Two-dimensional C++

Johannes Reichardt
Department of Computer Science
Hochschule Darmstadt
j.reichardt@fbi.h-da.de

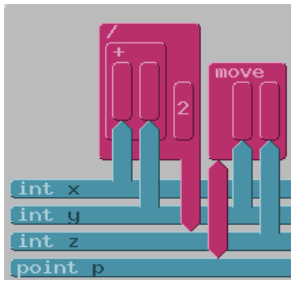
1 Introduction

The present visual language supports the imperative programming style and claims to improve program comprehension by resemblance. It can be characterized as follows: 1) Its scope is the static description of code, data and flow control as well as of the surrounding architecture including classes, components, modules, up to systems. 2) Its abstraction level is next to that of C++ including templates and the standard template library and allows straight-forward code generation. 3) The visualization is specified by editing boxes, bars, arrows and textual labels on a drawing area with relatively fixed editing positions. 4) The interface of the visualization is the drawing area itself, enabling further editing or navigating through the program by clicking on the bars, boxes and labels or by defining views via textual or topological queries. 5) The presentation of the visualization is by a (still hypothetical) wall-projection of the virtual drawing area.

Resemblance between the program domain and the problem domain can be achieved by a structure-preserving mapping known as a homomorphism. A program is a homomorphic image of the problem domain when the semantic relationships among the objects of the problem domain are preserved by the syntactical relationships among their counterparts in the program. The syntactical relationships can be encoded by a placement of syntactical elements onto meaningful positions in the two-dimensional design space, thus conveying information additional to the functionality of the code. The mapping distinguishes between two design levels. Algorithms are mapped using a process metaphor visualizing state and state transitions along a time axis. On the architectural level, the interrelations between concepts of the problem domain are encoded by different types of syntactical neighbourhood among class and module definitions. Different abstraction levels, in particular the levels of definition and use of types, are mapped onto different layers represented by boxes positioned on top of each other. Nesting of boxes is used to encode sub-classes and sub-modules, and boxes are positioned side by side when they denote concepts of equal right. A homomorphic image of a real-world process also requires the preservation of terminology for labelling the boxes, on ideal terms taken from a thesaurus of the application domain. Moreover, there are semantic relationships in the terminology to be preserved in that sub-boxes must be labelled by sub-concepts and super-boxes by super-concepts. A technical problem exists with the display of large coherent graphs. We propose some mechanisms of abstraction to scale down the amount of information to semantic or user needs.

2 Language Elements

Operands and Operations. A method definition consists of a sequence of operands (objects) represented by a vertical sequence of horizontal bars and a sequence of operations (method calls) represented by a horizontal sequence of boxes. Between the operands and operations, element, argument and assignment relations are visualized by arrows ending at or starting from the left, middle or right lower edge of an operation-box, resp. (see Fig. 1).

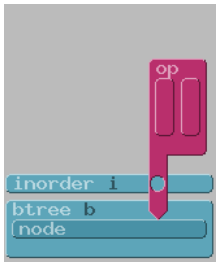



```
int      x,y,z;
point    p;

z=(x+y)/2;
p.move(y,z);
```

Figure 1: Arrows expressing element, argument and assignment relations, with generated code

Data Structures, Containers and Iterators. The hierarchical structure of objects is visualized by a vertical nesting of object-bars. We use four different types of object: Basic objects, records, containers and iterators used in conjunction with containers. The elements of a container are visualized by a single representative object (of type 'node' in Fig. 2) enclosed in the container bar (of type 'btree' and named 'b', in Fig. 2).




```
inorder  i;
btree    b;

b[i].op(...,...);
```

Figure 2: Container-access via iterator, with generated code

Access to a container element is visualized by an arrow ending at the representative object of that container and crossing an

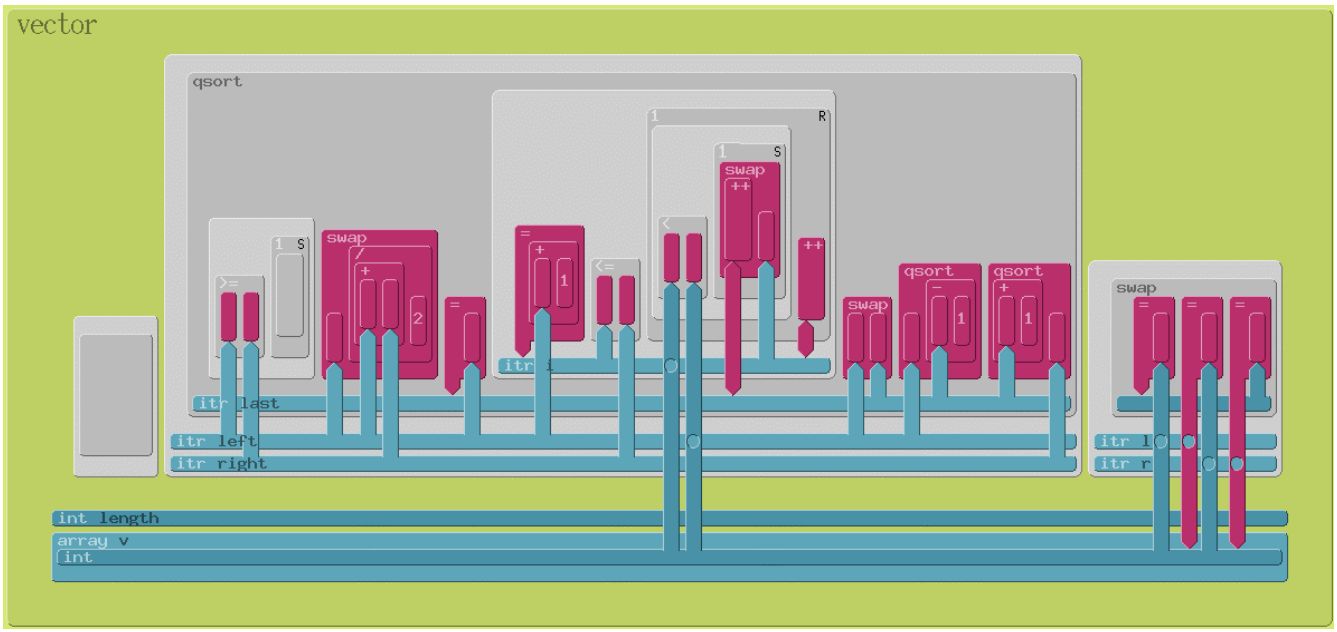
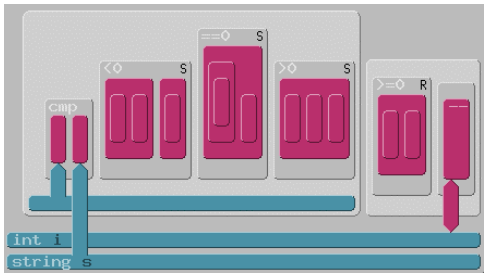


Figure 4: Class *vector* with methods *qsort* and *swap*

iterator object (of type 'inorder' and named 'i', in Fig. 2) above that container. The crossing-point of arrow and iterator is flagged by a marker which plays the role of a dereference operator.

Control Structures. A control structure is composed of case boxes of type *select* (S) or *repeat* (R) preceded by a designated operation which forms the first part of a condition. The label in the upper left corner of each case box forms the remainder of that condition, resp. (see Fig. 3). A case-box is executed if the result of the designated operation (first part of the condition) satisfies the second part of the condition consisting of a comparator followed by a constant, which form the label of the case-box.




```
if (cmp(…,s)<0){…} else
if (cmp(…,s)==0){…} else
if (cmp(…,s)>0){…};
do {…} while ((i--)>=0);
```

Figure 3: Flow Control, with generated code

Methods and classes. Both method and class definition can be represented as functional units each consisting of a sequence of boxes communicating via a sequence of bars. In the case of a method definition, the boxes represent method calls embedded in flow control, whereas the bars represent local objects and parameters. In a class definition, the boxes represent method definitions while the bars represent the attributes. Figure 4 outlines a class definition. C++ code generation is described in [Reichardt 2002].

3 Views

A fractal view of a box shows the scope of visibility relative to the box focussed on (see Fig. 5). It represents the minimum of information to comprehend and edit that box. The view has a logarithmic information-reduction effect and, as a homomorphic image, preserves resemblance. Other views are possible.

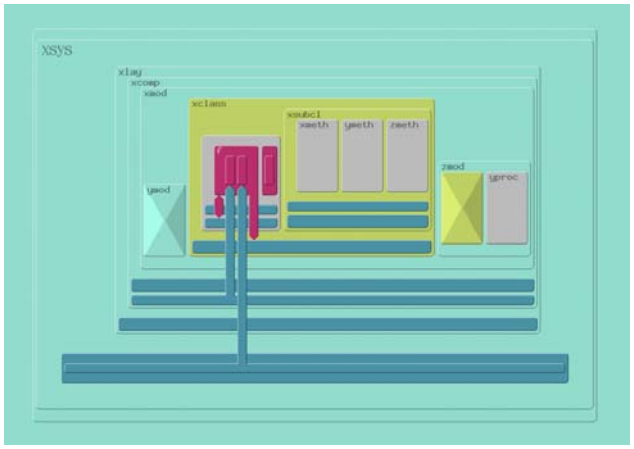


Figure 5: Fractal view of a module

References

REICHARDT, J. 2002. Method and System for Visual Programming. *German Patent 199 07 328.*