ELVE: An Interactive and Extensible Visualisation Tool for Logic Circuits

Grégoire Hirt, Ana Petkovska, and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
{gregoire.hirt, ana.petkovska, paolo.ienne}@epfl.ch

***Abstract*—We present a novel interactive open source tool for visualisation of logic circuits. The current state of the tool offers basic features for visualising circuits and provides a mechanism for easy extension through plug-ins in order to fit the user's needs. The interactive user interface allows the user to change how the circuit is visualised, as well as to compress, highlight or extract some logic for analysing the circuit. The proposed tool can be used to better understand the effects of the developed algorithms in research projects, for educational purposes, as well as for generating figures for technical documents.**

I. Introduction

It is well-known that the human brain processes visual information much faster and easier than text. This fact is already exploited by most *electronic design automation (EDA)* tools that can represent visually logic circuits for several reasons. First, visualising a circuit design enables understanding its structure, identifying structural patterns, comparing it to another design or circuit, etc. This is particularly helpful when developing algorithms that operate on logic circuits, such as algorithms for logic synthesis and mapping, because we can learn how the algorithms operate by visualising the affected regions and changes made on the circuit structure. In the same manner, such a tool can be efficiently used by educators and students. Finally, a good visualisation of circuits is often required to produce figures for technical documents, such as publications and technical reports.

For example, ABC [1], which is a widely-used academic software system for synthesis and verification of logic circuits, represents the circuit internally as a *directed acyclic graph (DAG)*. Then, it creates a textual graph description for the DAG and renders it in a graphical form. However, as Figure 1 shows, even for some small circuits these static images are unreadable due to the large number of entangled edges. In such cases, some interactive options, such as clustering and highlighting nodes, are required in order to fully exploit the power of visualisation.

On the other hand, existing tools dedicated to circuit visualisation are mainly developed for pedagogical use [2–6], so they are either bound to a specific *hardware description language (HDL)*, are restrained to drawing and editing circuits schemes, or are not publicly available.

The biggest drawback of all existing options is that they have limited capabilities and are not easily extensible, while in academia, we typically need a representation and visualisation of the circuit that is specific to the on-going research.

Thus, in this paper we present an interactive visualisation tool for representing the design and structure of logic circuits generated by different EDA tools. We can not predict nor implement all visualisation options and input/output formats that are required by different tools and projects, but instead we provide an initial open source implementation with basic options that the users can further extend according to their needs through dedicated plug-ins. We would like to encourage sharing of plug-ins between users, and thus, in the future, we plan to provide a web based system that would ease this process. Currently, the proposed tool and plug-ins have good performance for logic circuits of medium size with thousands of nodes, but a more sophisticated clustering algorithm is required for fast processing and practical visualisation of large circuits. Also, although we are currently focused on circuits resulting from logic synthesis and mapping algorithms, the tool can be easily extended to support designs from different stages of the EDA flow since all designs can be easily represented as a graph.

Following, in Section II, we describe our tool for visualisation of logic circuits and evaluate its performance. In Section III, we describe the related work. Finally, Section IV concludes the paper.

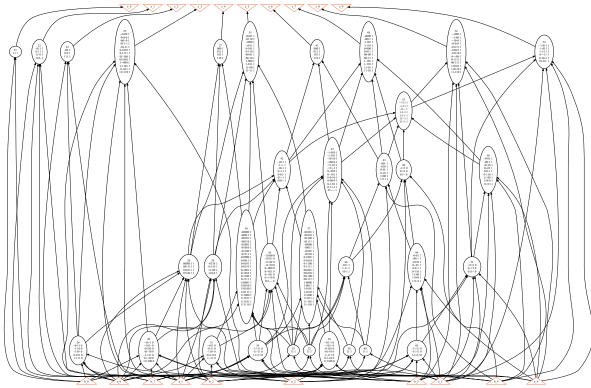
II. Interactive and Extensible Visualisation Tool for Logic Circuits

The proposed tool, which is called *ELVE Logic Visualisation Explorer (ELVE)* [1], is designed as an extensible and interactive graph viewer and manipulator, and is adapted for visualisation of logic circuits. It is developed as a multi-platform desktop application that consists of a core module with many primitives and features that allow basic manipulation and visualisation of graphs. Furthermore, the core module enables linking of plug-ins at runtime with which the user can extend both the graph processing pipeline and the GUI of the application. Figure 2 illustrates its software structure. In the following sections we describe our initial implementation of ELVE in details.

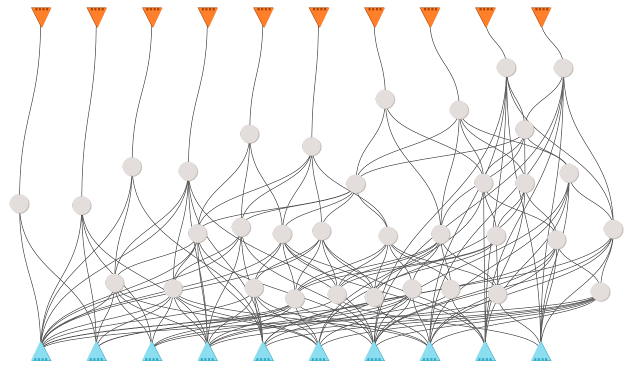
A. Qt as Application Framework

ELVE is implemented using *Qt* [7] that is a well-known open-source application framework. Qt is cross-platform and provides a *graphical user interface (GUI)* framework. Qt uses the C++ programming language, and thus it is articulated in a

[1] https://github.com/stdgregwar/elve



(a) A visualisation generated by ABC.



(b) A visualisation generated by ELVE.

Figure 1: Visualisation of a 5-bit multiplier mapped into 6-input lookup tables (LUTs). The graph consists of 44 nodes and 167 edges that are spread over 5 levels. Statically, the two visualisation are very similar and the connections in the lower levels are hard to follow. However, ELVE enables interacting with the graph and moving nodes that can help identifying easily their inputs and outputs. Moreover, we can dynamically highlight paths and nodes in the graph that eases the analysis of such logic circuits. The information shown in the nodes of ABC is now shown as a tooltip when the user hovers the pointer over a node.

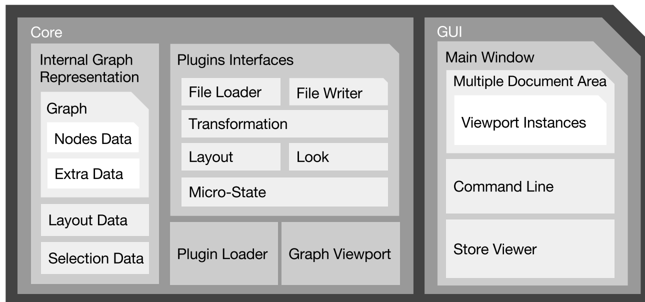


Figure 2: ELVE's software structure.

object-oriented manner by following a *model-view-controller (MVC)* pattern, which separates the internal representation of information from the user interaction. For ELVE, Qt is used as a widget toolkit that provides many predefined libraries of widgets (i.e., graphical control elements) that ease the construction of desktop based applications with a GUI.

For ELVE, Qt displays the main window and, most importantly, the *viewport* that represents the area in which the graph is drawn. Qt directly allows having interactive items that can be clicked and manipulated. For ELVE, these items are the nodes and edges of the graph representing the logic circuit.

B. The Core Module

The core module is the heart of ELVE. It defines the internal graph representation, provides the interfaces for the plug-ins, allows loading of plug-ins, and gives the graph viewport. In this section, we describe the data and systems comprising it.

1) Internal Graph Representation: ELVE's internal graph representation is an *immutable data structure* meaning that the graph data is never changed, but if we want to make some changes into the graph's structure (like clustering some nodes), we need to create a new graph. This allows to revert the graph to a previous state at any moment. The internal graph representation consists of the following parts.

- `Nodes Data` is a tabular structure that keeps the information for all nodes. The ID of each node represents its index in the table. For each node, we also keep the name, ancestor nodes, and type (which can be input, output, node, cluster, input cluster, or output cluster). Additionally, when extending the core module with plug-ins, any kind of data can be transparently attached using *JavaScript Object Notation (JSON)* fields, allowing tree-like data constructions to be stored in each node.
- `Extra Data` is a supplementary node information stored in a sparse-table. It stores extra nodes created by ELVE or its plug-ins. For example, a *cluster* is a newly created node that replaces several nodes in the graph, and thus it contains the IDs of its underlying nodes.
- `Layout Data` represents the information for the layout system described in Section II-B2. It contains the position of all nodes, as well as the layout constraints and looks.
- `Selection Data` contains the selections masks of the selection system described in Section II-B3.

Since each of these parts is also immutable, when we create a new graph, the ones that remain the same can be reused in order to preserve memory.

For `Node Data` and `Extra Data`, ELVE constructs a pointer based representation that allows to traverse the graph by following edges, and thus, any graph algorithm can be applied quickly and efficiently. To control how this representation is constructed, any plug-in of the type `Transform` and `File Loader`, which are described in Section II-B5, can provide an `alias` table and an `excluded` table. The `alias` table maps nodes through their IDs (from one ID to another), allowing to route edges to other nodes while the `excluded` table defines which nodes should be excluded from the visualisation. The combination of the two allows for various constructions of

the graph, from clustering nodes to extracting only a subset of them. Since the pointer based representation of the graph is created from these tables only when the graph is created, it does not impact traversal or visualisation performance.

The internal graph representation can be serialized as a *JSON* file or it can be saved in a binary format. This allows to save the state of the visualisation at any time and resume working with it later.

2) Graph Layout System: The key feature of the tool is drawing the graph with well-spread layout that minimizes the number of elements that overlap. By default, ELVE uses the simple force-directed Eades [8] layout algorithm. It is implemented as a small and multi-threaded physical engine, and treats the graph as a point system in which each edge represents a spring and all nodes are small positives charges that repel each other. A physical simulation then naturally reduces entropy and leaves the graph in a homogeneous spatial distribution.

The core module exposes interfaces to the physical system and to the implementations of forces. Thus, additional plug-ins can be written to enable new layouts that can either extend the existing physical force system or fully evict it if a different approach is required.

3) Selection System: A *selection mask* is a set of nodes represented through their IDs that can be used both as an input to or as an output of ELVE's commands. Each graph in ELVE has 10 different selection masks that allow having 10 different selections of nodes. The selection masks are visually represented by changing the color of the selected nodes and edges, and only one selection mask can be active and shown at any moment. For example, the command `group`, which allows manual clustering of nodes, takes the active selection mask and creates a cluster node that replaces the selected nodes.

The motivation for the selection masks is that they can display result of the algorithms and can control their behavior. For example, to open the fan-in cone of a node as a subgraph, we should select the node, run the `Fan select` plug-in that saves the fan-in cone as a selection mask, and then we can open the selection as a new graph. In this case, the selection mask acts both as a convenient way to store the data and to visually check the result. Basic set operations such as union and intersection can also be applied to the selection masks to obtain a new selection.

4) Data Pipeline: ELVE's core module is designed as a data pipeline, which is shown on Figure 3. The data pipeline receives the information for the logic circuit by reading a file, and finally provides an interactive visual representation of the graph. Each stage of the pipeline, except stage 2, is bound to one of the plug-in types described in Section II-B5. Each stage can be invoked both from the GUI and from the *command line interface (CLI).*

Once the file is read (stage 1) the graph is in ELVE's `store`, which is explained in Section II-C3, and one can interact with it trough the CLI loop (stages 2 and 3). This allows to transform the graph before showing it. Then, the user can choose to show the graph (stage 2) in ELVE's viewport by executing the GUI loop (stages 4 to 9). The primary layout stages 4 and 5 generate the layout system and run the layout algorithm for the first time before displaying the graph in stage 6. From this point it stays in a loop between the stages 8, 6 and 9, when the user can interact with the currently displayed graph. If in meanwhile a `Transform` plug-in is triggered, then it goes trough the stages 7, 4, and 6, but does not pre-simulate again. This is required for preserving the current layout of the graph as much as possible even when applying transformations. These shortcuts also allow the user to see the changes made to the graph in real-time.

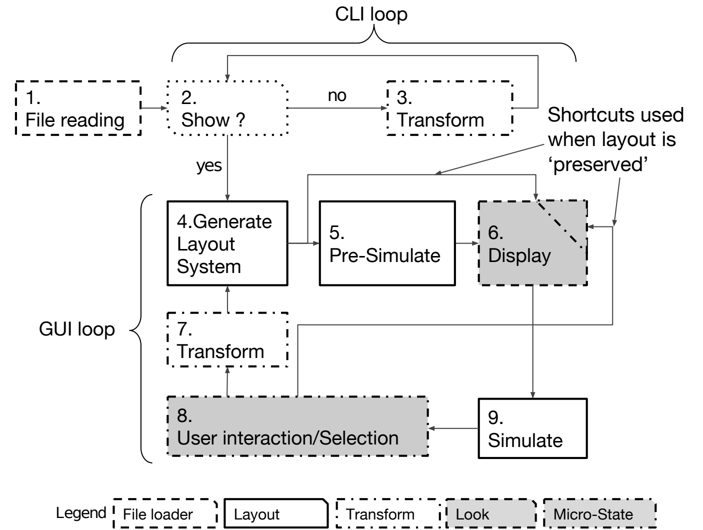


Figure 3: ELVE data pipeline. Any part of the pipeline can be replaced because they are implemented with plug-ins. The `File writer` plug-in is not present since it can be invoked at any stage.

5) Extension with Plug-ins: ELVE exposes interfaces for some of its modules and allows linking plug-ins that implement these interfaces at runtime. Each part of the data pipeline can be extended or replaced using a plug-in. The core module provides interfaces for the following six types of plug-ins.

- The `File Loader` interface allows to read a circuit description from a file and to initialise the internal graph representation. Each input file format can be supported by a different plug-in.
- The `File Writer` interface allows to read the internal graph representation and write either the circuit description into a file, or output the visualisation in a file for further use.
- The `Layout` interface allows to implement different layout systems that define the position of the nodes and create different visualisations of the graph.
- The `Look` interface allows to define how nodes and edges are drawn based on the graph data and the output of the layout stage.
- The `Transform` interface allows to implement algorithms, such as the clustering algorithms, that receive a

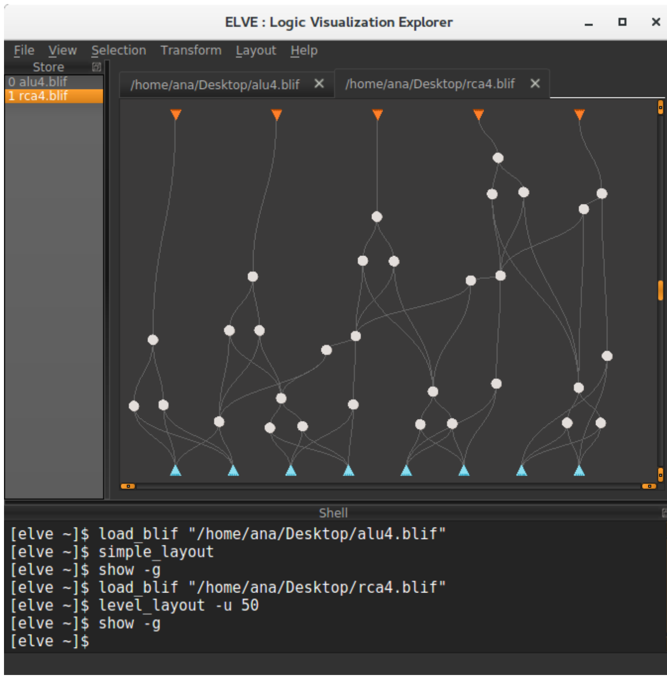


Figure 4: ELVE's GUI when two circuits (alu4 and rca4) are read into ELVE. Both circuits are available through the store menu on the left. In the viewport, the tab for the circuit rca4 is currently active and showing its implementation. We can see the executed commands in the shell shown at the bottom.

graph and output a new graph with a different set of nodes and edges.

- The `Micro-State` interface allows to redefine the behavior of the viewport in order to extend the user interaction or to add overlays to the drawing.

6) Plug-in Loader: The plugin loader is a helper module that automatically loads any plug-in placed in ELVE's dedicated folders. For each available plug-in, it tests if the plug-in is of the right type and if it is linked with the appropriate version of ELVE. This way any unsuitable plug-in would be reported and disabled.

7) Graph Viewport: The graph viewport is where the visualisation and direct interaction happen. It is a canvas where the graph is drawn and allows to interact with nodes by moving or selecting them. Any node information (as ID, level, type, etc.) can be shown as a tooltip when the user hovers the pointer over a node. The visualisation can be zoomed and dragged at any time to better inspect the graph. View helpers are available to reset the view position and zoom level, as well as to look at particular points.

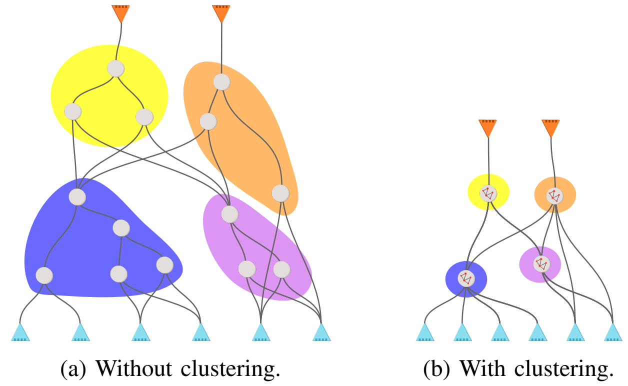
C. Graphical User Interface

The *graphical user interface (GUI)* of ELVE allows visualising circuits, but also interacting with them either through its menus or though its command line interface. Figure 4 shows the GUI when two circuits are read into ELVE. Each plug-in can extend the existing GUI by adding slots to the menu trees, assigning shortcut keys for its actions, and providing forms for custom control of the internal algorithms.

1) Viewport Instances: ELVE uses Qt to display a classical *multiple document interface (MDI)* GUI that allows viewing and transforming several logic circuits at the same time using tabbed views containing viewports. It also enables opening the result of some transformation in a new tab or showing separately sub-graphs of the original graph.

2) Command Line Interface: Additionally, we embedded the EDA *command line interface (CLI)* framework called *Alice*, which is provided as part of the tool *CirKit* [9]. The CLI can be used as an alternative to the GUI's menus because each available action in the menus is associated with a command. Actually, when we select an action from a menu, we first issue a command that triggers the action. With this, the history of all executed actions is available in the CLI as commands, and it can be used to obtain the same visualisation for a different logic circuit by invoking the exact same actions. Furthermore, the command history for a circuit is saved and attached to the JSON files that are used to save the internal graph representation.

3) Store Viewer: *Alice* CLI framework saves current data into a `store`. This allow to have multiple circuits loaded at once, but only the one on which we are working on is active. The `Store Viewer` is a panel in the GUI that allows the user to view which graphs are loaded in the `store`, to select which one is the active graph and to show the graph.



(a) Without clustering. (b) With clustering.

Figure 5: An example that illustrates the four clusters formed by the clustering algorithm implemented with the `Cluster` plug-in.

D. Implemented plug-ins

To demonstrate the extension with plug-ins and how they can be developed, we provide the following six plug-ins for start, but we plan to extend this set in the future.

- The `BLIF loader` plug-in implements a `File Loader` interface and enables visualising a circuit described using a *Berkley logic interchange format (BLIF)* file. It reads the file and translates it to the internal graph representation.
- The `SVG exporter` plug-in implements a `File Writer` interface and exports the current visualisation

Table I: Basic information and ticks per second achieved by ELVE after using the `Cluster` plug-in.

Benchmark	PIs	POs	AND nodes	Levels	Ticks per second	
					$\Theta = 2$	$\Theta = 50$
Adder	256	129	1020	255	1393.0	1898.3
Barrel shifter	135	128	3336	12	895.1	1491.6
Divisor	128	128	57247	4372	13.8	46.1
Hypotenuse	256	128	214335	24801	0.0	11.3
Log2	32	32	32060	444	37.1	121.2
Max	512	130	2865	287	829.9	1283.2
Multiplier	128	128	27062	274	73.9	150.6
Sine	24	25	5416	225	490.3	790.5
Square-root	128	64	24618	5058	90.1	161.1
Square	64	128	18484	250	33.8	150.3
Round-robin arbiter	256	129	11839	87	603.9	869.9
Alu control unit	7	26	174	10	3992.5	4157.0
Coding- cavlc	10	11	693	16	3087.9	3667.1
Decoder	8	256	304	3	1654.4	2444.5
i2c controller	147	142	1342	20	1436.9	2270.8
Int to float converter	11	7	260	16	4261.1	4498.8
Memory controller	1204	1231	46836	114	47.5	97.4
Priority encoder	128	8	978	250	2262.0	2904.6
Lookahead xy router	60	30	257	54	3156.2	3912.6
Voter	1001	1	13758	70	152.6	224.6

as a *Scalable Vector Graphics (SVG)* file. This can be particularly useful for producing technical documents.

- The `Simple layout` plug-in implements the Eades layout algorithm [8] through the `Layout` interface. It takes the internal graph representation as input and produces a simulable physical system that is displayed in the viewport.
- The `Level layout` plug-in differs from the `Simple layout` by adding vertical constraints on nodes to show their level in the graph.
- The `Fan select` plug-in implements the `Transform` interface and alters the current selection mask by selecting the fan-in or fan-out cone of the preceding selection.
- The `Cluster` plug-in implements the `Transform` interface and applies a simple clustering algorithm on the whole graph. The implemented algorithm traverses the nodes from the outputs to the inputs and forms non-overlapping clusters. The node on the highest level in each cluster is consider as a root node, and the cluster includes a cut of the root node in which all nodes, except the root node, have a single fanout.

E. Performance

ELVE already exhibits good raw performance for logic circuits of medium size that are composed of thousands of nodes. Since we aim for interactivity, we analysed the simulation speed of the layout algorithm that is measured in *ticks per seconds (TPS)*, where *ticks* are simulation steps. For a smooth interaction, this metric should be above 30.

For evaluating the performance we used the the arithmetic and control benchmarks from the the EPFL Combinational Benchmark Suite [10], and their basic information are given in Table I. The benchmarks are run by loading the BLIF files

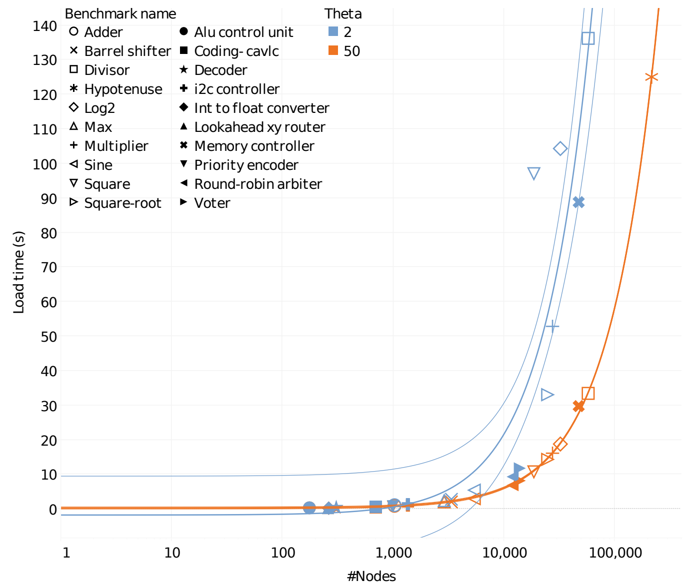


Figure 6: Dependency between the load time and the number of nodes. The number of nodes are shown on logarithmic scale. The thick lines present the trend lines for the given Θ parameter, while the thin lines present the confidence interval. The only omitted result is for the Hypotenuse benchmark when $\Theta = 2$ because it required 16.7 hours to finish, while for $\Theta = 50$ it finished in 2.1 minutes.

and simulating the `Level layout` for 400 ticks, which in most cases is sufficient for the layout to stabilize. For the experiments, we changed the parameter Θ that defines the accuracy of the Barnes-Hut [11] repulsive forces of the layout algorithm implemented with the `Level layout` plug-in. We used $\Theta = 2$ (high accuracy) and $\Theta = 50$ (low accuracy). The reported results for the load time and TPS represent the average over 3 runs to avoid runtime noise. All benchmarks are executed on an Intel i7-4790 CPU clocked at 3.60GHz, with 8 threads and 8MB of cache.

As Figure 6 shows, the loading time is linear with the number of nodes in the benchmark. But, structural properties, such as number of levels and interconnection of nodes, also matter for an accurate layout, so for $\Theta = 2$ all benchmarks do not follow strictly the trend line.

Since for benchmarks with more than 10000 nodes, the load time is too long and they require more than 30 ticks per second, we suggest to run a clustering algorithm as a preprocessing step, before running the layout algorithm, that would simplify the graph by decreasing the number of nodes. As shown on Figure 7, by running the simple clustering algorithm implemented with the `Cluster` plug-in, we decrease the number of nodes for 65% and the loading time for 58%, on average over all benchmarks. With this, as Table I shows, for most benchmarks we achieve good performance in TPS when $\Theta = 2$, but for big benchmarks we can trade the accuracy of the layout algorithm to improve performance.

However, just few of these circuits are truly explorable

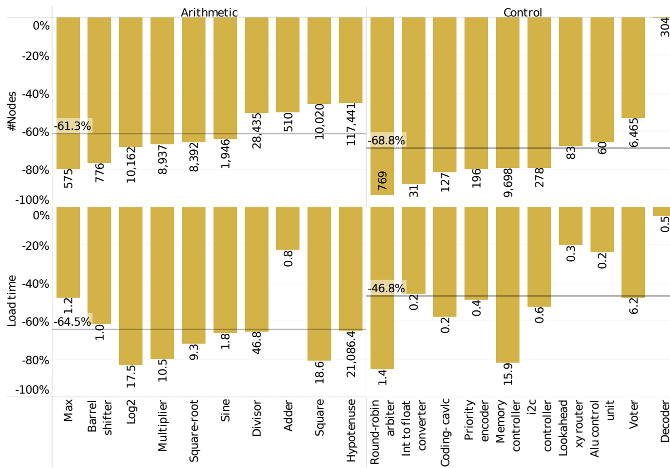


Figure 7: Reduction of the number of nodes and load time after executing the available clustering algorithm. The lines over all bars give the average for the corresponding set. The actual number of nodes and load time (in seconds) is shown next to the bars.

in ELVE due to the large number of nodes that prevent distinguishing nodes and connections on the screen. But, with a more sophisticated clustering algorithm, and by expanding and highlighting just the nodes of interest, we would be able to explore and analyse these circuits.

III. Related Work

The need for a circuit visualisation tool is recognised also by other researchers that either introduced visualisation options in their EDA tools or built special tools for visualisation.

Almost all tools, both commercial and academic, support visualisation of circuits. The problem is that often their options are quite limited, as the circuit visualisation is not their prime focus. Also, one can hardly customize these tools to suit its needs: the code of the commercial tools is typically not available; while for the academic tools, there are no platforms where these extensions can be easily shared. Moreover, the visualisation engine of each tool is very dependent on its internal structure, so it is hard to achieve some portability between different tools. Finally, the academic tools usually produce static images that do not allow any interaction.

The other option is to use tools whose main focus is visualisation of circuits, but these are mainly built for pedagogical use. For example, Shoufan et al. presented the interactive platform VISUAL-VHDL [2] for visualizing and simulating digital circuit written in VHDL. Later, Shoufan et al. presented the web based platform for visualization and animation of digital logic designs, called DLD-VISU [4]. Stanisavljevic et al. presented a system for digital logic design and simulation (SDLDS) [5] that consists of modules for design, simulation and evaluation. Hacker et al. presented a set of Window-based tools to teach elementary circuit design (WinLogiLab) [6]. Poplawski et al. wrote a set of simple Java applets [3].

However, since these tools are mainly built for educational purposes, the size of the designs that they can support is very limited, especially because among their main options is using Karnaugh maps and the Quine-McCluskey algorithm for optimisation. They aim more at presenting the algorithms and give the visualised circuits as examples or support. They are not targeting graph drawing and interaction efficiency, but are more focused on providing editors for the circuits. Finally, most of them are not publicly available.

IV. Conclusion

With this paper we propose an initial implementation of an interactive and extensible visualisation tool for logic circuits, called ELVE. Our main goal is to provide a rich but flexible tool that should mainly ease the development of algorithms and the production of figures for technical documents, but can also be used for educational purposes. ELVE is an open source tool and its source code is freely available for download.

In the future, we plan to implement more plug-ins in order to provide wider set of basic functions required for general use. We will also update the structures and algorithms to supports sequential circuits that imply having data loops in the graph. Finally, in order to encourage the sharing of plug-ins between users, we plan to create a web based system where users can upload their plug-ins and retrieve plug-ins developed by others. On the long term, we plan to provide a simpler way to write plug-ins using a script language to avoid compiling and compatibility issues.

Acknowledgments. We thank Mathias Soeken for providing us with the EDA CLI framework called Alice, as well as for the useful comments and ideas. We also thank Grace Zgheib and Andrew Becker for useful discussions.

References

[1] "ABC: A system for sequential synthesis and verification," Berkeley Logic Synthesis and Verification Group, Berkeley, Calif., http://www.eecs.berkeley.edu/~alanmi/abc/.

[2] A. Shoufan, Z. Lu, and G. Rößling, "A platform for visualizing digital circuit synthesis with VHDL," in *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education*, 2010, pp. 294–98.

[3] D. A. Poplawski and Z. Kurmas, "JLS: a pedagogically targeted logic design and simulation tool," in *ITiCSE08*, 2008, p. 314.

[4] A. Shoufan, Z. Lu, and S. A. Huss, "A web-based visualization and animation platform for digital logic design," *IEEE Transactions on Learning Technologies*, vol. 8, no. 2, pp. 225–39, Apr. 2015.

[5] Z. Stanisavljevic, V. Pavlovic, B. Nikolic, and J. Djordjevic, "SDLDS—system for digital logic design and simulation," *IEEE Transactions on Education*, vol. 56, no. 2, pp. 235–45, May 2013.

[6] C. Hacker and R. Sitte, "Interactive teaching of elementary digital logic design with WinLogiLab," *IEEE Transactions on Education*, vol. 47, no. 2, pp. 196–203, May 2004.

[7] "Qt," https://www.qt.io/.

[8] P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, no. 11, pp. 149–60, 1984.

[9] "Cirkit: A circuit toolkit," https://github.com/msoeken/cirkit.

[10] "The EPFL Combinational Benchmark Suite," http://lsi.epfl.ch/benchmarks, Integrated Systems Laboratory (LSI), EPFL, Lausanne, Switzerland.

[11] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, pp. 446–49, Dec. 1986.