

# The Transition of Context-free Textual Languages into a Visual Programming Notation via Graph Techniques and a Meta Tool

*Frank Bühler, Mike Callaghan and Paul Luker*

Department of Computing Sciences  
De Montfort University  
The Gateway  
Leicester LE1 9BH  
England

E-mail : fbuehler@acm.org

**Keywords:** visual programming, language design, program development.

## Abstract

*The design and implementation of a new visual programming language is a difficult task. The article presents a new meta tool which eases the initial design phase. The approach differs from existing techniques as it is based on an easy-to-use description language, each lexeme of which is represented as an individual node on the workplace and is described by a special code file. The broad aim of this research is to investigate how visual elements/techniques could be integrated into a concrete visual environment which supports the programming task. Thus, the emphasis lies primarily on the visualisation of structures or relationships rather than on implementation details. To keep the ongoing research as open as possible, the separation of the underlying semantics from the visual representation is of great importance.*

---

Copyright 1998 IEEE. Published in the Proceedings of Compsac'98, 17-21 August, 1998 at Vienna, Austria. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE.  
Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 732-562-3966.

---

## 1 Introduction

### 1.1 Background

Current textual languages like C, C++, or Java, to name the obvious, play an important role in program development. Compilers for these languages are integrated in a more or less comfortable graphical environment. While the applications which are built by these systems become increasingly more complex, new ways are required to enable the production of more robust programs, which have to be written in less time. The question which remains open is "what technique (or combination of techniques) is necessary to make the inherent structures of applications more visible in order to reduce the cognitive load for an application developer?". One promising field which addresses this question is visual programming (VP).

Since the wider availability of cheap graphical workstations and PCs, research in the area of VP has become more important. The authors are convinced that combining the strengths of proven techniques (such as object-oriented programming) and visual programming will yield a significant improvement in program design and development. Of course, much work has to be done to identify a methodology to form a new basis for this kind of development.

## 1.2 Important terms

Though the term “visual” is widely used, it is important to describe exactly how VP differs from traditional textual programming. A language is called a *visual language* if it handles visual information or supports visual interactions. Chang [Chang-1987] calls a language *visual* when objects handled by the language are visual, the language processes visual information, or if the language itself is visual. The language is called a visual programming language if it is actually used for programming with visual expressions (e.g. iconic sentences). Different classification schemes for VP have been published, for example, Chang [Chang-1987], Myers [Myers-1986], and Shu [Shu-1986], [Shu-1988]. It is important to notice that the definitions contained in those works differ significantly. A good classification of visual object-oriented languages is presented in [Burnett-1995]. There follows a more verbal description of what “visual programming” means.

A system may be called a visual programming system if:

- it covers important aspects of programming (i.e. syntax and semantics).
- it makes use of graphical techniques (for instance, the program design is simplified through the use of images and pictorial representations).
- ideally, the visual representation is not just a picture of the logical program structure but of the executable program itself (one example is *VisaVis* [POSWIG-1996]).
- the user interaction contains both textual and graphical elements.
- the aim of the system’s developer is to make use of non-verbal human abilities (i.e. right hemisphere tasks). Non-verbal and visio-spatial abilities are primarily located in the right hemisphere of the brain. Language and logical abilities are concentrated in the left hemisphere.

## 2 Related Work

A number of VP systems have been developed over recent years. In the following sections, two different approaches which have influenced the design of the meta tool described in this paper are outlined.

## 2.1 Vista

**Vista** (Visual Software Technique Approach) [SCHIFFER-1998], [BURNETT-1995] is a visual programming environment designed for software engineers. It is based on a *visual multi-paradigm language* which is firmly linked to the development system. It supports a mix of textual and graphical notation. Using Vista, it is possible to write event-driven and data-transforming Smalltalk applications.

A program in Vista consists of a hierarchy of building blocks called processors. Programming takes place by specifying processors and connecting them into a network. The interaction and control flow is triggered through tokens which are sent via the I/O ports of the processors. There are several kinds of processors, such as data processors or signal processors. Vista includes many useful features such as abstract processor classes, prototypes, and self-modification [SCHIFFER-1998].

In summary, Vista contains a very powerful programming model and a high degree of visual expressiveness, such that it outperforms many other VP systems.

## 2.2 Vampire

One obstacle to the design of new visual programming languages, as mentioned before, is the effort required to design and implement the language “under consideration”. As this experimental approach doesn’t always result in a usable system, it is very beneficial to make use of a rapid prototyping system, such as **Vampire** [MCINTYRE-1995], which “allows language designers to explore new concepts efficiently”.

*Vampire* is an acronym and stands for Visual Metatools for Programming Iconic Environments. The system was inspired by a graphical reasoning tool (viz. Furnas’ BITPICT system [FURNAS-1990], [FURNAS-1991]) to help create iconic programming languages for a variety of domains. Not only is the language creation process object-oriented but Vampire, based on a rule-based graphical system, supports the construction of languages which are themselves object-oriented. Attributed graphical rules, similar to those used in

BITPICT and ChemTrains [BELL-1993], are the vehicle for designing the visual language.

A Vampire rule frame consists of two sides. On the left side, it contains graphical elements (i.e. *constraint text* and *constraint graphics*) “which are matched against a runtime workspace” [MCINTYRE-1995]. The right side is the action side, which contains *action text* and *action graphics*. The textual areas contain Smalltalk expressions, the graphical components icons. When a match of the left side is found, the screen area is transformed to the contents of the right side. In this respect, it is directly comparable to production rules used during the specification of textual languages. To ease the task of designing the rules, a rule editor and an icon editor are used.

After the semantics of the iconic visual programming language have been defined, a program may be created through selections on pull-down menus. Thus, Vampire doesn't use a drag&drop mechanism, or any other advanced user interaction styles found in many other related systems. Using the run-time system, the program may be executed either in a static or dynamic execution mode. In the latter mode, execution is explicitly started by the user.

It can be concluded that Vampire is a powerful system, which promotes the discovery of new visual languages by reducing the amount of lines of code necessary to implement and test a new idea. McIntyre states that the system has some weaknesses in that it is not able to restrict legal syntax in a language and the interaction style is not very advanced. However, as the system may adapt to new language paradigms in a very flexible way, the benefits of Vampire outweigh its lack of abilities. The new meta tool presented in this article tries to overcome these problems, and shows an alternative approach.

## 3 The Visual Meta Builder

### 3.1 Motivation

In the previous section, two systems incorporating different levels of visualisation have been outlined. As graphical user interfaces have proven to be highly beneficial for human-computer

interaction the same is certainly true of programming tasks. Further, it is highly predictable that visual programming systems will play a dominant role in the near future, as they offer the ability to build systems to those without a rigorous software engineering background. This is borne out by the creation of new commercial development systems such as the Visual Age product family based on visual programming techniques (here the parts paradigm) [NILSSON-1997].

However, little research has taken place to develop a system that eases the design of new visual programming languages. As with any *visual language*, a *formal basis* and an underlying *visual metaphor* is needed. The formal basis could, for instance, rely on a mathematical theory or formalism or on a higher level building block. In this instance it is not necessary to invent a whole new (visual) language from scratch but to take a proven textual language and extend it so that it is well embedded within a visual environment. A suitable metaphor applied to the construction of an application could be for instance “the principle of substitution” (i.e. using specialisation), “the key-hole idea” (i.e. using software components which may be dragged into a pre-defined slot), or “the building blocks idea” (i.e. using composition of black-box components). So, when designing a meta tool, it is of great importance to use a generic underlying principle.

### 3.2 Introduction to the approach

In the approach presented here the *processor idea* is used wherein, simply, a processor specifies an “action” which will be performed after it has been triggered by an incoming event. Thus, a processor contains an imperative character. Further, processors are oriented towards ready-to-use, or reusable components. A program is called a processor program and consists of many interconnected processors placed on different layers of a graph.

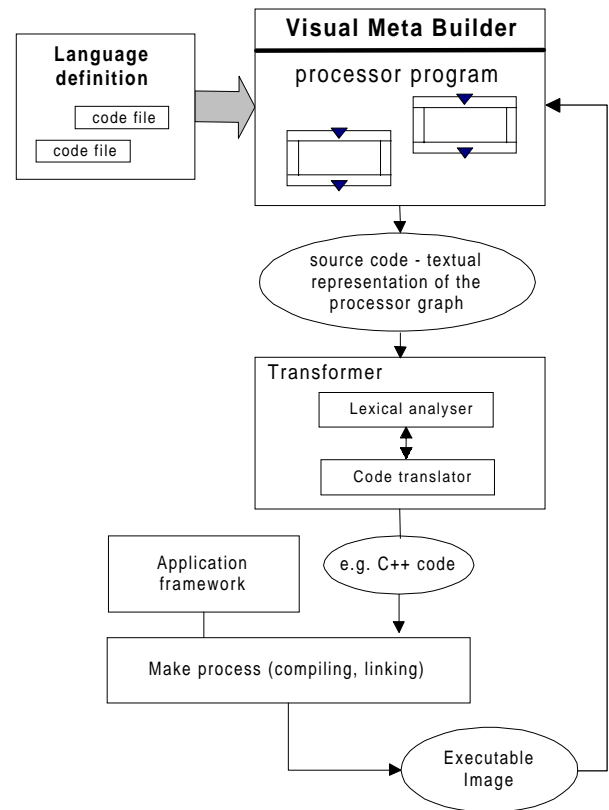
In general, there are several ways in which a visual language is executed or transformed in order to run an application. Firstly, visual symbols may directly be interpreted and executed (-> visual interpreter). Secondly, a visual program may be directly compiled into an executable image (-> image animation, “visual” compiling, direct

computing). Thirdly, a visual program may be transformed into an intermediate format which, in turn, is analysed and executed (-> indirect or language independent computing). Lastly, a technique could be applied so that code of a (semantically equal and known) textual language program is generated and “traditional” tools may be used to compile, link and run the application (-> language transformation).

As a graph is considered for the visual representation of a program it would be possible, for instance, to make use of “triple graph grammars” [SCHÜRR-1994], which are used to specify “rather complex **graph-to-graph translations** as languages of graph triples”. In this case, a processor graph and a graph based on the syntax and semantics of the underlying textual language together with their graph interrelationship would be defined. An incremental *graph parser* would then be needed for the execution of context-sensitive productions.

Another technique is supported by the tool shown in Figure 1. The processor nodes are defined by individual *code files* and transformed into their textual representation by scanning the graph and using their node description. An external transformer  $\tau$  is then necessary to generate the equivalent textual code (e.g. C++ code). Using this technique, the tool may be used as a general-purpose visual language system and may then be regarded as an alternative to Vampire. However, it should be noted that the tool presented here supports programming-in-the-small rather than program-ming-in-the-large. To enable programming-in-the-large several features such as the creation of packages, the use of libraries and more enhanced navigation techniques would be required.

One key aspect in designing a meta tool is that of domain-independence. To achieve this, a special node description language (here called a *node modelling language*) was developed. Both the syntax and the semantics of a processor are defined in a way such that different language types may be supported. This point is discussed in the next section, in which the theoretical underlying principles of the formal language are described.



**Figure 1:** The general architecture of the transformation process.

### 3.3 The underlying theory

Before continuing further, there will be a more formalised description of the applied techniques and principles. In general, when designing a new textual or visual language many different aspects such as orthogonality<sup>1</sup>, efficiency and program notation have to be considered. First there will be a description of the visual representation of a processor program, viz. by a graph. Then the underlying *node modelling language* will be presented. It is shown that any context-free language may be used to be translated into a visual programming language.

<sup>1</sup> Orthogonality means "that there should be not more than one way of expressing any action on the language" [MARCOTTY-1986].

### 3.3.1 Program representation by a graph

As mentioned earlier, a processor program is represented by a *3-dimensional graph*. A graph  $G$  in its basic form is a set of nodes  $N$  which are connected by a set of edges. As edges are used to describe relations  $R$  between nodes, a graph is defined as:

$$(1) \quad G = \{N, R\}$$

Graphs are used in virtually all branches of computer science. Thus, the authors judged it as a good starting point for the definition of a visual language. However, certain restrictions on the relation  $R$  are necessary to yield special classes of graphs supported by the *Visual Meta Builder*. Many different extensions of graphs are described in the literature, e.g. hypergraphs or higraphs in [HAREL-1988].

Several issues are important for the definition of the graph. To ease the users' interactions, data and parameters which are being processed may be presented in different visual ways and are directly linked to the nodes or edges on different layers  $l \in [1, max_l]$ . To make the coding flexible and to allow dynamic construction of new processors, a *node modelling language* has been specified. Like the hierarchy tool in VisaVis [POSWIG-1996], an instance is needed to keep track of the various defined processors and to observe the construction of well-defined processor programs.

A processor program  $V$  is defined as a 3-tuple:

$$(2) \quad V = (N, R, \alpha)$$

where  $N$  is a finite set of node elements  $\{n_0, \dots, n_m\}$  called nodes or processor nodes, and  $R$  the set of (attributed) relations  $\{r_0, \dots, r_n\}$  between any two nodes, defined as:

$$(3) \quad R \subseteq N \times N$$

The function  $\alpha$  assigns to a processor node  $n_i$  a set of processor attributes  $\alpha(n_i)$ .

As a processor program is *cycle free*, we restrict the possible graphs. Given the function  $r$  which connects a node  $n_i \in N$  to a node  $n_j \in N$ , we define  $r$  as  $r : N \rightarrow N$ . If  $r^0(n_i) = \{n_i\}$  and  $r^1(n_i) = \{n_j\}$  then  $r^{i+1} = r(r^i(n))$ .  $r^i$  may be interpreted as the

reachability of a node within  $i$  steps and is called *relational depth*. It follows:

$$(4) \quad r^+(n) = \bigcup_{i \in \{1, 2, \dots\}} r^i(n)$$

- with  $r$  restricted so that  $n \notin r^+(n)$

The topmost node of the graph is the root node. Only one root node may be specified within a processor program  $V$ . This node (denoted as  $n_{Root}$ ), which may only be placed once on a workplace, specifies important project parameters (e.g. project name and type). Given function  $\Omega$  returning the count of an existent processor type, and function  $\lambda$  returning the layer attribute value of a node, the graph definition is restricted accordingly:

$$(5) \quad G = \{N, R, \alpha\} \text{ so that } n_{Root} \in N,$$

$$\Omega(n_{Root}) = 1, \text{ and } \lambda(n_{Root}) = 0$$

- with 0 denoting the topmost layer

Given the above description of a graph, the question arises how a lexical unit (called a lexeme) may be represented as a node within the graph. A processor represents an action executed at a given time. Thus, a general transformation could be that of a function  $f_i$ . The advantage of using a function is that the intended action may be changed by using different function attributes, which can be called *specialisation*.

A textual language which is aimed at being visualised is traditionally described by a set of terminal symbols  $t_i$ , a set of non-terminal symbols  $u_i$ , a set of production rules  $p_i$ , and a start symbol  $s$ . A language is then described in BNF notation by the following (simplified) equations [cf. WIRTH-1996, p 7]:

$$\begin{aligned} \text{syntax} & ::= \text{production syntax} / \emptyset. \\ \text{production} & ::= \text{identifier} "=" \text{expression} "." \\ \text{expression} & ::= \text{term} / \text{expression} "|" \text{term}. \\ \text{term} & ::= \text{factor} / \text{term factor}. \\ \text{factor} & ::= \text{identifier}. \end{aligned}$$

In the presented approach, both terminals and non-terminals may be described by means of processor nodes. In its simplest form, a node may represent one or more terminal symbols or one non-terminal symbol from one equation. To allow the direct transformation of textual languages written in

(E)BNF into equivalent visual languages, non-terminals may be used as so-called block nodes. Block nodes may be refined by other (terminal and/or non-terminal) nodes placed on another layer of the graph. Hereby, the transformation is not strictly fixed, which gives the language designer the freedom to group the nodes according to given needs. Next, a simple example is presented to highlight the basic idea.

Given the following set of equations P

(p<sub>1</sub>) *AnyTextualLanguage* ::= *NonTermA NonTermB*.  
 (p<sub>1</sub>) *NonTermA* ::= *termA1 termA2 termA3*.  
 (p<sub>3</sub>) *NonTermB* ::= *termB1 NonTermC termB2*.  
 (p<sub>4</sub>) *NonTermC* ::= *termC*.

The equivalent equations P' of the visual language could look like this (version 1):

(p<sub>1</sub>') *AnyVisualLanguage* ::= *Root-Node*.  
 (p<sub>2</sub>') *Root-Node* ::= *A-Node B-Node*.  
 (p<sub>3</sub>') *A-Node* ::= *a-node*.  
 (p<sub>4</sub>') *B-Node* ::= *b-node*.  
 (p<sub>5</sub>') *C-Node* ::= *c-node*.  
 (p<sub>6</sub>') *a-node* ::= *F<sub>A</sub>(attr\_a1 attr\_a2 attr\_a3)*.  
 (p<sub>7</sub>') *b-node* ::= *F<sub>B</sub>(attr\_b1) C-Node F<sub>B</sub>(attr\_b2)*.  
 (p<sub>8</sub>') *c-node* ::= *F<sub>C</sub>(attr\_c)*.

The right side in equation p<sub>7</sub>' is semantically equivalent to “F<sub>B</sub>(attr\_b1 attr\_b2) C-Node” as the order of attributes within a node is irrelevant. Another transformation could look like this (version 2):

(p<sub>1</sub>') *AnyVisualLanguage* = *Root-Node*.  
 (p<sub>2</sub>') *Root-Node* = *A-Node B-Node*.  
 (p<sub>3</sub>') *A-Node* = *a-node*.  
 (p<sub>4</sub>') *B-Node* = *b-node*.  
 (p<sub>5</sub>') *a-node* = *F<sub>A</sub>(attr\_a1 attr\_a2 attr\_a3)*.  
 (p<sub>6</sub>') *b-node* = *F<sub>B</sub>(attr\_b1 attr\_b2 F<sub>C</sub>(attr\_c))*.

In version 1 of the example, a well-formed sentence of the visual language is “F<sub>A</sub>(attr\_a1 attr\_a2 attr\_a3) F<sub>B</sub>(attr\_b1 attr\_b2) F<sub>C</sub>(attr\_c)”. In version 2 it is “F<sub>A</sub>(attr\_a1 attr\_a2 attr\_a3) F<sub>B</sub>(attr\_b1 attr\_b2 F<sub>C</sub>(attr\_c))”. Both sentences are semantically equivalent. In the latter case, the resulting b-node is more complex, with the benefit that the number of nodes is limited. The relations between the nodes expresses the neighbourhood relation. Thus, when connecting two nodes, the semantic link represents a “followed by” relation.

Using the described method, a visual language may be designed which consists of ready-to-use components. If a valid sentence of the language is totally constructed by composition, the visual language may be characterised as a visual language with black-box nodes. If a well-formed sentence may not be totally constructed by composition, but uses specialisation, it is a visual language with white-box nodes.

### 3.3.2 Implementation and representation of a graph node

After defining the graph and the linkage to textual languages in a more formal way, it is necessary to describe how a node and the relationships between the nodes may be implemented and, more interestingly, visually represented. There are certainly many different ideas on design and implementation. However, defining and specifying a “nice looking” graph is one thing, implementing it is another. This section describes the layout implemented in a working prototype system.

As described already, a node is represented by a processor symbol and defined by a *code file*. In general, a node may be regarded as an abstract object definition of data and/or code. Another view would be that it is analogous to a non-terminal or terminal in a formal language definition as described in the previous section. By “applying” a processor, a production rule is carried out and the terminals, linked to the processor, are used to define the processor template code. In any case, a processor node is meant to represent a part of a well-formed language construct with its inherited syntax and semantics. The reader should, however, be aware that the actual meaning of the language construct is less important than how it may be parameterised and placed at the right location of the application framework within the transformation process.

In order to follow an open concept, the *node modelling language* contains many different options. Table 1 gives a sketch of the most important entries defined for each lexeme. It clearly shows the separation of the semantics from its visual representation.

lexeme	domain restrictions/ constraints
title = ident	Title of an object node.
bitmap = filename	Visual representation of a node <i>bmp file</i> .
bitmap_err = filename	Visual representation when syntax error.
pin_descr = <count> ", " { x,y }	General slot definition of a processor element.
help = filename	Reference to help file.
lexeme = ident	Lexeme, syntactical name.
group = ident	Group to which lexeme belongs.
next_lexeme = ident { , ident }	Possible successor nodes. These entries refer to traditional adjacency lists.
layer = (((">"   "<") number   number { "," number }   "any")	Layer information - with number in [0, max <sub>layer</sub> ]. Using the layer attribute it is possible to define a root processor or to specify other restrictions. Layer 0 is the topmost layer.
max_count = number	Specification of how often an element may be placed on the workplace.
error_msg = error_text	The message which should appear if an error is detected.
block = ("yes"   "no")	Restriction whether element may be "sub-levelled".
type = {"atomic", ...}	Type of an element. Intended for future use if other types of nodes (e.g. composite nodes) are necessary.
count = ident	Number of slots in total.
slot<i> = slot_direction, slot_type, spec_option	Specification of slot attributes analysed at run-time.
slot_direction = ("in"   "out"   "inout") slot_type = ("event"   "data"   ...) spec_option = ("must", "optional")	
connection<i> = (type1 "->" type2   "nil")	Specification of possible (typed) connections.

**Table 1:** The basic node modelling language.

Besides the node modelling language, further features may be implemented. For example, to allow blocking, a group of processors may be set into a processor folder (similar to a folder editor). This is a logical grouping not specified by the underlying syntax.

In general, it should be possible for an experienced user to change the pictorial symbol of the processor. This is achieved by introducing the

*bitmap* attribute which contains a reference to a bitmap file.

As for other development tools, a syntax checker is required. This is achieved by a component called *graph checker* which evaluates the nodes at run-time. The representation of a visible processor should indicate whether the processor is fully specified (by all *must* parameters) and syntactically correct. This could be implemented by an alternative processor node (see *bitmap\_err*) or by other visual means (e.g. use of colour). Thus, a special error element may be defined. Further, visual feedback is needed to inform the user that the program may be generated and executed.

## 4 A sample program definition

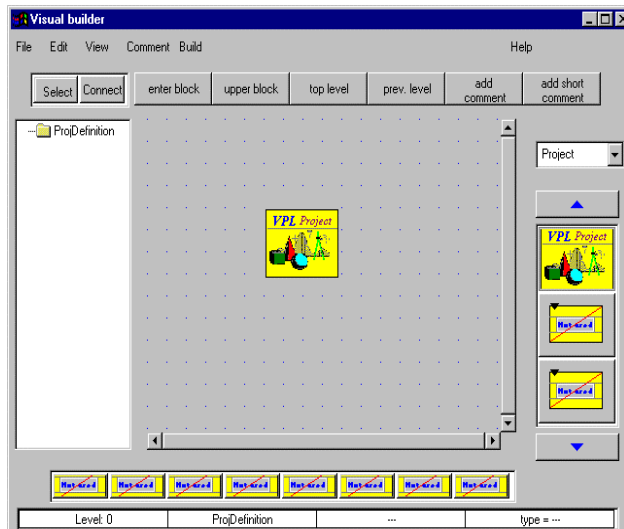
### 4.1 Defining the visual syntax

To prove the concept, a small visual language (called VPL-1) was designed and a working prototype system was developed to show the integration of an underlying language and its transformation. This is common to many other approaches (see for example [DIJKSTRA-1976]). VPL-1 contains some basic definitions of an imperative/procedural language like C (see [KERNIGHAN-1983]) integrated within a visual program environment. For a different approach see [GLINERT-1990b].

The idea behind this is that the authors follow (to some degree) the process which has taken place in the textual programming field. Thus, comparisons may be made more directly and the benefits which are hoped for are more evident.

### 4.2 The current prototype system

On the following pages, hard copies of the current prototype system provide a first impression of the proposed system. The system is written in C++ and makes intensive use of Ilog Views 2.4, a special class library. The external transformer is written in C and uses a recursive descent parsing algorithm as described in [WIRTH-1996]. The first step in designing an application is to place a *ProjectDefinition node* on the workplace (see Figure 2).



**Figure 2:** VPL-1: Defining the project.

By clicking the right mouse button over the processor node, various project parameters may be specified. (This is not shown here.)

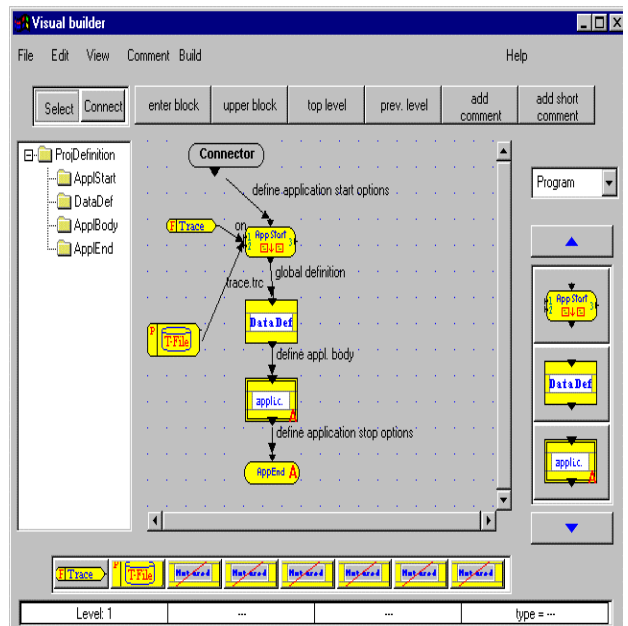
The next step is to define the application's main structure. This is achieved by selecting the project node and clicking on the button labelled "enter block". If there is no block already linked to the node, the user has to confirm the creation of a new layer. This is necessary as the current system doesn't maintain a *bit string list* to keep track of free/used layers.

If the user confirms the creation of a new block, a *connector element* is automatically placed on the new layer. This is done to simplify the coding and administration of the connections between different layers. This connector element may not be deleted! Next, the application has to be defined. From the processors' category list the entry "Program" has to be selected. An application, in its simplest form, consists of an *ApplicationStart* node, a *DataDefinition* node, an *ApplicationBody* node, and an *ApplicationEnd* node (see Figure 3).

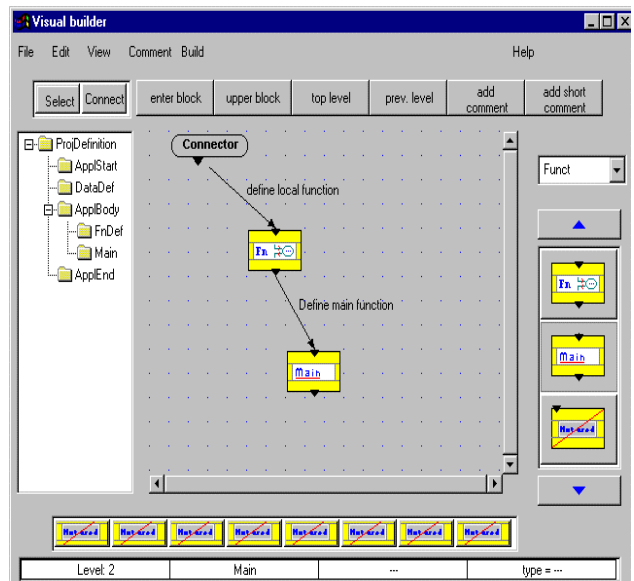
Next, the category "Funct" is chosen. By selecting the *ApplicationBody* node and creating a new layer, the application may be coded by adding further nodes (see Figure 4). Finally, data and command nodes may be placed on the workplace (see Figure 5).

After defining the application, the executable program is built by selecting the *ProjectDefinition*

node through the tree gadget and choosing the command "Generate" from the build menu. Finally, the generated program is started by clicking the "Execute" command from the build menu.



**Figure 3:** Defining the application's structure.



**Figure 4:** Defining a function and the main entry.

