DrawSE2: an application for the visual definition of visual languages using the local context-based visual language specification

Gennaro Costagliola, Mattia De Rosa, Vittorio Fuccella, Vincenzo Raia
Dipartimento di Informatica, University of Salerno
Via Giovanni Paolo II, 84084 Fisciano (SA), Italy
{gencos, matderosa, vfuccella}@unisa.it

Abstract

We present DrawSE2, a new web application that allows the definition of visual languages in a more visual way (according to the local context-based visual language specification). The tool allows the user to create visual language elements, define their attaching areas (the hotspots through which language elements can be connected), and define how they can be linked together to form admissible language sentences. The tool also allows semantic attributes to be defined and enables semantic translation (e.g., to a textual representation). The visual language thus defined can then be used in a diagram editor that allows to draw visual sentences of the language, check their correctness and get their semantic translation.

Keywords: *visual languages, local contex, webapp.*

1. Introduction

Visual languages have been used as part of systems that use visual representations to facilitate communication. Visual sentences include diagrams, maps, images, and pictures, which are used to communicate mental concepts that require spatial settings to be described appropriately. Their purpose is to make it easier for people to communicate, since, when done correctly, visual communication is more direct and instantaneous than spoken or text communication.

This is why visual languages can be found in a variety of contexts, from art to engineering. However, if they are badly designed, they can be difficult to interpret and compose, defeating their purpose. This, for example, may occur when a language has many syntactic rules that bind elements that can be far apart in a sentence. For example, in textual programming languages, one might consider the matching parenthesis in languages such as C or Java.

DOI reference number: 10.18293/DMSVIVA22-014

One way to overcome the problem of syntax dependencies between far language elements is to add shape information to each element, like in the Scratch block visual language. This reduces the composition of a visual sentence (a program in this case) to the creation of a puzzle, with a very simple syntactic rule: "a visual program is syntactically correct if and only if each block (tile) well interlocks with its neighbors". The local shape constraints on each block in this case guarantee the correctness of the whole visual program, regardless of how many elements it is made of.

This is also why block languages are now very popular for teaching introductory programming to non-experts, as well as for prototyping and scripting purposes [23].

We have previously shown [5–10] that many well-known and widely used visual languages (such as unstructured flowcharts, data flow languages, and entity-relationship diagrams) can be syntactically specified mostly using local constraints, rather than complex grammars. This simplifies the design of visual programming languages from a syntactic perspective.

Our methodology, known as *local context-based visual language specification*, only requires the language designer to define the *local context* of each symbol of the language. The local context of a symbol is the set of attributes that define the local constraints that need to be considered for the correct use of the symbol and are the interface that a symbol exposes to the rest of the sentence.

We also defined a way to do a semantic translation of a visual language based on the local context. In particular, we use XPath-like expressions to define the semantic translation rules for the language. These expressions allow us to specify rules for each single language element, rather than defining semantic rules for complete phrases. For a given node in the abstract syntax graph returned by the syntactic phase, we can use these expressions to gather values from its neighbors to be used in the translation. The translation is then expressed by writing simple source code that prints these values.

Although defining a visual language using the local context methodology does not require writing a grammar (which can be quite complex even for the most experienced users), defining the language can still be time-consuming and identifying at first glance the relationships between the various components of the visual language can be difficult.

For this reason in this paper we propose a new tool, a web application called DrawSE2, that allows the definition of visual languages in an almost visual way. The user can create the visual elements of the language (symbols and connectors) by putting together predefined shapes, then define their attaching areas (the hot spots on which symbols and connectors can be attached). The so prepared language element can be positioned on a canvas and visually related by adding placeholders on their attaching areas. This allows the user to define in a simple way and with immediate visual feedback which symbols can be linked together and which cannot.

The tool also offers the possibility to specify attributes such as the number of admissible occurrences of a symbol or how many times an attaching area can be used through contextual menus or panels. If a semantic translation is required (e.g. to a text representation), the tool also offers the possibility of defining the semantic specification by using a tabular interface.

The visual language so defined can then be used to instantiate a diagram editor that allows one to draw sentences of the language, verify their correctness, and get the semantic translation.

The paper is organized as follows: Section 2 describes previous work in this field; Section 3 describes the local context-based visual language specification; Sections 4 and 5 describe our design and DrawSE2, respectively. Finally, Section 6 concludes the paper with a discussion on future work.

2. Related Work

In the past years significant research has been done regarding visual languages and their applications to different scenarios [4,11–13]. Moreover, several strategies have been developed to model diagrams as visual languages sentences. A diagram has been represented either as a set of attributed symbols with typed attributes representing the "position" of the symbol in the sentence (*attribute-based approach*) [17], or a set of relations on symbols (*relation-based approach*) [25]. The two approaches may look different, but both consider a diagram as a set of symbols and relationships between them, that is, a spatial-relationship graph [2] built by adding a node for each graphical symbol and an edge for each spatial relationship between them.

In contrast to the relationship-based approach, where relationships are explicitly represented, the attribute-based approach requires the relationships to be derived from the attribute (equal) values.

Based on these representations, various formalisms have been proposed to represent the syntax of a visual language, each associated with custom scanning and parsing techniques, e.g. (Extended) Positional Grammars [16], Reserved Graph Grammars [30], Constrained Set Grammars [21], Relational Grammars [27] (for other approaches and details see [14] and [22]). In general, such visual grammars are defined by specifying an alphabet of graphical symbols together with their "visual" appearance, a set of spatial relationships generally defined on symbol position and attaching points/areas, and a set of grammar rules, usually in a context-free like format even though their descriptive power is mostly context sensitive.

A large number of tools exist for prototyping visual languages. These are based on different types of visual grammar formalisms and include, among others, VLDesk [15], DiaGen [24], GenGed [1], Penguin [3], VisPro [31], AToM3 [20], VL-Eli [19] and its improvement DEViL [26], and tools dedicated to 3D visual languages such as [28, 29]. However, our *local context-based visual language specification* [10] goes a step further by completely removing the grammar specification.

Despite the fact that context-free rules are well known, it is not easy to define and read visual grammars. This may explain why these technologies have failed to move from laboratories to real-world applications. Many visual languages used today are syntactically simple languages that focus on basic graphic elements and their expressiveness, so there is no need to specify complex grammatical rules.

3. Local Context Specification of Visual Languages

The local context-based visual language specification allows the definition of a visual language both syntactically and semantically, and also enables the definition of a semantic translation of the visual language sentences (e.g., in text format). Its main feature is that it does not make use of grammars in order to allow easier specification of languages. It has been successfully applied to the definition of various visual real word languages, showing that often there is no need for a grammar definition.

A full description of the methodology can be found in [10]. For reasons of space, it is not possible to describe it here in detail, so we will only indicate here its main features while referring to [10] for a complete definition and examples.

According to the local context specification, a visual language is a set of visual sentences on an alphabet of *symbols* and *connectors* (i.e. *language elements*). Each of them is characterized by the following attributes:

- a unique name;
- a graphical appearance;
- the minimum and/or maximum numbers of admissible occurrences in any sentence of the language;
- one or more attaching areas. Each area is characterized by a *unique name*, its *shape* and *location* on the symbol or connector, a set of *local constraints*, such as the number of possible connections to the area (referred to as *connectNum*), and a *type* used to force legal connections among symbol and connector attaching areas. In fact, a connector area can be attached to a symbol area only if they have the same type.
- a number of symbol level constraints involving more than one attaching area;

Moreover, in order to allow a meaningful visual language translation, *textual attaching areas* are possible, i.e. attaching areas that are designed to only contain text. In this case, the local constraints define instead the set of admissible values for the text (e.g. through a regular expression).

The syntactic analysis uses this information (local to the symbol/connector) to check the correctness of the language and produce a graph called abstract sentence graph. This contains a node for each symbol/connector and an edge between all the symbol-connector pairs that are connected.

The Local Context-based Semantic Definition (LCSD) allows the semantic translation of a visual language. The LCSD consists of a sequence of *semantic rules* for each element of the language. Each rule either calculates a *property* or executes an *action*. The properties are calculated through *procedures* making use of XPath-like expressions and possibly validated through a *post-condition*. An action depends on properties and attributes.

Through the post-conditions, an LCSD may better refine the syntactic structure of the language sentences; through the actions, it provides a translation of the sentences. The semantic analysis algorithm uses a data flow model of execution in order to run the semantic translation rules specified for each language element (as opposed to defining semantic rules for complete phrases). In particular, the XPath-like expressions are exploited to gather values from the element neighbors (and use them in the translation). The new methodology was in part implemented as part of the LoCoMoTiVe tool [7]. In contrast to DrawSE2, it is a desktop application that allows the definition of the visual language only partially through the GUI. It also does not allow a visual representation of how language elements can be linked. In designing DrasSE2, we, therefore, sought to overcome these limitations.

4. Designing a visual representation of visual languages

In designing DrawSE2 we had not only the goal of creating a tool that would allow defining (through the local context technique) a visual language in a simple way, but that would also make the created definition easy to understand. To achieve this we decided that the best way was for the definition to be as much as possible a visual language itself.

To this end, we decided that the visual elements comprising the visual language should simply be placed in the language definition (in the x/y position that the author deems most appropriate). Having shown the language elements, the next most important thing is how the elements of the language can be related to each other in an admissible way. The local context definition uses types (designer-defined names) associated with attaching areas to indicate that a connection is permissible between areas with the same type.

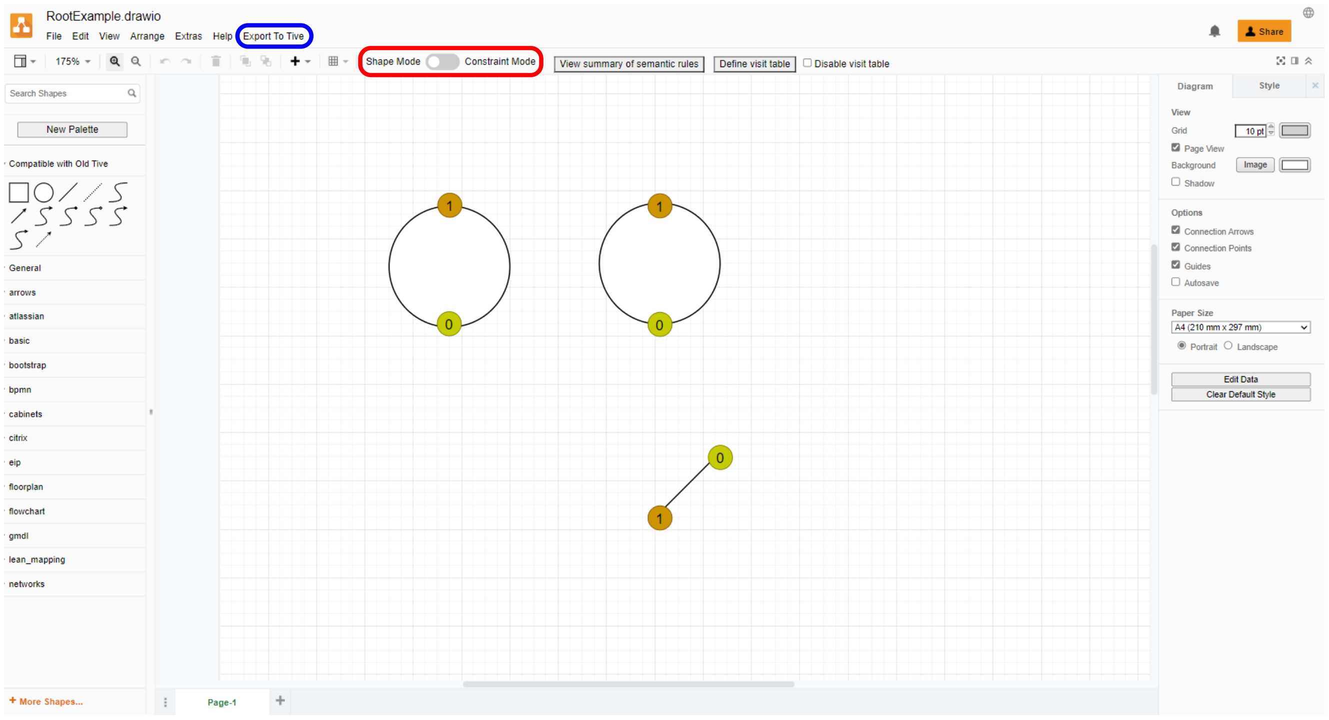
This type of information can be trivially visually displayed by a hyperedge (as it connects two or more attack areas together). In our early designs, we tried some of the typical ways to visually represent a hyperedge, but we noticed that, for example in Euler-style visualization, as the number of language elements increases it becomes increasingly visually heavy and difficult to understand (and difficult for the user to draw). For this reason, we decided to use numbered placeholders (colored circles with a number inside them) to indicate that attaching areas with the same placeholder represent allowable connections, as shown in Figure 1.

Although it would be easy to show other information in this type of visualization, such as the number of permissible occurrences for a language element or the number of permissible connections for an attaching area (simply by showing the related text next to the relevant element), we decided to exclude such information from the immediately accessible visual representation because it made it too visually heavy, and decided that such information should be visible only after selecting a specific language element.

These design decisions were then included in DrawSE2.

5. DrawSE2

DrawSE2 is an application that allows visual language definition according to the local context specification in a visual/GUI way, making it easy to define how symbols can be linked together and minimizing the amount of code to be written. This is expected to make the definition of a visual language easier and more understandable even for users with minimal knowledge of grammars and programming. DrawSE2 is based on *diagrams.net* [18], a web application that allows diagrams to be composed using predefined sym-



175%

Grid 10 pt

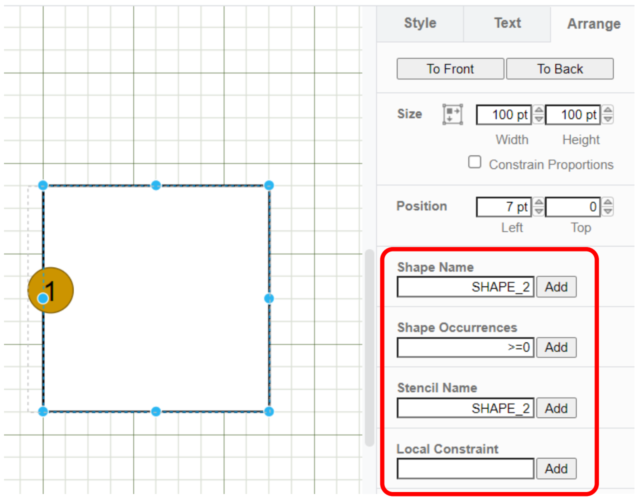
Paper Size A4 (210 mm x 297 mm)

Figure 1: DrawSE2 in shape mode.

bols and connectors, but does not include syntactic/semantic analysis capabilities.

DrawSE2 has two modes: the shape mode in which one can define visual language elements (symbols and connectors, including the composition of predefined elements), element attributes, and semantic rules; and the constraint mode in which one can define the attaching areas for the language elements and the admissible connections between them. It is also possible to export the created language, i.e., to run an instance of an editor (also based on *diagrams.net*) in which it is possible to draw diagrams of the newly defined language and use the syntactic/semantic analysis functions to check for correctness and obtain the semantic translation (e.g., into text) of the drawn diagram.

Figure 1 shows a screenshot of DrawSE2 in shape mode. It is possible to switch to constraint mode and vice versa using the corresponding switch (highlighted in the red box): this changes the display of the canvas contents and the side menu. There are also menus and buttons offering features typical of graphic editors (and already included a *diagrams.net*) such as zoom, undo/redo, for changing properties and styles of graphic elements, etc., which are then available both in shape mode and in constrain mode. Finally, it is possible to export the language using the “Export to TiVe” menu (highlighted in the blue box): this causes the editor to open in a new browser tab. In the next sections, we will describe each mode in detail.



Size: 100 pt Width, 100 pt Height

Position: 7 pt Left, 0 Top

Shape Name: SHAPE_2

Shape Occurrences: >=0

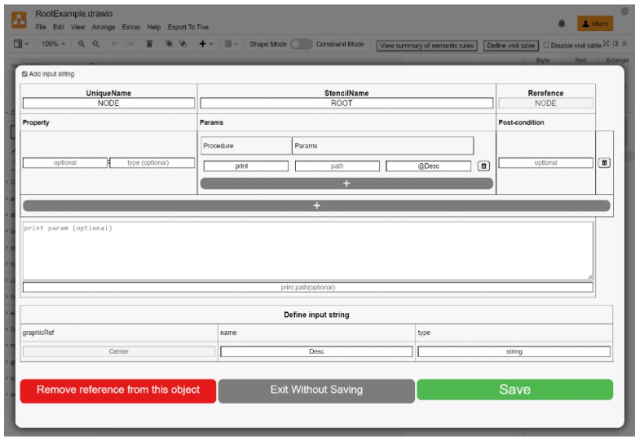
Stencil Name: SHAPE_2

Local Constraint:

Figure 2: Defining the attributes of an element.

5.1. Shape Mode

The Shape mode allows one to define the symbols and connectors used in the language. For a symbol or connector to become part of the language, it is sufficient for it to be placed on the canvas by dragging it from the panel on the left, which contains a set of predefined symbols and connectors. One can also create their own custom symbols by putting together several predefined elements. Such



☑ Add input string

UniqueName	StencilName	Rerefence
NODE	ROOT	NODE
Property	Params	Post-condition
optional / type (optional)	Procedure: print; Params: path, @Desc	optional

print param (optional)

print path(optional)

Define input string		
graphicRef	name	type
Center	Desc	string

Figure 3: Table for defining semantic rules.

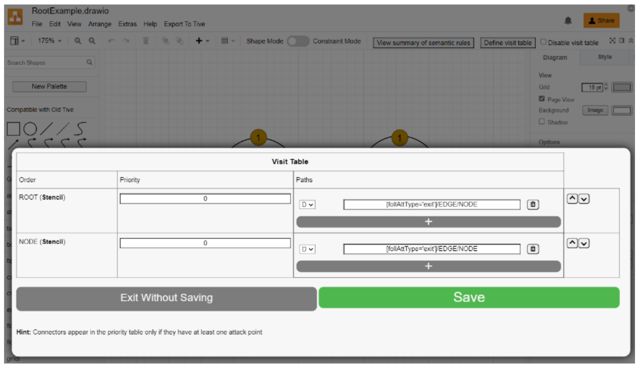
predefined elements also include lines and curves so there is considerable flexibility, considering also that it is possible to import graphic elements in png format. This is accomplished by selecting individual elements and using the group function found in the context menu (accessible by right-clicking).

By selecting a symbol it is also possible to define the following attributes (from the local context methodology) thanks to the panel on the right: name; number of permissible occurrences in a language sentence; expanded name (shown to language users), local constraints involving multiple attaching areas (more on this in the next section). An example is shown in Figure 2.

Also in this mode, semantic rules can be defined. To perform this action there is a "Define semantic rules" option in the context menu accessible for each element. In this case, an editable table will be shown that will allow these rules to be defined congruently with the local context methodology.

In particular, the table, as shown in Figure 3, allows defining the list of symbol properties and how these are to be computed. There is also a text area in which the code to produce the semantic translation of that language element can be entered. For convenience, there is the listing of the textual attaching areas of the symbol, if any (since they are likely to be referenced in the code that produces the semantic translation, since they may contain user-written text). Finally, there are buttons to save, close, or remove the table altogether.

The "Define visit table" button at the top of the GUI allows one to define the visit table, which is used to define the order in which language elements will be visited during semantic analysis/translation. In particular, it is possible to define the order and priority of each element, and the paths associated with them. Again, there are buttons to save and close, as shown in Figure 4.



Visit Table		
Order	Priority	Paths
ROOT (**Stencil**)	0	D [follAttType='exit']/EDGE/NODE
NODE (**Stencil**)	0	D [follAttType='exit']/EDGE/NODE

Hint: Connectors appear in the priority table only if they have at least one attack point

Figure 4: Example of a visit table definition.

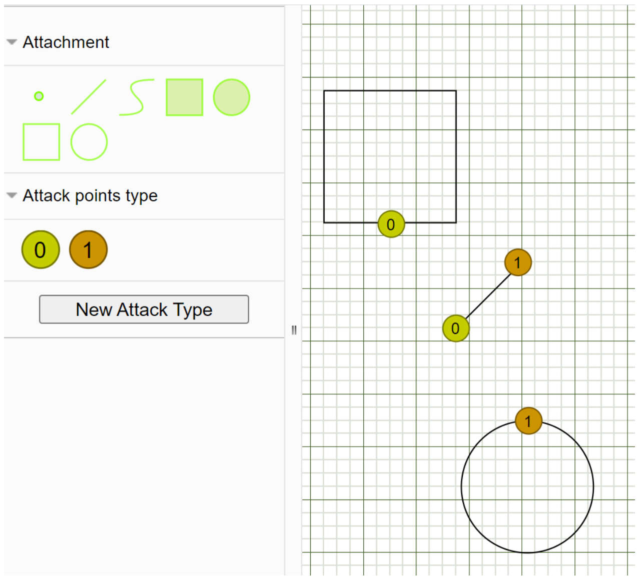
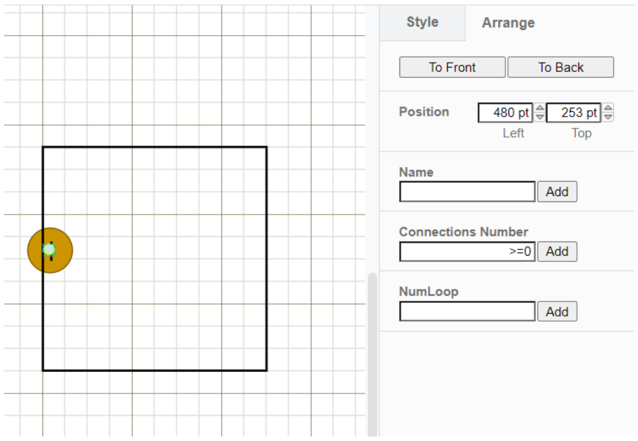


Figure 5: Example of using "New Attack Types".



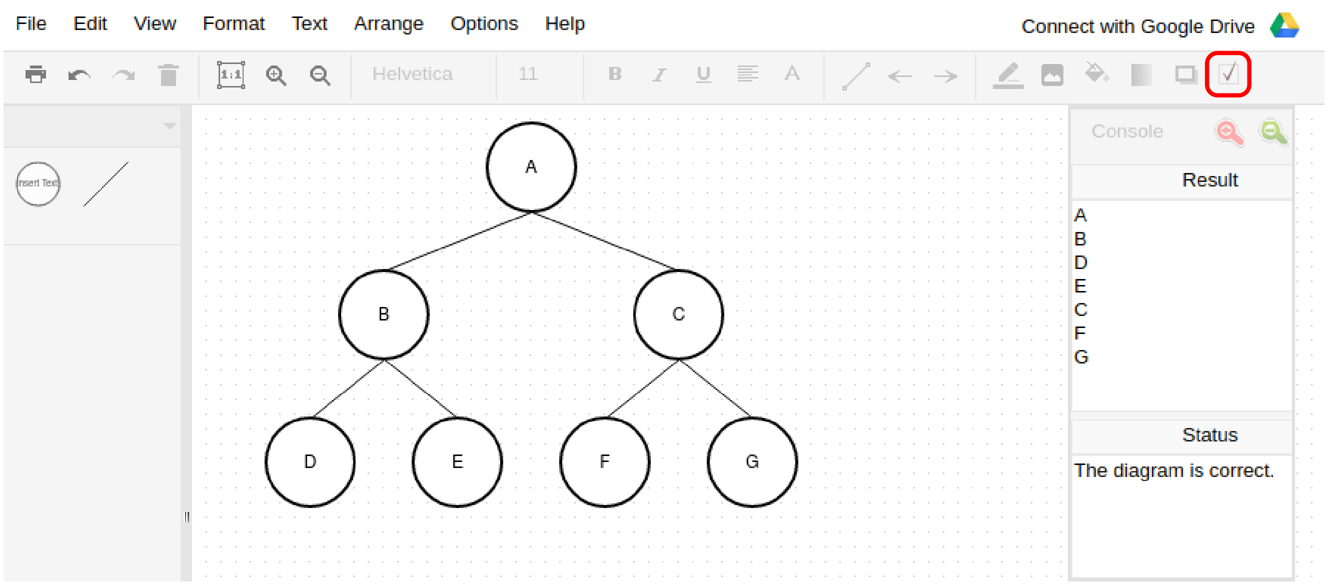
Position 480 pt Left, 253 pt Top

Name

Connections Number >=0

NumLoop

Figure 6: Defining the properties of an attaching area on a symbol.



```
A
B
D
E
C
F
G
```

The diagram is correct.

Figure 7: Example of editor created for the tree language. The button highlighted in the red box launches the correctness check and semantic translation. The semantic translation (a preorder visit) is shown on the right panel.

5.2. Constraint Mode

The Constraint Mode allows one to define the attaching areas for each symbol or connector in a visual way. When switching to constraint mode, the language elements remain visible but become uneditable, and the side panel on the left instead shows the set of attaching areas ("Attachment" panel in Figure 5) that one can drag onto the language elements in the canvas in order to add them.

It is also possible to define how language elements can be connected to each other. This can be done by creating placeholders (through the "New Attack Types" button). They are represented graphically by circles of different colors and containing numbers. Placeholders with the same number can be placed on the attaching areas of language elements in order to indicate that these elements can be connected together in a valid visual sentence, as shown in Figure 5.

It is also possible to define properties (from the local context methodology) for each attack type, including: name (optional: useful if one needs to reference it in the semantic specification); maximum number of elements that can be attached to it; limits on self-loops (connector leaving and arriving on the same area), as shown in Figure 6.

5.3. Visual language editor: TiVe

The TiVe editor uses the visual language definition and is also based on *diagrams.net*. It differs from it in that it shows in the left panel only the symbols and connectors that are part of the language, and that it adds a button that allows the user to perform the correctness check of what has been drawn. If successful, the semantic translation (if defined) or a message confirming correctness will be shown, or an error message otherwise. Figure 7 shows an example of this editor.

6. Conclusions and further works

We presented DrawSE2, a new tool for defining visual languages in a more visual way (according to local context methodology). The user can create the visual elements of the language, define their attaching areas, and define how they may connect with each other by placing placeholders on their attaching areas. The tool also allows the user to define the semantic attributes and the semantic translation through tabular GUI.

Future work will involve empirical evaluation of the software by performing user studies involving both experts in visual languages and users with no experience in the field. Regarding the latter users, it is planned to involve computer science students studying compilers (of textual programming languages). After this evaluation, the software will be further refined according to the received feedback and by correcting any remaining bugs.

7. Acknowledgment

The authors thank Gerardo Brescia and Vincenzo Caputo for their support in implementing DrawSE's earlier versions.

References

[1] R. Bardohl. Genged: a generic graphical editor for visual languages based on algebraic graph grammars. In *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, pages 48–55, Sep 1998.

[2] R. Bardohl, M. Minas, G. Taentzer, and A. Schürr. Handbook of graph grammars and computing by graph transformation. chapter Application of Graph Transformation to Visual Languages, pages 105–180. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[3] S. S. Chok and K. Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, pages 185–194, New York, NY, USA, 1998. ACM.

[4] G. Costagliola, M. De Rosa, A. Fish, V. Fuccella, R. Saleh, and S. Swartwood. Knotsketch: A tool for knot diagram sketching, encoding and re-generation. In *The 22nd International Conference on Distributed Multimedia Systems*, pages 16–25, 2016.

[5] G. Costagliola, M. De Rosa, and V. Fuccella. Local context-based recognition of sketched diagrams. *Journal of Visual Languages & Computing*, 25(6):955–962, 2014.

[6] G. Costagliola, M. De Rosa, and V. Fuccella. Local context-based recognition of sketched diagrams. In *The 20th International Conference on Distributed Multimedia Systems*, pages 321–328. Knowledge Systems Institute, August 2014.

[7] G. Costagliola, M. De Rosa, and V. Fuccella. Extending local context-based specifications of visual languages. *Journal of Visual Languages & Computing*, 31, Part B:184 – 195, 2015.

[8] G. Costagliola, M. De Rosa, and V. Fuccella. Fast prototyping of visual languages using local context-based specifications. In A. Guercio, editor, *The 21st International Conference on Distributed Multimedia Systems*, pages 14–22. Knowledge Systems Institute, August 2015.

[9] G. Costagliola, M. De Rosa, and V. Fuccella. Fast prototyping of visual languages using local context-based specifications. *J. Vis. Lang. Sentient Syst.*, 1, 2015.

[10] G. Costagliola, M. De Rosa, and V. Fuccella. Using the local context for the definition and implementation of visual languages. *Comput. Lang. Syst. Struct.*, 54:20–38, 2018.

[11] G. Costagliola, M. De Rosa, V. Fuccella, and M. Minas. Visual exploration of visual parser execution. *Multimedia Tools and Applications*, 81(1):299–317, 2022.

[12] G. Costagliola, M. De Rosa, V. Fuccella, and S. Perna. Visual languages: A graphical review. *Information Visualization*, 17(4):335–350, 2018.

[13] G. Costagliola, M. De Rosa, and M. Minas. Visual parsing and parser visualization. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 243–247, 2019.

[14] G. Costagliola, V. Deufemia, and G. Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Trans. Softw. Eng. Methodol.*, 13(4):431–487, Oct. 2004.

[15] G. Costagliola, V. Deufemia, and G. Polese. Visual language implementation through standard compiler–compiler techniques. *Journal of Visual Languages & Computing*, 18(2):165 – 226, 2007.

[16] G. Costagliola and G. Polese. Extended positional grammars. In *Proceeding 2000 IEEE International Symposium on Visual Languages*, pages 103–110, 2000.

[17] E. J. Golin. Parsing visual languages with picture layout grammars. *J. Vis. Lang. Comput.*, 2(4):371–393, Dec. 1991.

[18] JGraph Ltd. diagrams.net. https://www.diagrams.net, 2022.

[19] U. Kastens and C. Schmidt. Vl-eli: A generator for visual languages - system demonstration. *Electr. Notes Theor. Comput. Sci.*, 65(3):139–143, 2002.

[20] J. d. Lara and H. Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *Fundamental Approaches to Software Engineering*, pages 174–188, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[21] K. Marriott. Parsing visual languages with constraint multiset grammars. In *Programming Languages: Implementations, Logics and Programs*, pages 24–25, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[22] K. Marriott and B. Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages & Computing*, 8(4):375 – 402, 1997.

[23] Y. Matsuzawa, Y. Tanaka, and S. Sakai. Measuring an impact of block-based language in introductory programming. In *SaITE 2016 - Stakeholders and Information Technology in Education - IFIP Advances in Information and Communication Technology*, volume 493, pages 16–27. Springer, Cham, 2016.

[24] M. Minas and G. Viehstaedt. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *Proceedings of the 11th International IEEE Symposium on Visual Languages*, VL '95, pages 203–, Washington, DC, USA, 1995. IEEE Computer Society.

[25] J. Rekers and A. Schurr. A graph based framework for the implementation of visual environments. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 148–155, Sep 1996.

[26] C. Schmidt, U. Kastens, and B. Cramer. Using devil for implementation of domain-specific visual languages. In *Proceedings of the Workshop on Domain-Specific Program Development*, page 38, 2006.

[27] L. Weitzman and K. Wittenburg. Relational grammars for interactive design. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 4–11, Aug 1993.

[28] J. Wolter. Devil3d - A generator framework for three-dimensional visual languages. In *Proceedings of the 18th International Conference on Distributed Multimedia Systems, DMS 2012, August 9-11, 2012, Eden Roc Renaissance, Miami Beach, FL, USA*, pages 171–176. Knowledge Systems Institute, 2012.

[29] J. Wolter. Specifying generic depictions of language constructs for 3d visual languages. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 139–142, 2013.

[30] D.-Q. Zhang and K. Zhang. Reserved graph grammar: a specification tool for diagrammatic vpls. In *Proceedings. 1997 IEEE Symposium on Visual Languages (Cat. No.97TB100180)*, pages 284–291, 1997.

[31] D.-Q. Zhang and K. Zhang. Vispro: a visual language generation toolset. In *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No.98TB100254)*, pages 195–202, 1998.