TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Mart Jõgi 142755IAPB

Diagrammatic Logic Editor

Bachelor's thesis

Supervisor: Paweł Maria Sobociński
MSc

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Mart Jõgi 142755IAPB

DIAGRAMMLOOGIKA REDAKTOR

bakalaureusetöö

Juhendaja: Paweł Maria Sobociński
MSc

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mart Jõgi

18.05.2021

Abstract

Peirce's existential graphs are a visual alternative to algebraic notation for logical and existential expressions. Currently there are no tools for programs to interface with existential graphs. This work attempts to implement two programs as a proof-of-concept. An editor for existential graphs, and an evaluator which takes a graph and can answer questions based on it. The editor is successfully implemented as a web app and included in this work, but an evaluator was unable to be created and is left merely as a suggestion.

This thesis is written in English and is 14 pages long, including 5 chapters, 13 figures and 1 table.

Annotatsioon

Diagrammloogika redaktor

Peirce'i olemasolugraafid on visuaalne alternatiiv algebrale predikaatloogika väljendamiseks. Hetkel ei leidu programme, millega oleks võimalik olemasolugraafe kasutada muuks, kui vaid nende vaatamiseks. Käesoleva töö eesmärk on välja arendada tarkvarasüsteem, millega Peirce'i olemasulugraafe (spetsiifiliselt Beta variante) luua ja kasutada. Süsteem koosneks kahest komponendist: diagrammredaktor, millega olemasolugraafe luua ja salvestada nende semantikat säilitavas formaadis, ning evaluaator, mis võtab salvestatud olemasolugraafi ning teostab selle põhjal mingisuguseid kasulikke arvutusi.

Diagrammredaktor sai implementeeritud veebirakendusena ning on antud tööle lisatud, evaluaatori loomiseks ei leitud head lahendust ning see jääb antud töö raames vaid soovituseks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 14 leheküljel, 5 peatükki, 13 joonist, 1 tabelit.

List of abbreviations and terms

CI	Continuous Integration - In gitlab specifically it is an automatic script that runs when code is pushed, commonly used to test, build, package, and deploy code.
EGIF	Existential Graph Interchange Format
ID	Identifier
(software) library	A prepackaged collection of reusable code, solving a specific issue or simplifying a specific workflow
px	Pixels
SQL	Structured Query Language
URL	Uniform Resource Locator
Cut node	Outlined/Shaded region on the graph, representing negation of the subgraph inside
Generator node	A subtype of predicate nodes. Rendered as a triangle and assumed to be true about one and only one identity
Predicate node	A text node that represents a statement about all connected lines of identity. Sometimes (including in this paper) drawn with a bounding box
Tap node	A circular node that acts as a connection point for drawing lines of identity, can be used to split/join or end a line.

Table of Contents

List of Figures

List of Tables

1 Introduction

In 1896, Charles Sanders Pierce invented a system of logic diagrams called "existential graphs"[1]. There are three variants, each building upon the previous: Alpha, which deals with propositions, Beta, which adds the notion of identity, and Gamma, which attempts to add second order logic but was never completed. These existential graphs offer a neat alternative to algebra for describing and manipulating logical and existential statements.

With the popularity of algebraic systems, this kind of diagrammatic reasoning has been left on the wayside. People who might find these graphs helpful are unlikely to ever hear of them, and if they do, will find that the only way to use them is to physically draw them on paper or to use very generic drawing/diagramming software which does not understand the semantics and will only output an image to look at.

The goal of this thesis is to create a visual editor specifically for the β variant of Peirce's existential graphs. This would have value as an educational aid, but even more importantly can serve to create more easily parseable output for other programs to do something with. (for example, a database might take an existential graph as input in order to run it as a query, or use it to define validations or schema)

2 Analysis

This section will outline the goals for the project, list existing software libraries that were considered to fulfill those goals, and justify the choice between them.

The goal is to have two separate components: an editor that can be used to create diagrams, and an evaluator that can take a diagram and do something useful with it, for example to evaluate queries on it.

The two components should not be strongly coupled - instead it should be possible to make lots of different evaluators that all take the same output from the editor and do something different to it. For example there could be logic evaluators that test statements, query evaluators that convert a diagram into a database query, and templating systems that use the diagram as a template.

The editor would be the core output of this work, while the evaluator would serve as a proof-of-concept example that other evaluators could be based on.

2.1 Diagram editor

The diagram editor is the primary component of this work, and should include (at the minimum) the following functionalities

- Adding the following elements to a diagram:
  - Lines of identity
  - Predicates
  - Cuts (outlined and/or shaded regions that may contain other elements)
- Connecting lines of identity to predicates
- Saving/Exporting the diagram in a format that maintains the semantic meaning of the diagram

Additionally, the following would be good to have

- Subgraphs (grouping/collapsing parts of the graph)

Looking for existing libraries that could be used for the editor, three main requirements were considered:

1. The library should be free to use (paid libraries might create licensing issues for this work)
2. The library should work at an appropriate level (low-level enough to allow the necessary customizations, but high-level enough to not create too much extra work)
3. The library should have enough documentation to be able to work with it (and to be able to evaluate the other requiremenets)

Additionally, to save time, it is preferred that the library is easily usable in a typescript+react project, since that is the stack the author is currently most familiar with.

Given those requirements, the following libraries were considered.

Table 1. Analysis of different javascript diagramming libraries

Syncfusion[1]	non-free
orgChart[2]	non-free
Kendo UI[3]	non-free
JSPlumb[4]	non-free
DHTMLX[5]	non-free
Rappid[6]	non-free
JointJS[6]	too low-level, and documentation is intermixed with Rappid
mxGraph[7]	deprecated

1 https://ej2.syncfusion.com/react/documentation/diagram/getting-started/
2 https://www.orgchartpro.com/
3 https://demos.telerik.com/kendo-ui/diagram/index
4 https://jsplumbtoolkit.com/
5 https://dhtmlx.com/
6 https://www.jointjs.com/
7 https://github.com/jgraph/mxgraph

Basic Primitives[8]	Too high-level, specific to organisational hierarchical charts
rete-js[1]	Bad support for react and typescript
React diagrams[2]	Bad documentation
GoJS[3]	Too low-level, possible licensing issues because free only for 'academic' use
litegraph[4]	Limited documentation - Customizing to fit our needs would be too low-level
D3[5]	Too low-level
diagram-js[6]	Appropriate level, but hard to find documentation
bpmn-js[7]	Possibly too high-level, hard to find documentation
react-flow[8]	documentation, scope both okay, only missing functionality is built-in nesting support

For the reasons outlined in the table, react-flow was chosen for the diagram editor.

2.2 Evaluator

The evaluator is the second component of the work, and needs the following functionalities

- Loading a diagram created in the diagram editor
- Evaluating queries based on the diagram, for example checking if one graph is locically implied by another

Looking for languages/tools that could be used to build the evaluator, the first to be considered was Prolog. Prolog is a logic programming language where the program

8 https://www.basicprimitives.com/
1 https://rete.js.org/#/
2 https://projectstorm.gitbook.io/react-diagrams/
3 https://gojs.net/
4 https://github.com/jagenjo/litegraph.js?files=1
5 https://d3js.org/
6 https://github.com/bpmn-io/diagram-js
7 https://bpmn.io/toolkit/bpmn-js/
8 https://reactflow.dev/

consists of only facts and rules, and then Prolog is able to check whether a given statement is provable.

Some initial attempts were made to express an example diagram in Prolog, but these attempts immediately ran into roadblocks both with encoding the diagram itself, as well as with defining negative statements (to convert the closed world of Prolog into an open world). Failing those, the search continued for other languages or tools that could be used.

Another keyword that stuck out was OWL (Web Ontology Language), which has the full expressive power of first order logic, but no clear examples or instructions could be found to determine if it is also able to solve statements or just express them.

Attempts were also made to look for what had been done with peirce graphs previously. Many papers, for example [2], reference specific formats such as EGIF that a peirce graph can be encoded into. But similar to OWL, clear examples of EFIG or programs that take advantage of it could not be found.

Due to time consraints, no other tools were able to be found.

3 Implementation

This section will outline the process of development, and the roadblocks encountered along the way.

3.1 Timeline

The 10 weeks (of 2 days each) available to work on this thesis were split up into the following stages: planning & research into peirce graphs (2 weeks), working on the editor (2 weeks), formatting the editor output (1 week), working on the evaluator (2 weeks), polish (1 week), writing the document (1 week), and 1 week spare.

In practice, everything went according to plan for the first few stages, up until it was time to work on the evaluator. During the evaluator weeks, attempts to use Prolog and searches for a different tool were all unsuccesful. By the time it was decided to leave the evaluator out of scope, full effort was already needed for the document, leaving the editor in it's relatively basic state as the only completed output of this project.

3.2 Terminology

Predicates in the diagram are represented by "predicate nodes", rendered as a rectangle with a solid outline, containing text. Cuts are represented by "cut nodes", which are rendered as a rounded rectangular area with a dashed border. Lines of identity are represented by the connections between different nodes.

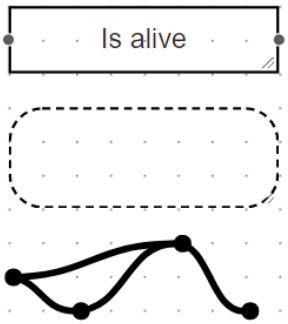

Figure 1: Diagram elements

To allow lines of identity to end without connecting to anything, "tap nodes" are also implemented. Tap nodes are to be interpreted as a part of the line of identity that makes up it and all of it's connections. Conceptually, the line of identity can be thought to only exist at the points where nodes are connected. This means that two connections that

cross each other without meeting at a tap node should be thought of as if they went past each other without touching. Likewise, if a connection goes into and out of a cut with no connection inside the cut, it should be thought of as if it actually went around. And if a line crosses multiple cut borders with no connections in between, it should be thought to "jump" straight over. Ideally the author of the graph would make sure such inconsistencies are ironed out by moving the nodes and connections appropriately and adding extra tap nodes where needed.

For example, the crossing in figure 2 should be understood as two separate lines of identity, one connecting A and B, and the other connecting C and D.

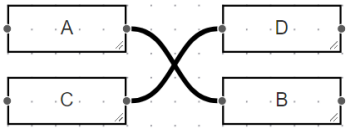

Figure 2: Crossing lines of identity

In this sense the diagrams in figure 2 and figure 4 are semantically equivalent. If the author intended for the lines to represent a single identity, they should add a tap node to connect the lines explicitly. Otherwise, while the diagram in figure 2 is valid, it is recommended to move things around so that there aren't any unattached crossings, as in figure 4.

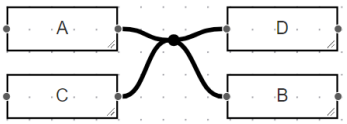

Figure 3: Explicit crossing using a tap node

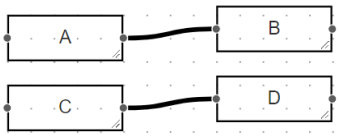

Figure 4: No crossing

Finally, taking inspiration from the paper "Compositional diagrammatic first-order logic."[4], "generator nodes" are semantically equivalent to predicate nodes, but represent a predicate which is true for exactly one identity. Generator nodes are rendered as rightwards-facing triangles, though unlike in the linked paper we will have them bulged out somewhat to allow for more space for the text

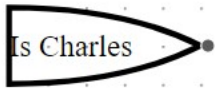

Figure 5: Example of generator node

3.3 Project setup

The core frameworks used for the editor were create-react-app and react-flow, chosen due to the author's familiarity with React and the availability of a suitable library for it.

Material-UI was used for any generic components like buttons and panels, and Ramda was used to simplify basic transformations.

Basic type checking was added by enabling and using typescript, but no unit or integration tests were planned for the editor. Since the design is relatively open and the amount of custom logic in the editor is very small, automatic tests were not considered worth it to implement for the editor. For any advanced functionality in the future, Jest would likely be used. Testing would also be critical for the evaluator, but the exact tool would depend on the framework/language that the evaluator would be written in.

For deployment, continuous integration was set up on gitlab. The CI script would take every push on to the "gl-pages" branch and automatically build and deploy it[1]. This serves as a basic test that everything compiles fine and it's not "just on my machine", and enables interested parties to see and test the most recent version of the code at any time, without having to download and compile the whole codebase themselves.

3.4 Roadblocks

Taking one of the official examples as a base, implementing the basic elements of lines, predicates, and cuts was rather straightforward.

The first difficulty was with the cut node, as it needed to hold other elements inside. Since react-flow does not (yet) have support for nested elements, it was instead implemented as a box that could be clicked through. React-flow also does not allow clicking through a node, so to bypass that, the node itself was made 1px in size, with the rest of the node overflowing the container. This allowed the overflowing part to be made click-through by using pointer-events: none.

To further the illusion of the cut node actually holding other elements inside, a feature was attempted that when a cut element is moved, it drags all overlapping elements along with it. This worked okay, but was visually confusing because the other elements only moved after the cut was dropped. Furthermore, sinze resize can only be done from the bottom right corner, adding more space to the left side of a cut required moving all

1 CI deployments were hosted at https://mart.jogi.pages.taltech.ee/iaib/, though that address is likely no longer available by the time this work is published

elements inside the cut individually. Finally, it was already possible to move a cut together with overlapping nodes simply by drawing a selection box around them before dragging. For these reasons, the drag-along feature was disabled.

Due to their triangular shape, generator nodes were not able to fit a lot of text without being massive themselves. To remedy this, a bulge was added to the shape to reduce the amount of wasted space outside the inscribed rectangle, the text was allowed to slightly break out of the container, and the use-fit-text library was used to automatically shrink the text as needed.

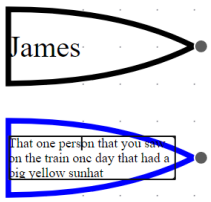

Figure 6: Generator nodes

4 Results

This section will describe the application that was implemented. The application is a react app that runs standalone in the browser. It consists of an empty canvas, a minimap, and a sidebar containing the different node types. The user can drag node types into the canvas, where the nodes can be edited and connected to each other. Nodes and connections on the canvas are able to be selected and deleted. The predicate and cut nodes are also able to be resized, and the cut node can be clicked through (when it is not selected).

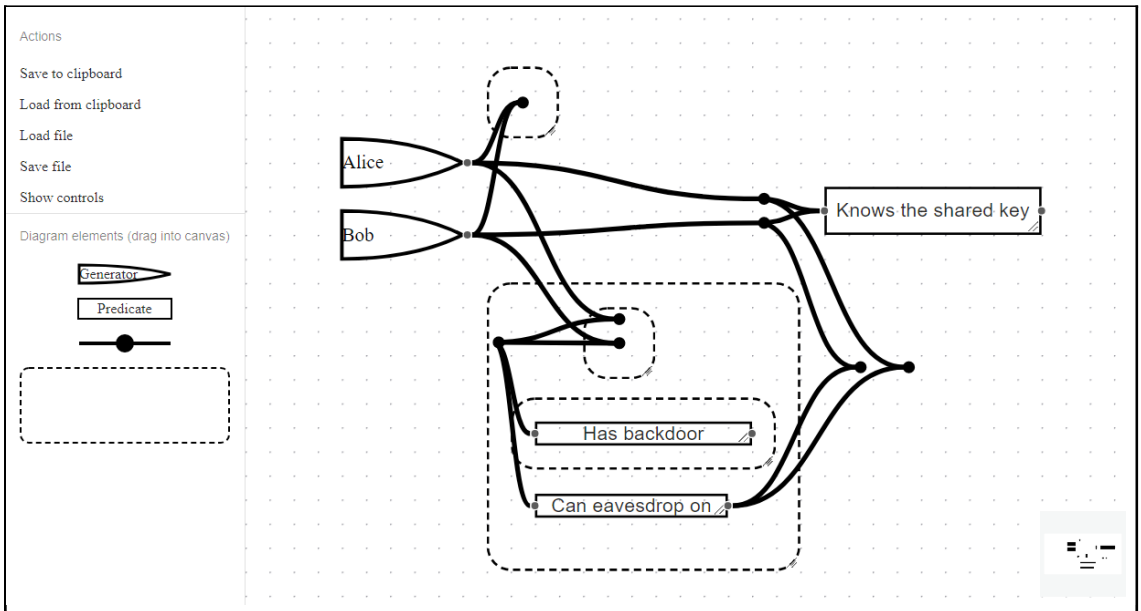

Figure 7: Full editor UI

This diagram is equivalent to the statements "Alice and Bob are different", "Alice and Bob know the shared key", "It is not true that there is some third person, who is neither Alice nor Bob, who is able to eavesdrop on Alice and Bob without having a backdoor"

The created diagram is automatically saved to session storage, to safeguard against losing work to accidental refreshes. The user also has the option to save manually either by copying the diagram to the clipboard (making it easy to paste into another program or share in instant messaging applications), or by downloading it as a file.

The biggest limitations right now are that predicates with arity > 2 need to be split up into binary/unary predicates, and that lines of identity that cross a lot of cuts can be awkward to draw due to the requirement to insert tap nodes at every level.

Figure 8: Example from [3]

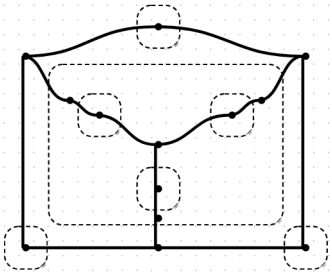

Figure 9: Figure 8 implemented in the editor

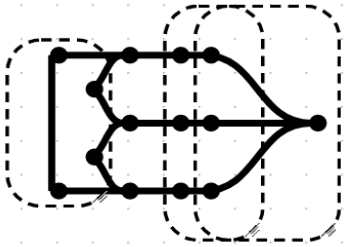

Figure 10: A compact equivalent alternative to figure 8

Aside from those cases, a lot of graphs can be implemented directly and look very similar.

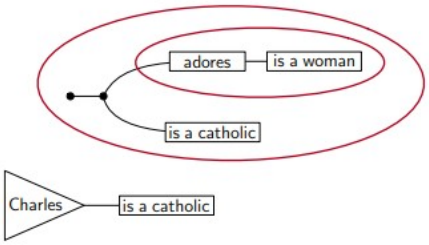

Figure 11: Example from [4]

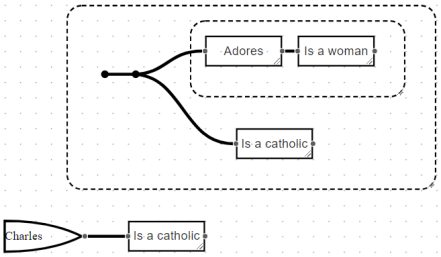

Figure 12: Figure 11 implemented in the editor

5 Further development

This section will cover ideas that were not implemented due to time constraints, but that would be useful to have.

5.1 In-editor proof mode

If a good engine can not be found to make an automatic evaluator, an alternative could be to have a "proof" mode in the editor itself. When in the proof mode, the user would be limited to only making changes to the diagram that are semantically valid.

A proof mode would not be as convenient if the user just wants an answer, but would be excellent as a teaching/learning aid. Proof mode could save as a list of actions, allowing users to share and review the whole proof, as well as to undo/redo during the proof, which is important because some transformations are only valid in one direction (for example, outside a cut it is always valid to cut a line of identity into multiple, but it is not allowed to join two separate lines of identity together)

5.2 Predicate arity

Some predicates can have different meanings depending on where a line is connected. for example a line of identity can connect to the predicate "Is the parent of" as either the parent or the child. Currently, the editor only distinguishes between connecting on the left and on the right. It would be more expressive if each node could have an arbitrary number of different labelled connection points, so that it would be unambiguous which line of identity corresponds to which part of the relation.

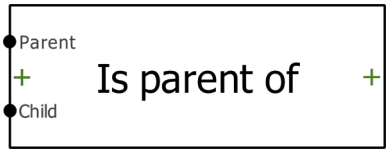

Figure 13: Mockup of arbitrary-arity predicate

5.3 Saving diagrams online

Similar to other platforms like Google Docs, it would be convenient to be able to save diagrams in the app itself.

One solution to do this would be to use localStorage. While that would be device-specific and diagrams would be lost if the storage is cleared, it would enable users to save diagrams without having to register accounts or rely on a server to be available to host. It is also the simplest to implement.

Alternatively, to allow diagrams to persist for a long time and across devices, the editor itself could be registered as a Google Drive app, and be made able to store the created diagrams in the user's own drive. This would allow the user to save diagrams online without even having to register for another account just for this app, and would still bypass the need to create and host our own backend services.

5.4 Sharing diagrams

In order to avoid having to send/upload files and instruct users to load them, it would be convenient to share a diagram just by sending a URL.

If there is going to be a backend to store the user's diagrams, then it's simply a matter of making sure every diagram has a unique ID, adding a "public" flag that the user can toggle, and giving the user the option to copy a direct link that contains this ID.

If the editor is registered as a Google Drive app, or a similar system is implemented that allows storing diagrams on a different platform, then that platform's native file sharing would be able to be used.

Otherwise, the diagram could also be encoded into the URL itself, similar to how PlantText does it. Such links would have the benefit of working fully clientside, with no need to maintain servers to store the diagrams, and the same link would always produce the same diagram rather than referencing a file that the owner might edit later.

5.5 Visual consistency

The current implementation allows the user to draw a line of identity through a cut, without having a tap node inside. Such cases are not obvious from the diagram data, so the semantics are that if there's no node inside the cut, we interpret that diagram as if the line didn't cross the cut at all.

A better solution would be to detect whenever a line of identity crosses a cut, and if it does to automatically create a tap node inside to make the crossing explicit. If needed, then this detection could be made simler by removing the curviness from lines and cut borders. To avoid the automatic nodes from becoming frustrating, it should also be made easier to delete a tap node from the middle of a line.

One option would be that when a tap node is deleted, all connected tap nodes will be connected to each other. That is a bit ambiguous though, since there are many ways to connect the remaining nodes if there's more than two of them.

Another option is to have deleting the node cut the line as normal, but also allow dragging it over another tap node, which would delete it while connecting all it's other connections to the node that it was dragged over.

5.6 Copy and paste

For many diagrams, it would be convenient to be able to copy and paste parts of the diagram to reuse. This should be relatively simple to implement by detecting the copy and paste shortcuts. On copy, the "selected" elements, which are already tracked, would be copied into the clipboard. On paste, elements in the clipboard would be pasted onto the diagram, their positions would be slightly shifted to make it visually obvious, and the pasted elements would be selected (deselecting any elements that were already selected)

References

[1] Roberts, Don D. "The existential graphs." Computers & Mathematics with Applications 23.6-9 (1992): 639-663.

[2] Sowa, John F. "Existential Graphs and EGIF."

[3] Sowa, John F. "Peirce's tutorial on existential graphs." (2011): 347-394.

[4] Haydon, Nathan, and Paweł Sobociński. "Compositional diagrammatic first-order logic." International Conference on Theory and Application of Diagrams. Springer, Cham, 2020.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Mart Jõgi

1 Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Diagrammatic Logic Editor", supervised by Paweł Maria Sobociński

1.1 to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;

1.2 to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.

2 I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.

3 I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

18.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.