ViCoLL – a Visual Compositional Logic Language

Anne Håkansson, Lars Oestreicher, Torsten Jonsson and Andreas Hamfelt

Computer Science Division, Department of Information Science

Uppsala university, Box 513, S -751 29 Uppsala, Sweden

Anne.Hakansson@dis.uu.se

1. Introduction

CombiLog is a logic programming language that uses a small set of basic operators to build larger structures. Although the programs are built strictly through composition, the resulting code is often difficult to read and understand. Empirical studies on program creation both with VPL and traditional programming indicate that visual approaches may outperform traditional approaches to programming. Thus, CombiLog could gain largely from a visual interface to support the development process. With a visual interface the resulting system can be regarded as a declarative Visual Logic Programming language, ViColl.

2. The interface of CombiLog

The visual programming language ViCoLL (a VIsual Compositional Logic Language) is intended to fulfil the goals with VPLs by making the programming more accessible as well as improve correctness and programming speed. Therefore, the language supports conceptual simplicity [6], concreteness, directness, explicitness and liveness [2]. ViCoLL supports simplicity by using only a few simple concepts and concreteness by actively expressing the content of the program. Direct manipulation by using *drag-and-drop* to put the boxes on the screen and connect them with other provides the directness, and explicit semantics are stated textually and visually through object names and metaphors. Finally, the liveness will be supported since ViCoLL will provide semantics feedback when a program is being edited. It can be further enhanced if ViCoLL can be made to support continuous visualisation.

ViCoLL is built on top of the declarative programming language CombiLog [3]. It preserves the notion of being a declarative language by not applying any procedural meaning on how different operators actually are executed, how they are combined into larger operators with more content or how they are placed in the interface. Furthermore, it provides an interpretation in a set-based symbolic language since CombiLog has a model-theoretical semantics, which corresponds to logic displaying sets in graphical diagrams.

Some fundamental ideas in the graphical interface of ViCoLL have emerged from Puigsegur et al. [5], [1] where *Higraphs* are used for development of visual Logic Programming Language. Higraphs are graphs where nodes are related to Venn diagrams. Compared to ViCoLL, there are some differences, though. ViCoLL does not involve the whole concept of Higraphs but uses a mix of both the ideas behind and the appearance of Venn diagrams and Euler diagrams in three-dimensional form.

Venn-Euler diagrams can facilitate visualisation of model-theoretical composition of programs. Our aim is that in program development, the model-theoretical semantics will correspond to the programmer's mental model of the problem and its solution. This may help the programmer to think in more set-based manner when putting operators together. To be able to create an application to solve a problem by using CombiLog, the problem has to be described in terms of CombiLog's operators, which have to be combined into larger structures.

Both Venn diagrams and Euler circles tend to get cluttered if many curves or circles are used in the diagrams. This ha been considered in the interface. Therefore, the Venn diagram is used in a slightly different manner where the circles may not be intersections. The *intersection* of circles will correspond to the *and*-operator in CombiLog. The *disjunction* will correspond to the *or*-operator. Here an elliptic shape is used and may connect other circles that are similarly coupled by *and*- or *or*-operators. To minimise the negative effects with the Venn diagrams, a three-dimensional environment is used. With 3D-presentation more information can be displayed, still avoiding the cluttering of circles. Also, the declarative interpretation may be preserved if all operators can be used in all directions.

Beside *and*- and *or*-operators, the elementary predicates (*id*, *const*, *cons* and *true* [4]) are visually represented. Moreover, when a new operator is constructed from smaller operators it is visually represented. One of the most significant issues in ViCoLL is that all operators have a consistent layout. All operators are labelled with a word that describes the action, e.g., by the word *Identity* or *Id* on the id-operator. This label is not produced by the system, but relies on the user to pick a suitable word or a metaphor to illustrate a new created operator's action.

Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments (HCC'01)
0-7695-0474-4/01 $17.00 © 2001 IEEE

IEEE COMPUTER SOCIETY

Venn diagrams are represented with frames surrounding the circles in a diagram and therefore a kind of visual frame is used in this interface to surround the operators and their action-label. This reduces the cluttering in the interface and simplifies the illustration of the content in the application during development. This is also useful in order to facilitate *programming in large*.

The Venn diagrams' possibility to shadow areas in the circles will not be used because this would minimise the readability in the interface. Anyway, this effect can still be achieved if the circles are semi-transparent since when these circles are coupled, this semi-transparency will become lesser, i.e. a coupling circle has a slightly darker hue than a single circle.

To simplify programming using ViCoLL, the complexity has been reduced by letting the most significant operator, the manipulation operator — *make* — be implicit in the interface and put automatically into the code. Technically, this is possible only if the number of arguments can be made present in the interface. The arguments are not illustrated in the example picture but could easily be inserted in the spheres.

Example in Prolog (transitive closure):

```
t_closure(X,Y) <- p(X,Y).
t_closure(X,Y) <- p(X,Z),
          t_closure(Z,Y).
```

and the corresponding example in CombiLog [3]:

```
t_closure <-
   or(make[ 1,2] (make[ 1,2] (p)),
      make([ 1,3] (and(make[ 1,2,3] (p),
           make[ 3,1,2] (t_closure)))).
```

And the corresponding interface for this example is:

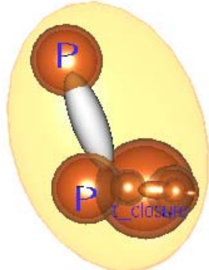

Figure 1. An example of a ViCoLL structure

In figure 1, the ellipse corresponds to the *or*-operator in CombiLog. The purpose of this operator is to couple different possible solutions and to keep the program or application as a unit. In comparison to the Prolog language, this operator corresponds to the combining of different clauses for a single predicate. At both ends of the circle, spheres are placed, which corresponds to the *and*-operators. The number of spheres put together is due to the properties of programming language. At the top of the figure the single sphere corresponds to the first statement in CombiLog example (p). This has the same meaning as the content within a single predicate clause in Prolog. In the case where two spheres are put together it corresponds to the second statement in the CombiLog example (p and t-closure). By letting these spheres be shadowed when sharing the same area (i.e. they become less transparent in that area) it is indicated that they are sharing the same tuples. These spheres may represent individual terms as well as operators. What the spheres represent in turn will not be present in the interface until the user explicitly asks for it. If so, the label that contains information about the tuples, the performance of the *make*-operators and how different tuples are connected, will be unfolded into a similar sub-structure, see figure 2.

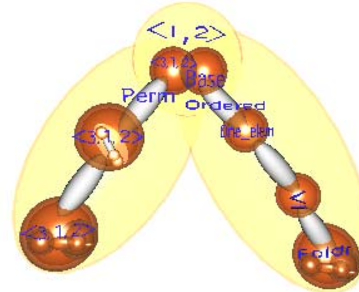

Figure 2. Unfolded Sort program.

3. Summary

This paper presents a visual logic programming language. This language is called ViCoLL. It is designed to support the programmer to write logic programs more easily. In the interface the program components are combined into larger structures. This may help the programmer to think in a more set-based manner when putting operators together. Comparing their intensions to the visual representation, the programmer may find it easier to judge if the model in ViCoLL corresponds to their mental models of the language model-theoretical semantics.

4. References

[1] Agusti J., Puisegur J. and Robertson D., A Visual Syntax for Logic and Logic Programming. Journal of Visual Language and Computing 9, 1998, pp. 399-427

[2] Burnett M., Atwood J., Ambler A., Implementing Level 4 Liveness in Declarative Visual Programming Languages, TR 98-60-03, Oregon State University, 1998.

[3] Hamfelt, A. & Nilsson, J. F. “Towards a Logic Programming Methodology based on Higher-order Predicates.” Journal New Generation Computing vol 15, no 4, 1997, pp. 421 – 448

[4] Hamfelt, A. & Nilsson, J. F. “Inductive Synthesis of Logic Programs by Composition of Combinatory Program Schemes.” In Flener, P. (ed.) Logic Based Program Transformation and Synthesis, Lecture Notes in Computer Science vol 1559, Berlin: Springer-Verlag, 1998, pp. 143 – 158.

[5] Puigsegur J., Agustí J., Robertsson D,. Proceedings of VL'99, September 6-9, Boulder, Colorado, USA, 1996.

[6] Puigsegur J., Schorlemmer M., Agustí J., From Queries to Answers in Visual Logic Programming. IEEE Proceedings of VL’97, Sept 23-26, Capri Italy, 1997.

IEEE COMPUTER SOCIETY