of usability problems. It is important that the development ecosystem for GP 2 provides a good user experience to ensure that the language is accessible to students and newcomers, whilst providing experts with the tools that they need to work efficiently.

Graphical languages are commonly used in domain specific contexts. Often such languages focus on providing intuitive visualisations for the described process, and also have mechanisms for preventing the user from ever producing an invalid program. Applying similar ideas to the design of the editor will hopefully lead to an intuitive utility that is useful both as a development environment and teaching tool for GP 2.

There are several tools available for working with graphs that provide functionality such as graph editing, visualisation and layout. Many of these tools support common graph formats such as DOT and GML. Whilst the conversion between these and the GP 2 format is relatively trivial, having the editor work with such formats directly would allow the user to more easily integrate GP 2 with other parts of their workflow.

2 Literature Review

It is important to have a strong understanding of the language constructs of GP 2 when designing a visual development environment for it. Such an understanding ensures that the environment can be designed in a way that is both intuitive and appropriate to the language. Literature detailing the design of GP 2 is reviewed, and a little mathematical background is provided (a more in depth explanation of the underlying mathematics can be found in [1], although is not required here).

An overview of existing approaches to visual programming is then given. This covers languages specifically based on graph transformation, and also relevant languages from other domains. Special attention is paid to the visual representations employed by these languages.

Finally, work relevant to the implementation and development of the environment is discussed, to ensure suitable tooling and procedural choices can be made throughout the project.

2.1 Graph Programming With GP 2

2.1.1 The Language

GP 2, the successor to GP [2], is a programming language that performs computation through a sequence of transformations on a *graph* [1]. A graph with label alphabet $\mathcal{L}$ is a system $\langle V, E, s, t, l, m \rangle$. V is the set of nodes and E the set of edges, $s : E \to V$ and $t : E \to V$ are functions that assign a source and target node to each edge, and finally $l : V \to \mathcal{L}$ and $m : E \to \mathcal{L}$ provide the nodes and edges with a labelling.

GP 2 uses the *double-pushout* approach for graph transformation. This performs graph transformations by matching and applying conditional rule schemata against the input (or *host*) graph in order to produce an output (or *derived*) graph.

A rule $\langle L \leftarrow K \rightarrow R \rangle$ is constructed from two graphs, L, and R. These share a common *interface* K which forms an inclusion on both the graphs. Additionally, a *condition* on the rule may be specified, which must be met in order for the rule to be applied. An example of such a rule can be seen

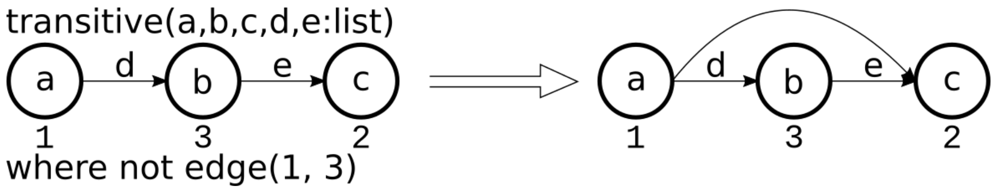


Figure 2.1: A GP 2 Rule.

in Figure 2.1, here the interface consists of all the nodes, and the curved edge is added by the rule.

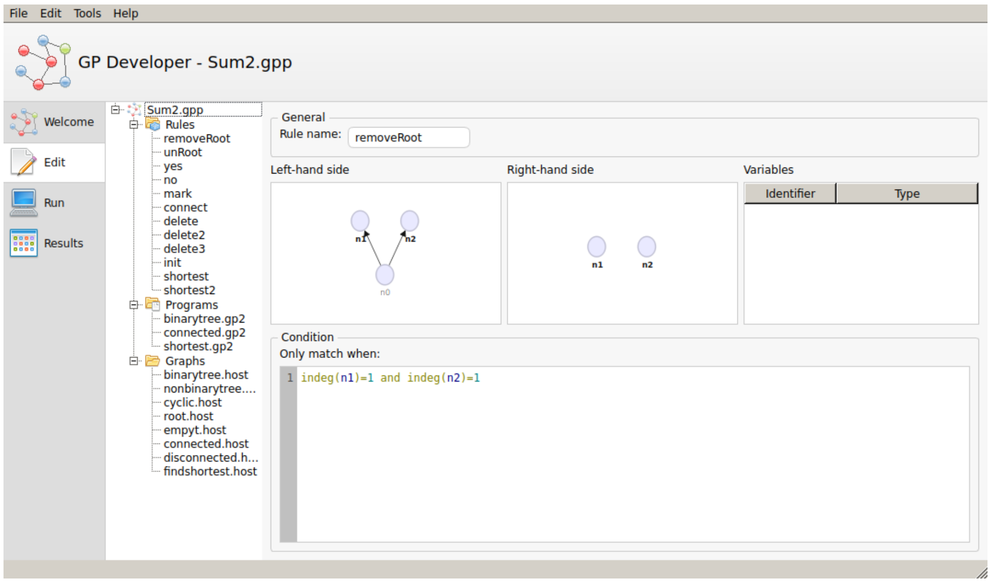
A brief outline of the transformation process is as follows. First, the *LHS* graph L is matched to the input graph G by means of an injective morphism - a structure preserving map that maintains distinctness, $g : L \to G$. Then $g(L - K)$ is deleted from G, as long as no node to be deleted has an incident edge outside of $g(L)$ - this is known as the dangling condition. Finally, all nodes and edges from $R - K$ are added back to the graph.

Each label in GP 2 is *typed*, and must either be a list, atom, integer, or string. The labels in rules may be expressions, which allow transformations to compute functions on labels. Variables in the LHS are pattern matched against the input, and may be used in expressions in the RHS. Note that the expressions in the LHS must be *simple*, meaning that they contain no composite expressions that would cause the pattern matching to be ambiguous.

GP 2 introduced the idea of *marking*. Nodes and edges may be marked in a certain way, and the condition on a rule can require a certain marking. This of course can be represented visually through the use of colours, or different styles of line.

Programs in GP 2 compose rules using a set of simple constructs: Sequential composition, with the `;` operator, as-long-as-possible application with `!`, non-deterministic choice from a set of rules listed between curly braces `{}`, and two conditional branching constructs: `try-then-else` and `if-then-else`. These both branch depending on whether or not an expression is applied successfully, with the difference that `if` discards the result of a successful application.

Such an approach to programming lends itself well to visualisation, as graphs already have a well understood and intuitive visual representation. Both the host graph and the rules can be drawn and manipulated visually, although more standard textual syntax is still required to express conditions on the rules, and the program itself.



Rule name: removeRoot

Only match when:

```
indeg(n1)=1 and indeg(n2)=1
```

Figure 2.2: The GP Developer interface with a rule open.

2.1.2 Tooling

There already exists a graphical IDE for GP 2, GP Developer [3], a screenshot of which can be seen in Figure 2.2. This provides an integrated visual environment in which to construct and execute GP 2 programs but is prone to crashing, lacks portability, and contains several bugs that negatively impact the user experience.

The system is implemented in C++ using the QT graphical toolkit. The graph editor itself is bespoke, but utilises the OGDF library for graph layout. These choices do not prevent the application from being portable, however several instances of platform specific code were identified in the current version of the tool. At the time of writing, support is only provided for Linux.

GP Developer implements several interfaces for working with the components of a GP 2 program. A simple text editor is provided for the programs themselves, and a graph editor and visualiser is used for the host and derived graphs. The rule editor makes use of both of these, presenting the user with two side by side graph editors for the LHS and RHS graphs, and a text editor for the condition on the rule. The implementation described by [3] automatically adds a corresponding node or edge to the RHS whenever one is created in the LHS. It also visually indicates the modifications made by the RHS through the use of colouring. This functionality is absent from the current version of the tool, and the only indication of what elements are in the interface is provided by the IDs displayed below nodes.

GP Developer provides an improvement on working with GP 2 purely with a text editor and the command line compiler. However, its user experience is marred by a number of bugs. Most severely, the editor crashes regularly - it did so several times during testing, causing work to be lost. The internal state can also become inconsistent with the interface, requiring the application to be restarted. The editor also doesn't prevent the user from creating invalid rules and graphs, but is unable to parse them, resulting in very odd behaviour when the editor is restarted. Finally, and with lowest severity, it is possible to construct graphs with overlapping edges, meaning that one or more edges are not visible.

2.2 Related Work

Several other graph transformation systems come equipped with visual environments. Many of these are specifically aimed at the domain of model transformation - an approach to software engineering based on modelling the changing relationships between objects in a system [4].

Additionally, visual programming is also employed in other domains, such as education and media production. Such languages are often used to give non programmers access to more powerful abstractions, without them having to learn a fully fledged textual programming language.

2.2.1 GrGen

GrGen [5] is a graph transformation language with a focus on efficiency. It compiles to .NET code, which can then be included in other applications. Like GP 2, GrGen is typed, but unlike GP 2 it distinguishes between the base types for nodes and edges, and provides an inheritance model with custom types.

Rule editing in GrGen is purely textual, but it does provide a graph visualiser for the purposes of debugging, seen in Figure 2.3. This visualiser is capable of showing detailed steps of the graph rewriting process, which is especially useful in an educational context [6]. Rule application uses the single pushout approach by default, but the double pushout approach is available on request [7].

Rules in GrGen are composed of patterns, which specify a structure to match, and rewrite rules, which specify structure to either modify or replace. Patterns can contain NACs - negative application conditions, which describe structure that must not be present in order for a match to occur. This is analogous to GP 2's conditions.

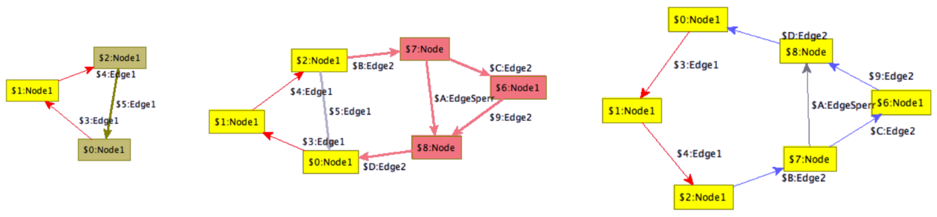


Figure 2.3: GrGen showing the rewrite steps of a rule. (Taken from [7].)

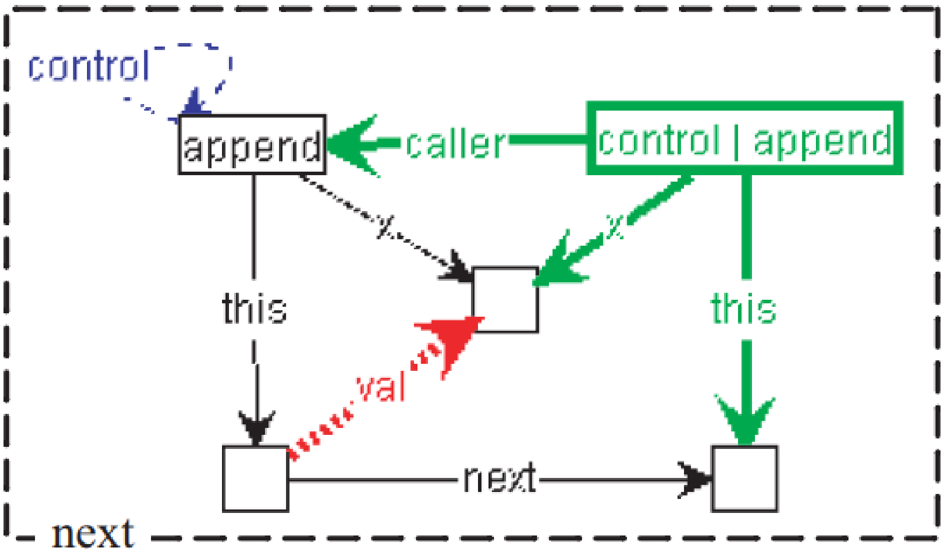


Figure 2.4: A rule in GROOVE. (Taken from [8].)

2.2.2 GROOVE

Rules in the GROOVE [8] model transformation tool take a very different form. As can be seen in Figure 2.4 they are composed of only a single graph, with multiple styles of nodes and edges for specifying the elements to be created and deleted, and the conditions under which the rule should not be applied.

The tool is also an interactive state space explorer. A user may experiment with different orders of rule application to discover reachable graphs, and the system is also capable of automatically exploring the entire space.

2.2.3 FUJABA

FUJABA (From Uml to Java And Back Again) automatically generates Java code from a combination of UML diagrams, *state charts*, and *collaboration diagrams* [9]. As the name suggests, FUJABA also aims to reconstruct

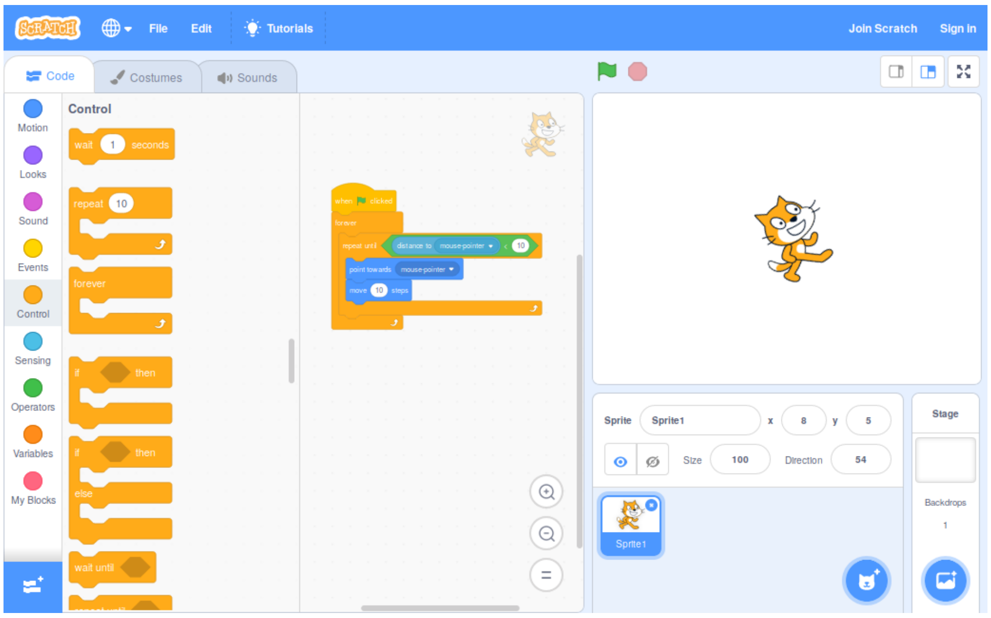


Figure 2.5: The scratch programming environment.

these diagrams from Java code.

As in GROOVE, rules are a single graph with styled elements, however the process by which rules are matched is rather different. The transformation rules (which are referred to as collaboration diagrams) form some of the nodes in a *state chart* - a flowchart that defines the behaviour of a particular object. Each rule contains a *this* node, which is matched directly to the object the state graph describes. This matching process is analogous to the *rooted* extension of GP 2 [10].

2.2.4 Scratch

The Scratch [11] programming language is a visual language designed for programming education, specifically for use with children. As can be seen in Figure 2.5 it has no textual syntax, and users create programs by combining pre-defined "blocks". It is impossible to create syntactically invalid programs, as invalid combinations of blocks will quite literally not fit together.

Scratch behaves in a dynamic manner. Groups of blocks can be selected with the mouse, and their effect will instantly be shown in the "stage" pane. Scripts can also be modified mid-execution, with feedback provided in a similarly instant manner. Such dynamism allows users to quickly prototype changes, and build an understanding for the behaviour of different blocks.

2.3 Representing Graphs

2.3.1 The DOT Format

Currently all graphs in GP 2 are stored in a simple internal format. First the list of nodes is specified, as pairs of identifiers and labels. Following these are the edges - tuples of identifiers, sources, targets, and labels [10].

Whilst the simplicity of this format makes it easy to work with, it is not compatible with other graph tools. Many such tools support common formats, such as DOT [12], the format used and popularised by the graphviz suite. Thankfully, conversion between the GP 2 format and these is a fairly trivial task.

A directed graph in DOT is stored as a list of connected nodes with optional attributes, specifying the styling and labels for the edges. A node may also be given attributes, by placing them on a line containing just the node and no connections.

2.3.2 Graph Layouts

When drawing graphs it is important to choose a layout that makes the structure of the graph easily understandable for the user. There are many so called *graph layout* algorithms that take an abstract representation of a graph and produce an intelligible concrete layout [13].

Of course, the degree to which a graph is understandable is subjective. Graph layout algorithms make use of metrics that typically correspond to the *simplicity* of the layout, and thus hopefully to how easy the graph is to interpret.

There are multiple libraries that provide access to graph layout algorithms. GP Developer utilises OGDF [14], and the Graphviz [15] suite, from which the DOT format originates, is also commonly used for this purpose.

2.4 User Centered Design

User centered design is a design process that aims to ensure user's needs are appropriately met by an interactive system [16]. This is achieved by consulting users as part of the design process, especially whilst gathering requirements.

A typical user centered design process begins by identifying stakeholders in the system. Once this is complete, a thorough investigation into their needs is performed, and used to determine the requirements for the system.

Design only begins once this process has been completed. Once a design is produced, users should then be asked for feedback, and the design can then be shaped according to their responses.

Ideally such a process should be iterative, slowly building on designs based on user feedback, until a solution is found that appropriately satisfies all the requirements.

3 Methodology

3.1 Approach

To ensure the deliverable for the project properly fulfils the requirements of the users it is important to follow standard software development and user centered design principles.

First stakeholders will be identified, and used to define usage scenarios that inform a set of requirements. Once the initial set of requirements has been decided the design phase will begin. After a design is produced stakeholders will be asked for feedback, and if time allows the design will then be iterated on.

3.2 Stakeholders

There are two main groups of stakeholders in the project: Students, who wish to learn GP 2 and gain an understanding of graph transformation principles in general, and researchers, who wish to develop and test graph transformation algorithms.

It is worth noting that the precise needs of these two groups may not be entirely compatible, necessitating compromise in the design.

3.2.1 Students

Most students will likely have limited experience with GP 2 and graph transformation principles in general. Their usage of the tool will have to provide them with the means to build intuitions about the behaviour of GP 2.

Students will primarily work with fairly simple rules and graphs. They may follow external documentation, and want to visualise and modify provided programs and graphs. These existing files may lack layout information.

As students will be developing an understanding of the GP 2 language it is important that they can experiment with the behaviour of programs by modifying and executing them on host graphs.

Students may require to work with graph programs on a range of machines, depending on what is available to them at home and in study spaces. These could include desktop computers, laptops, or even mobile touch-screen devices. These machines may run a range of different operating systems.

3.2.2 Researchers

Researchers will already have a strong understanding of the GP 2 system and graph transformation. The tool will have to augment their interaction with GP 2 without applying to many limitations on their existing workflow.

Researchers will design and construct more complex graph algorithms. As a result, some of the graphs that they work with may be of considerable size.

They will also have existing programs that need to be worked with, and will need to save their work in order to revisit it at a later date. It may also be required to export visualisations for inclusion in papers and other documents.

Finally, researchers will need to test and execute their programs on different inputs. They will then require to view and perhaps compare outputs of program executions.

3.3 Requirements

Now that stakeholders and usage scenarios have been defined, it is possible to produce a broad set of requirements for the tool. These are divided into four classes for the areas of the design that they affect.

3.3.1 System Requirements

The tool must be portable to every mainstream desktop and laptop operating system. It should also be accessible with several pointing devices, at the very least both mice and touchpads. The tool must also be highly robust. It should be an extremely rare event for a user to encounter a run-time error that causes them to lose work.

3.3.2 Editing and Visualisation Requirements

The tool must be capable of creating and editing host graphs and rules. It must support all GP 2 features, such as marks, rooted nodes and bidirectional edges.

For interacting with large graphs it must be possible to alter the zoom level of the graph, and pan to different locations.

The tool must provide a visual representation for both host graphs and rules. The representation for rules should make their behaviour intuitively understandable, allowing inexperienced users to easily identify the created and deleted elements.

3.3.3 File Input and Output Requirements

The tool must be able to open and save programs and graphs in the GP 2 textual formats. Additionally it should be possible to import and export to and from a format that allows for visualisation in external tools, such as graphviz or gephi.

When a graph or rule without layout information is opened, the tool should be capable of producing a layout to display it with.

3.3.4 Execution Requirements

It must be possible to produce programs in the tool, and then execute these programs on a set of inputs, and view the produced outputs, including failure states. Ideally it would be possible to not just view the completed output of a program, but explore the effects of individual rule applications.

4 Design

4.1 Scope

Due to the limited time scale for the project, the focus will initially be on producing a functional host graph and rule editor. If time allows, this will be expanded to a full program editor, fulfilling the "Execution" class of requirements. It is important to recognise that there may not be time to do this within the project period.

4.2 Platform Choice

There is a range of available platforms to which the tool could be targeted. As per the system requirements, whichever platform is chosen, the tool should be OS agnostic.

4.2.1 Native Implementation

The most flexible option would be to implement the tool using a native programming language, such as C++ or Python. Users would download a standalone program and run it on their machines.

This would likely allow for the development of a particularly fast tool, especially if a low level language was used. It would also provide the greatest flexibility with regards to available libraries and the ability to interact with the system

However, this option is the worst for the portability of the tool. Although it is possible to write OS portable code in such languages, the chances of behaviour being subtly different between different systems is high, and care would have to be taken to ensure all libraries (such as graphical toolkits) were completely consistent across all the supported systems.

4.2.2 Existing IDEs

Many IDEs allow for third party plugins or addons to be developed for them that leverage the existing functionality of the IDE for a new language or programming system.

For example, Eclipse, most commonly used as an IDE for Java, is the basis for the "Eclipse Modelling Framework", an IDE for Model Driven Development.

Although using such an IDE provides common functionality (such as file management) for free, the complexity of implementing specific GP 2 constructs would be increased due to the comparative inflexibility of the provided plugin architecture. The implementation would also be limited in control of the overall user interface of the editor, which could potentially hinder the ability to produce an intuitive experience.

4.2.3 The Web

The web is an almost ubiquitous platform for the distribution of content and applications. Every modern general purpose operating system has support for at least one web browser, making it the obvious choice for portability of the tool.

A web based solution would most likely be slower than a native implementation, as the application would be run entirely within a browser sandbox. The solution would also be limited in how it can interact with the rest of system, meaning that calling the GP 2 compiler and running a produced program would be challenging, if not impossible.

However, as the focus will initially be on producing just a graph and rule editor, the tool will be produced for the web, due to the unparalleled portability that it provides. Should the project get as far as implementing the execution of programs, a tool such as electron could be used. This uses the node.js runtime to allow access to native operating system APIs, whilst still maintaining portability.

4.3 Language and Tooling

4.3.1 JavaScript

The only language directly supported by modern web browsers for producing interactive pages is JavaScript. The language is dynamically typed, and has received criticism for the possibilities for uncaught errors that this creates.

Due to its state as a first class citizen within the browser, debugging JavaScript is a relatively trivial task. Most browsers include a JavaScript debugger in their developer tools by default. This is not always true of the many "compile to JavaScript" languages, used by those who wish to develop for the web in a different language.

4.3.2 Elm

Elm [17] is a purely functional, statically typed language, designed specifically for building web applications. It compiles to JavaScript. Elm provides a set of standard libraries for producing web applications, not unlike the "react.js" JavaScript library.

Additionally, Elm has a built in debugger, which includes "time-traveling" functionality, meaning that it can step backwards and inspect previous application states. This lessens the impact of not being able to use the built in browser debugger to its full extent.

Elm promises to "eliminate runtime errors". Whilst the veracity of such a bold claim is questionable, the nature of its type system means that errors are caught at compile time far more often than they would be with JavaScript.

As it is required to produce a robust and relatively runtime error free tool, Elm will be preferred over JavaScript. The developer is also more experienced with other functional languages such as Haskell, meaning that implementation will likely be easier and faster in Elm.

4.4 Pointing Devices

4.4.1 Mouse and Touchpads

As described in the requirements it is imperative that the editor supports both desktop and laptop computers. This means it must provide an interface suitable for being used with both mice and touchpads. Specific care will have to be taken with ensuring the editor is usable with the latter, as certain types of inputs are harder to perform on touchpads than they are on mice.

4.4.2 Touchscreens

Although it is anticipated most usage of the editor will happen on either desktop or laptop computers, it is not entirely inconceivable that a user may desire to be use it on a more portable touchscreen device. Focus will not be placed on providing such functionality, but it will be taken into consideration if time allows.

4.5 Rule Visualisation

There are many ways in which graph transformation rules can be visualised, some of which are used in the other tools discussed in the literature review. It is important that the chosen representation makes the behaviour of a rule visually clear.

4.5.1 Rules as single graphs

It is possible to represent a transformation rule as a single graph, as is done in GROOVE [8]. Created, deleted, and preserved elements are distinguished by means of different colours or shapes. It is not clear how to relabel preserved elements in such a representation, as they are only present in the rule once, so such an approach must be discarded for GP 2.

4.5.2 Colour coding

Alternatively a similar idea could be applied to a rule represented as two separate graphs. Colours would still be used to distinguish between preserved, created, and deleted parts of the structure, but preserved elements would be present in both sides and could have different labels in each.

However, as GP 2 already assigns meaning to different coloured elements with "marks", such an approach would be confusing. Shapes could be used as an alternative to different colours, but there is no pre-established visual convention for the meanings of these as there are with colours.

4.5.3 Preserved Structure

The chosen approach makes the preserved structure of a rule clear through the emboldening of user interface elements, and ensures that the preserved elements are always positioned identically in both the left and right side of the rules. The rationale for this is that it allows for differences between the sides of the rules to be seen at a glance.

As bold nodes already have a meaning in GP 2 in the form of roots, the emboldening of the ID drawn below a node will show that it is preserved. Additionally, the outline of nodes and the path colour of edges will be drawn in a darker shade than created and deleted edges. This will only apply to edges with the "none" and "dashed" mark, as otherwise the path colour will need to correspond to the mark. So that preserved marked edges are also clear, preserved edges will also be drawn at double width.

5 Implementation

5.1 User Interface

The final deliverable of the project consists of two separate interfaces: a host graph editor, and a rule editor. Both of these make use of a common graph editor component for visualising and interacting with graphs.

5.1.1 Graph Editor Component

The graph editor component provides all the graph editing functionality that is common between the host and rule editors. Graphs are rendered as SVGs, which are modified as the graph is interacted with.

The graph operations and their associated interface actions provided by the component are presented in Table 5.1. It is worth noting that the ability to set attributes (marks, labels, etc.) is not included in this common part of the application, as the implementation of such actions is specific to the graph and rule editors. However, as the component is responsible for the rendering of all graphs, it can draw graphs with both host and rule attributes.

Node layout in the graph editor component is purely manual, but edge pathing is done automatically. Multiple edges between the same nodes are disambiguated by varying the radius of the curve of each individual

Table 5.1: Graph editor interface

Operation	Action
Create Node	Double click background
Move Node	Left click and drag node
Create Edge	Right click and drag from source to target
Select Item	Left click item
Delete Selected Item	Press delete on keyboard
Zoom In/Out	Mouse wheel up/down
Pan	Left click and drag background

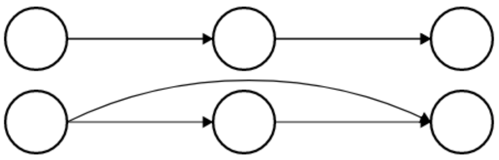


Figure 5.1: A comparison showing the effect of the edge curving heuristic.

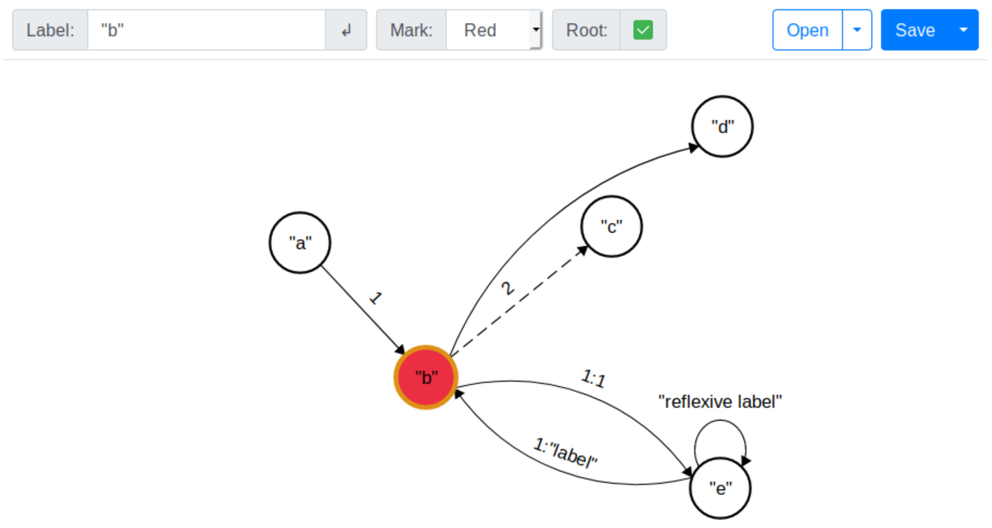


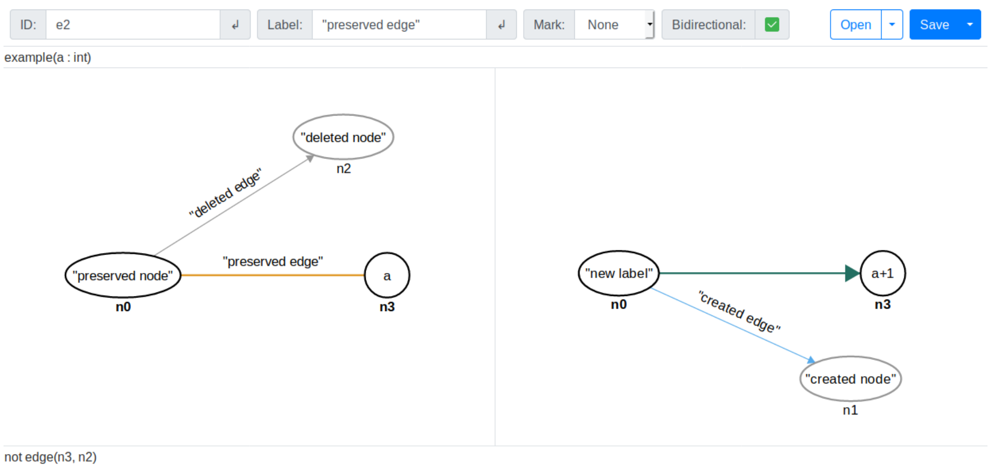
Figure 5.2: The host graph editor

edge. To prevent a situation where two or more edges completely overlap, a simple heuristic is used. Any edge that would otherwise be straight and passes through a node has a curve applied to it. The effect of this can be seen in Figure 5.1. The same graph is drawn twice, with the heuristic applied in the lower drawing only.

5.1.2 Host Graph Editor

The host editor consists of the common graph editor and a top toolbar. The graph editor is configured to not display node IDs, as these are not relevant in host graphs. The toolbar contains widgets for the editing of host graph item attributes, and two buttons for saving and opening graph files. A screenshot can be seen in Figure 5.2.

Selected items are highlighted in yellow, and can be edited with the toolbar. Changing the selected mark and modifying the root checkbox will update the selected item instantly. Updating the label requires an enter key-press or clicking the button to the side of the input. These actions are disabled when the inputted label cannot be parsed to prevent the creation of invalid graphs.



example(a : int)

not edge(n3, n2)

Figure 5.3: The rule editor

The open and save buttons can be clicked to upload and download graphs stored in the DOT format. Clicking the drop-down on the side of each button additionally provides the option to use the GP 2 textual format.

5.1.3 Rule Editor

As can be seen in Figure 5.3, the rule editor is similar in appearance to the host graph editor. The toolbar has an additional input for changing node and edge IDs, and there are two side by side graph editors for the left and right hand side of the rule. Unlike the host graph editor, these are both configured to display node IDs. Text-boxes above and below the graph editors contain the rule declaration, and the condition, respectively.

In the rule editor the toolbar allows setting the "any" mark, and also creating bidirectional edges. These attributes are not available in host graphs, as they are wildcards describing when a rule should match.

As described in the design chapter, the chosen rule visualisation positions preserved nodes identically in both sides of the rule. In order to maintain this constraint, moving an interface node on one side will also move it on the other.

Creating a node in the left side of the rule will create a node in the same position and with the same ID on the right side of the rule. The same is true for edges if both the source and target are in the interface. This means that changes made by the rule must be explicitly specified by modifying the right side.

The ID text box works in an equivalent manner to the label text box; only allowing a change to be committed if the input is valid. A node ID may only be set if it is not used in either side of the rule. If it is desired to have an interface node, it must be created in the left side of the rule. The restriction on edge IDs is less strict, only applying to IDs in the same side of the rule as the edge being edited. This allows for creating preserved edges that have different source and target nodes in each side of the rule.

5.2 Internals

5.2.1 Graph representation

Graphs are stored internally using the elm-community/graph library [18]. This represents graphs using adjacency matrices stored as fast mergeable integer maps [19] for efficient update and query.

The library allows arbitrary data to be stored as "labels" on graph elements. The datatype shown in Listing 1 labels both nodes and edges, and contains the GP 2 attributes of every element. When set for a node the boolean `flag` field indicates that the node is a root, and when on an edge it indicates if the edge is bidirectional.

```
type alias Label =
    { id    : String
    , label : String
    , mark  : Mark
    , flag  : Bool
    }
```

Listing 1: The "label" datatype used to store GP 2 attributes

Multiple edges with the same source and target nodes are not supported by the chosen graph library, so this is achieved by labeling each edge with a non empty list. Each item in this list represents a single edge between the source and target. If an operation would make the list on an edge empty, the edge is removed.

5.2.2 File Input and Output

Both editors allow for saving and opening rules and graphs in the GP 2 and DOT formats. Although the elm-graph library supports converting a graph into a DOT string, this would not work with the way in which multiple edges are implemented. Instead, the separate elm-dot-lang [20] library is used to build an abstract representation of a DOT graph from the internal graph, which can then be converted to a string. The entirety of the implementation for the parsing and output of GP 2 formatted graphs and rules is bespoke.

If either a DOT or GP 2 file is opened that does not contain position data for nodes it must be laid out before it can be displayed. A distribution of graphviz [15] compiled for the browser using emscripten [21] is used for this purpose. This distribution, known as viz.js [22] receives DOT strings and adds position information to them. A range of other output options are also supported, but these are not currently used by the editor. GP 2 files without layout information are first converted to DOT so that they can be laid out.

DOT graphs use attributes on nodes and edges to control their appearance. These attributes are used to encode the GP 2 attributes. Marks use the "fillcolour" attribute on nodes and the "colour" attribute on edges. (With the exception of the dashed mark, which changes the "style" attribute to "dashed"). Root nodes are given the "bold" style attribute, and bidirectional edges the "both" direction attribute.

In GP 2, rules are two distinct graphs. However, the editor outputs a single DOT graph file for rules, such as that seen in Figure 5.4. DOT attributes are set to distinguish between deleted, preserved, and created elements.

Deleted nodes are rectangles, created nodes are diamonds, and preserved nodes are the default ellipses. Preserved nodes may have a different label in the separate sides of the rule, and these two labels are both placed on the node, separated by a forward slash. A change in mark is displayed by colouring halves of the node separately. A node's rooted status in the left and right hand side is stored in the list of styles.

There is no concept of preserved edges in the DOT file, as a preserved edge can be connected to different nodes in the left and right side of the rule. A deleted (left hand side) edge has rectangular arrowheads, and a created (right hand side) edge has diamond arrowheads. Preserved edges will have the same ID, and this is disambiguated by prefixing the IDs with the side of the rule in which each edge is located.

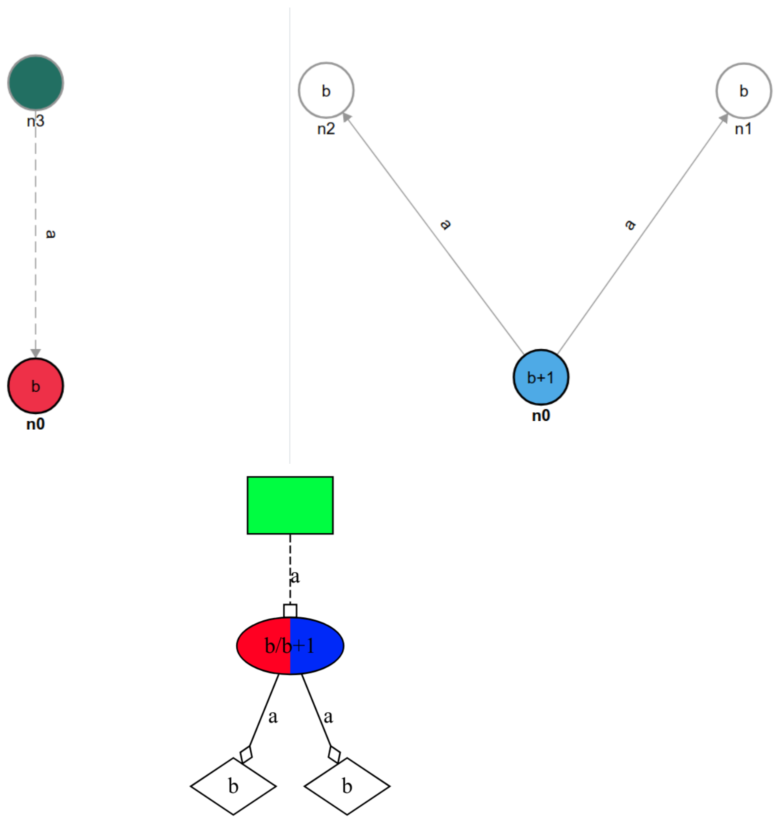


Figure 5.4: A rule in the GP 2 and DOT formats

6 Results and Evaluation

6.1 Tests

In order to analyse the robustness of the editor, several test cases were produced. These were tried on both GP Developer, and the new editor, so that a comparison could be made between the two. The full results of these tests can be viewed in Appendix A.

In every test case where an invalid input was provided, the editor responded by providing an error message, such as the one seen in Figure 6.2. Under the same test cases GP Developer behaved strangely, creating seemingly empty, nameless files. These files could not be fixed from within GP Developer, and necessitated usage of an external editor to correct. This behaviour can be seen in Figure 6.1.

It was possible to create invalid files from within GP Developer itself by entering invalid attributes when editing nodes and edges. In addition, certain invalid attributes would also cause the application to crash. The new editor did not suffer from any such problem, as it validates all input before updating the graph.

The editor did sometimes fail to open valid DOT inputs, due to limitations with the DOT parsing library that was used. This happened specifically with DOT files that set default attributes for nodes and edges, or global attributes for the entire graph. Ideally this library will be replaced in the future.

Overall the editor is significantly more robust than GP Developer. Although select valid inputs result in error cases, no way in which the editor could be caused to actually crash or otherwise fail critically without providing an error was discovered.

6.2 Stakeholder Feedback

As described in the methodology chapter, once the implementation was completed a selection of stakeholders were asked to use the tool and provide feedback. Students were selected who had limited or no experience with GP 2 and graph theory, and several researchers who work on GP 2

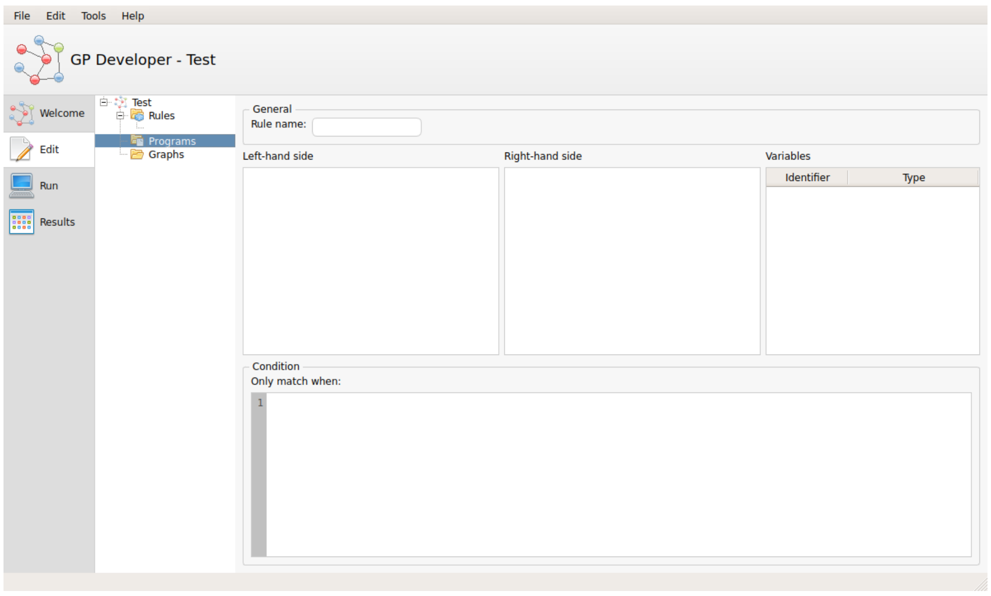
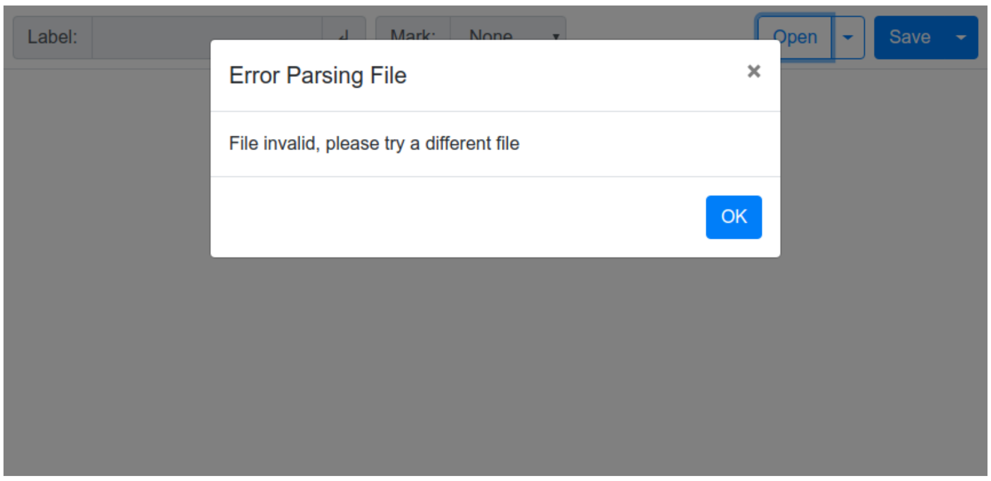


Figure 6.1: GP Developer upon opening an invalid rule



Error Parsing File

File invalid, please try a different file

Figure 6.2: The editor showing an error when an invalid file is opened

within the department were contacted.

Without being given prior instruction most of the testers did not initially understand how to interact with the editor. Although some found the actions with which nodes and edges are created through experimentation, the majority had to be told how to do this. Several users did comment that the interface was easy to use and allowed them to work quickly, but only once the initial hurdle of feature discovery was overcome.

Having to confirm label and ID inputs with the enter key caused some confusion. Many selected a node or edge, inputted a valid label or ID, and then deselected the element without pressing enter, and were surprised to discover that the label or ID had in fact not been changed.

Testers also found the lack of descriptive error messages provided by the application unhelpful. This was especially true for students who attempted to enter an invalid label, as they were unaware of the precise grammar rules for GP 2 labels, and no information is provided as to why the label is invalid.

Students were given access to a small amount of literature on GP 2 before testing the application, so had at least a basic concept of a rule. Despite this, many of them were initially confused by the rule editor, and didn't realise that the two panes corresponded to the left and right hand side.

The behaviour of the rule editor received almost universally positive responses. Users found the automatic creation of elements in the right hand side to be a helpful and intuitive feature. A minority of testers did express frustration that they were unable to layout interface nodes differently in each side, and also that they were unable to re-add a node to the interface having deleted it from one side, and instead had to create a brand new node in the left hand side.

6.2.1 Suggested Improvements

Several improvements were suggested by the feedback received from stakeholders. Unfortunately due to time constraints these will not be applied within the project period.

Feature discovery was a major problem for first time users of the application. There are two obvious ways for this to be improved. The first option would modify the interface and add a toolbar from which actions could be selected. Combined with expressive icons this would make actions almost instantly discoverable. The second option is simpler, and would keep the current interface, but add a help dialog or intro text for first time users. Thus retaining the advantageously fast editing provided by the current editor once the interface is learnt.

Label and ID inputs could be improved with the addition of an error tooltip explaining why the input failed to parse. Additionally, when a valid input is provided, deselecting the element could automatically apply the change, without the need to press the enter key.

Finally the rule editor could be improved with an arrow between the two editor panes, to correspond to the arrow commonly present in visualisations of rules in GP 2 literature. Alternatively, labels could be used to specify that one pane is the left hand side of the rule, and the other the right. This would allow first time users to more easily see how the rule editor relates to the rules they see in literature.

7 Conclusion

Over the course of the project two editors were developed, one for GP 2 host graphs, and one for GP 2 rules. Due to time constraints it was not possible to combine these two into a fully fledged development environment capable of writing and executing GP 2 programs.

The editor is robust, and no way in which it could be crashed was identified during the testing stage. This is a significant advancement from the previous tool, GP Developer.

Feedback on the produced editors was mixed, but suggested that the net contribution was positive. Testers found the interface for creating and visualising rules especially helpful.

Overall the result of the project is a good first step for improving the GP 2 development ecosystem, but further testing and development is required.

7.1 State of the editor

7.1.1 Featureset

The editor supports all of the features currently included in the main GP 2 distribution, such as marks, roots, and bidirectional edges. Host graphs and rules can be created and edited, and the editor is also capable of converting both of these to and from the GP 2 and DOT formats.

Files without positional information are automatically laid out when opened, and although node layout is performed manually within the editor, a primitive edge routing algorithm is implemented.

7.1.2 Limitations

No facility for the editing of GP 2 programs is provided in the current tool. As a result, it is not possible to use the editor for the entire workflow of the development of a GP 2 program.

Also, the editor is not suitable for use on a touchscreen. This effectively limits its usage to people with access to desktop computers and laptops.

7.2 Future work

As specific focus was placed on the design of the rule visualisation used by the tool, it would be useful to compare this to the visualisations provided by other graph transformation systems, and determine if the approach of maintaining the positions of preserved elements is helpful. Unfortunately, there was not enough time to perform such a comparison during the project period.

In order to experimentally test this, human test subjects could be presented with rules in several visualisations, and asked to identify corresponding elements from different sides. Response times and accuracy rate could then be used as a metric to determine how successful each visualisation is.

There are also some additions that could be made to the implementation itself. Primarily of course would be expanding the editor to support the editing and execution of GP 2 programs. This would require using a tool such as electron, to allow for interfacing with native operating system APIs to call the GP 2 compiler.

Once such work has been completed, the tool could be further expanded to allow for richer dynamic interaction, inspired by the likes of Scratch [11]. The addition of a visualiser for rule applications as is used in GrGen [5], or a state space explorer like that seen in GROOVE [8] would potentially help users more quickly build intuitions for the behaviour of the system.

A Tests

Test Case	Expected Result	Actual Result	Actual Result (GP Developer)
Set an invalid label in a graph	The program prevents the action, displaying an error	As expected	The label is set, and an invalid file is created
Set a node's ID to one that already exists in the same side of a rule	The program prevents the action, displaying an error	As expected	The ID is set, and continuing to interact with the program causes it to crash
Open a valid rule with layout information	The rule is displayed	As Expected	As Expected
Open a valid graph with layout information	The graph is displayed	As Expected	As Expected
Open a valid rule without layout information	The rule is laid out and then displayed	As Expected	As Expected
Open a valid graph without layout information	The graph is laid out and then displayed	As Expected	As Expected
Open an invalid rule	The program displays an error message	As Expected	A nameless, empty rule is created
Open an invalid graph	The program displays an error message	As Expected	A nameless, empty graph is created
Import a DOT rule	The rule is displayed	As Expected	N/A
Import a DOT graph containing attributes set on the graph as a whole	The graph is displayed	The file fails to parse	N/A

Bibliography

[1] D. Plump, ‘The design of GP 2’, in *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011.*, 2011, pp. 1–16. DOI: 10.4204/EPTCS.82.1.

[2] ——, ‘The graph programming language GP’, in *Algebraic Informatics, Third International Conference, CAI 2009, Thessaloniki, Greece, May 19-22, 2009, Proceedings*, 2009, pp. 99–122. DOI: 10.1007/978-3-642-03564-7_6.

[3] A. Elliot, ‘Towards an integrated development environment for GP 2’, Bachelor’s Thesis, University of York, 2013.

[4] H. Ehrig, C. Ermel, U. Golas and F. Hermann, *Graph and Model Transformation*, ser. Monographs in Theoretical Computer Science. Springer, 2015. DOI: 10.1007/978-3-662-47980-3.

[5] E. Jakumeit, S. Buchwald and M. Kroll, ‘GrGen.NET - the expressive, convenient and fast graph rewrite system’, *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 3–4, pp. 263–271, 2010. DOI: 10.1007/s10009-010-0148-8.

[6] M. Kroll and R. Geiß, ‘Developing graph transformations with gr-gen .net’, *Applications of Graph Transformation with Industrial releVancE-AGTIVE*, vol. 2007, 2007.

[7] J. Blomer, R. Geiß and E. Jakumeit, *The grgen .net user manual*, 2011. [Online]. Available: http://www.informatik.uni-bremen.de/agbkb/lehre/rbs/kurs/GrGen-Manual.pdf.

[8] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon and M. Zimakova, ‘Modelling and analysis using GROOVE’, *STTT*, vol. 14, no. 1, pp. 15–40, 2012. DOI: 10.1007/s10009-011-0186-x.

[9] U. Nickel, J. Niere and A. Zündorf, ‘The FUJABA environment’, in *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, 2000, pp. 742–745. DOI: 10.1145/337180.337620.

[10] C. Bak, ‘GP 2: Efficient implementation of a graph programming language’, PhD thesis, University of York, UK, 2015. [Online]. Available: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.684629.

[11] J. Maloney, M. Resnick, N. Rusk, B. Silverman and E. Eastmond, ‘The scratch programming language and environment’, *TOCE*, vol. 10, no. 4, 16:1–16:15, 2010. DOI: 10.1145/1868358.1868363.

[12] *The DOT language*, https://graphviz.gitlab.io/_pages/doc/info/lang.html, Accessed: 2019-02-14.

[13] G. D. Battista, P. Eades, R. Tamassia and I. G. Tollis, *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall PTR, 1998.

[14] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein and P. Mutzel, ‘The open graph drawing framework (OGDF).’, *Handbook of Graph Drawing and Visualization*, vol. 2011, pp. 543–569, 2013.

[15] J. Ellson, E. Gansner, L. Koutsofios, S. C. North and G. Woodhull, ‘Graphviz-open source graph drawing tools’, in *International Symposium on Graph Drawing*, Springer, 2001, pp. 483–484.

[16] C. Abras, D. Maloney-Krichmar, J. Preece *et al.*, ‘User-centered design’, *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*, vol. 37, no. 4, pp. 445–456, 2004. DOI: 10.1.1.94.381.

[17] E. Czaplicki, *Elm*, version 0.19, 2019. [Online]. Available: https://elm-lang.org/.

[18] elm-community, *Elm-graph*, version 6.0.0, 2018. [Online]. Available: https://package.elm-lang.org/packages/elm-community/graph/6.0.0/.

[19] C. Okasaki and A. Gill, ‘Fast mergeable integer maps’, in *Workshop on ML*, 1998, pp. 77–86.

[20] M. Brandly, *Elm-dot-lang*, version 1.1.2, 2019. [Online]. Available: https://package.elm-lang.org/packages/brandly/elm-dot-lang/1.1.2/.

[21] A. Zakai, ‘Emscripten: An llvm-to-javascript compiler’, in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ACM, 2011, pp. 301–312.

[22] M. Daines, *Viz.js*, version 2.1.2, 2018. [Online]. Available: http://viz-js.com/.