Seamless Integration of Control Flow and Data Flow in a Visual Language

Honitriniela Randriamparany  
Computer Science Department  
University of Geneva  
rue Général Dufour 24  
CH-1211 Geneva 4, Switzerland  
Honitriniela.Randriamparany@cui.unige.ch

Bertrand Ibrahim  
Computer Science Department  
University of Geneva  
rue Général Dufour 24  
CH-1211 Geneva 4, Switzerland  
Bertrand.Ibrahim@cui.unige.ch

Abstract

In the visual programming domain, the stress of research is laid on the use of visual formalism, which is considered to be more intuitive than the textual formalism, in the programming task. Some visual languages are based purely on data flow. With such languages, the execution order depends on the availability of data and it is therefore difficult to specify some programming constructs such as control structures. On the other hand, a pure control-flow based language has shortcomings with respect to data processing.

Many visual languages based on the data-flow paradigm are supplemented with control structures to specify repetitive behavior in programs. In our case, the visual language initially used the control-flow paradigm. We then enriched this language with the integration of data-flow. This article presents the advantages of this integration. Then, it explains (mostly from the visual formalism point of view) how we decided to handle the combination of the two paradigms in our language, named DIVA-cd. The article presents through examples the specifications of various control structures in the DIVA-cd language.

1. Introduction

A large number of visual programming languages use graphs to represent programs or modules. The semantics of the arcs which connect the nodes of a graph depends on the paradigm used by the language. With the data-flow paradigm, arcs convey data to be manipulated. With a language based on the control-flow paradigm, arcs convey control signals which activate the execution of the destination nodes.

Imperative, or control-flow, languages were the first programming languages, and were based on the von Neumann model. They define computation in terms of a sequence of discrete operations. The programmer defines the order in which operations will take place. Control-flow can sometimes be represented graphically, for instance in the form of flowcharts [3][13] or Petri Nets [11]. Control-flow graphs consist then of nodes linked together by arcs (or edges). Nodes represent tests and actions (sometimes using data internally) and arcs represent the flow of control between nodes. Control-flow visual languages neglect the data structures. Data is represented textually, usually as variables within nodes, rather than by any graphical symbol.

Data-flow is both a programming concept and an implementation technique. In software engineering, data-flow hints at the flow of information between entities (functions) which process data. In the data-flow model, a program is represented by a directed graph where nodes represent functions and where arcs represent the flow of data between functions. The execution of a node occurs when all of its operands are available, so that the functions are executed in an order resulting only from the data dependencies.

Admittedly, both paradigms have their own merits. But if we combined their advantages together, we could create a more powerful system. A pure data-flow language needs programming constructs to treat complex problems. A data-flow visual language is considered powerful if it provides extra constructs and predefined functions. On the other hand, a control-flow visual language needs new formalisms to be able to directly address data by representing the flow of data between nodes.

A common goal is to attach importance to data in the processing and, at the same time, to offer more flexibility in the specification of the control structures (selective routing of data, iteration, recursion, ...). It has frequently been the case that people start from a purely data-flow language and add to it complex control structures, very often a different construct form for each control structure. Several similar works have been made in this direction (see section 4).
In our case, we started from a simple language which was originally based exclusively on the control-flow paradigm. We integrated the data-flow paradigm into it. In our language, we can thus consider the data in the execution processing and we can easily specify, without additional constructs, the various control structures (see section 3).

2. The DIVA System

The DIVA ("Développement Interactif et Visuel d’Application") system is a software development environment based on a semi-formal visual specification language, the DIVA-cd language (the "cd" stands for "control+data"). It aims at bridging the gap between the "clients" of an application and its developers. DIVA has evolved from an older system named IDEAL [5][6] that was intended for the development of computer-based learning material and that only supported a control-flow formalism. DIVA data-flow description capabilities adds to this initial environment. To support data-flow, we introduced data description capabilities based on strong typing.

### 2.1. Visual formalism

Program specification in our formalism is represented by a kind of typed directed graph. The nodes of the graph usually contain actions (activity nodes) and the arcs represent the flow of control (or data) of the program. The graphical decoration of an activity node indicates the type of action that is described in the node (see Figure 1)

#### 2.1.1. Activity nodes

A detailed description of the activity nodes initially available is given in [8]. These nodes are:

- **message node**: which contains text that has to be shown to the user.
- **documentation (or comment) node**: it does not represent any action. The content is there just to enhance the readability of the graph.

- **directive node**: it contains some text describing an action that has to be done by the program.
- **call (or subgraph reference) node**: it names another graph that should be invoked there.
- **test (or condition) node**: it contains a predicate. A counter is implicitly associated with each predicate and is incremented at runtime each time the predicate evaluates to true. Test nodes are usually grouped vertically in adjacent and connected boxes (cascade of tests) that are evaluated in sequence from top to bottom until one of the predicates evaluates to true, in which case the evaluation stops. Arcs originating from tests can be labeled with a number or an interval of numbers. The labels determine the order in which arcs will be used if tests evaluate to true more than once: at run-time, a counter is associated with each test node and the flow of control is determined based on the value of the counter and the labels attached to the node’s outgoing arcs; the counter is incremented each time the node’s predicate evaluates to true and a control token is sent through one of the control-flow edges leaving that test node.

Two other activity node types were added to deal with control parallelism [9]. They are:

- **guard node**: like a test node, it contains a predicate. Its specificity is that it allows parallel processing of its successors. It is different from other activity nodes since, to be activated, it requires that all its predecessors finish their execution. It thus acts as a synchronization mechanism.
- **spawn node**: it names another graph that should be invoked in parallel with the calling one. The parameters of the called graph are then considered as communication channels between the two graphs.

The graphical representation of all these activity nodes is given in Figure 1. The activity nodes were initially designed for the control-flow model. For the data-flow integration, we added, in our formalism, data nodes and additional decorations (data terminals) on the activity nodes to specify the data to be used. Since control and data flows are in the same diagram, activity nodes are connected with two kinds of arcs. Control tokens have no internal representation and travel along control-flow arcs, while data tokens have a type describing their structure, and travel along data-flow arcs.

#### 2.1.2. Data nodes

Data nodes are meant to represent data containers with some specific run-time behavior (store and forward). Data nodes must have activity nodes as predecessors and successors. They cannot be directly connected to other data nodes. We will define here the data node types: bag nodes, queue nodes and datastore nodes (see Figure 1).

A **bag node** operates as a homogeneous container for either control or data tokens. A control-token bag will be rep-
2.1.3. Data terminals. Data terminals represent node parameters in the data-flow view. There is a separate node decoration for each node parameter. To each data terminal corresponds a data type and possibly a data manipulation. In [7], we explain how data types and data manipulations are represented graphically. Data manipulation, at input terminals as well as at output terminals, is reflected in the arc shape (see Figure 2).

A node input parameter is represented by an input terminal, and an output parameter by an output terminal. Various node behaviors could be obtained according to the requirement in relation to the input parameters. For this, we distinguish between two kinds of input terminals attached to nodes:

- **A triggering input terminal:** it is an input terminal for which a new data token is required to trigger the execution of the node. If an initial value is defined, it will be used as an initial data token that doesn’t need to be produced by another node. The use of the token consumes the token, and a new one is needed before the node execution can be triggered again.

- **A non-triggering input terminal:** it is an input terminal the content of which is available at all times. It must therefore have a default initial value. Whenever a new token arrives, its value simply replaces the current value.

The graphical representation of these three kinds of data terminals (triggering input terminal, non-triggering input terminal, and output terminal) is given in Figure 1. A node’s output terminal could be connected directly to one of its input terminals. The next activation of this node will thus use the new value produced by the last activation of the node.

Graphs can have parameters (input and output parameters). Decorations for these parameters are carried by the header node (graph beginning node). Like node parameters, graph parameters are represented by data terminals. There must be a match between the graph parameters and the parameters of any call node or spawn node that refers to this graph.

![Figure 2. Representation, on arcs, of data manipulation (selection and composition)](image-url)

2.1.4. Control-flow and data-flow views. There are two separate viewing modes in the DIVA system: the control-flow mode and the data-flow mode. In both modes, all nodes will be showing. In the control-flow mode however, only control-flow arcs are shown and connect two nodes directly, and in the data-flow mode, only data-flow arcs are shown and connect an output terminal of a node to an input terminal of another node. The user is allowed to switch between these two modes at any time and the nodes remain in the same place when the mode switch takes place. We could have chosen to show both kinds of arcs simultaneously, but we feared display overload.

In the control-flow mode, a little graphical decoration (we call it a parameter handle) is added to every node that
has input and/or output parameters. By clicking on these decorations, the user will be able to see, in a pop-up window, details about the parameter types. So, it is possible to examine the parameters of a node while in the control-flow view. Parameter handles are represented by a shape. There will be one parameter handle for the input parameter list and another parameter handle for the output parameter list. The first kind will point towards the center of the node, and the second will point towards the outside of the node. The parameter handles for the graph input and/or output parameter lists are attached to the header node (see Figure 3). Even if data nodes are only useful in the data-flow view, they are displayed in the control-flow view to avoid node overlap during node creation.

Figure 3. Example of a header node in the control-flow view

In the data-flow view, there is a separate node decoration for each node/graph parameter. The data-flow view allows the user to create data paths between different nodes that have been created in the control-flow view or inserted while in the data-flow view. In the latter case, the insertion point can be placed either over an empty area, in which case the new node will not be initially connected to any other node, or placed over a node that has output parameters. It cannot be placed over a node that does not have at least one output parameter, as it would not make sense to talk about data-flow if there is nothing coming out of the source node.

In our formalism, the data flow is mainly data-driven, especially for the activity nodes. The execution of a node is carried out as soon as all the data that it needs is available. Behavior that is similar to the demand-driven data-flow is carried out as soon as all the data that it needs is available.

2.2. Execution model

We initially started by assuming that the designers would first specify the control flow of the application, and that data flow would be added at a later stage. But there are cases where the specification is data-driven and there is no point in explicitly specifying a control flow. To cope with the whole spectrum of applications, from pure control flow to pure data flow, we have decided to remove the constraint on having the control-flow graph be completely connected. Instead, wherever control-flow arcs exist between two nodes, the code generator will ensure that the two nodes get executed in sequence, but if only data-flow arcs connect two nodes or group of nodes, it is the availability of data that will guide the execution sequence.

The execution model can be defined as follows. If there is one or more control-flow arcs, then a control-flow token is required for the execution of the node to start. If the node has one or more triggering terminals, then a new data-flow token is required on each of them. Since non-triggering terminals always have a value available, they are not involved in triggering the execution of the node, unless the node has no control-flow predecessor and has no triggering terminal (i.e. the node has only data-flow predecessors connected to non-triggering terminals). If a node has only non-triggering terminals, and no control-flow predecessor, its execution can start anytime the graph, in which the node is defined, is invoked. After it has been executed once, it will be re-executed anytime one of its inputs changes, i.e. anytime a new token arrives on one of its (non-triggering) input terminals. This mechanism allows one to build reactive systems, i.e. systems for which the outputs are recomputed each time one of the inputs changes.

3. Control structures in DIVA-cd

The specification of most branching and looping constructs found in common programming languages, using the DIVA control-flow based language, has been presented in [8]. These constructs use the test node type described in section 2, in conjunction with the definition of a predefined predicate named "true" that always evaluates to true (synonyms like "otherwise" are also accepted). This section shows the use of these constructs with the combined data-flow/control-flow paradigms of the DIVA-cd language. We will illustrate the "for" and "while" iterations by specifying the factorial calculation and the highest common factor. The dot product calculation of two vectors illustrates the "foreach" iteration. We want to emphasize that we do not use additional constructs for these specifications.

3.1. The "For" iteration

The first version of the factorial calculation uses the "for" loop. Using a pseudo-syntax close to Ada and Pascal, we can write a pseudo-code for this calculation, in the following way (n is the function parameter):

```plaintext
Fact := 1;
for i := 1 to n
do Fact := Fact *i;
```

With DIVA-cd, a test node with the predefined predicate true is used (Figure 4). This node has two outgoing arcs. The first, labeled 1..n, is used when the counter associated to the test node is between 1 and n (included). The second arc is followed when the counter is equal to n+1, and the processing ends because the destination node does
not have any successor.

Figure 4. Factorial calculation using a for loop

At the beginning of program execution, the graph input parameter value is copied to the test node input parameter n. The test node is evaluated each time a control token arrives (the first token comes from the header node). Since the predicate is always true, the control-flow arc labeled 1..n will be followed if the counter is lower or equal to n, i.e. a control token is sent along this arc. At the same time, the counter value is sent to the calculation node (the bottom-right one). This node has two input parameters: one, i, is represented by a triggering terminal and the other, Fact, by a non-triggering terminal. The node will thus be executed each time both a new control token and a new data token (on the triggering terminal) have arrived. Since the input parameter Fact is represented by a non-triggering terminal, it has a default value (which is put between brackets beside the parameter name). After each iteration, the output parameter value of this node is sent, at the same time, to its input parameter and to the graph output parameter. When the program finishes, the graph output parameter already contains the last computed value.

3.2. The "While" iteration

Using the While iteration to specify the factorial calculation, we can have the following pseudo-code (N is the function parameter):

\[
\begin{align*}
\text{Fact} &:= 1; \\
\text{while } N > 0 & \text{ do begin} \\
\text{Fact} &:= \text{Fact} \times N; \\
N &:= N - 1; \\
\text{end};
\end{align*}
\]

With DIVA-cd, two cascading test nodes are used. The first contains the condition for the execution of the action and the second contains its complement. For the factorial calculation (Figure 5), the first test node checks if the value of N is positive. The two activity nodes, one for the calculation of Fact and the other for the decrementation of the value of N, are in the control-flow loop. The data-flow indicates the data flow between these nodes.

Figure 5. Factorial calculation using while iteration

At the beginning of the processing, the value of parameter N is sent to three input parameters (one for the first test node, one for the Fact calculation node and one for the node to decrement N). The first action carried out will be the first test node evaluation since it receives the first control token coming from the header node. As long as N remains positive, the outgoing control-flow arc of this node is followed, which makes the loop possible. The two activity nodes are carried out each time a new control token arrives and the data terminal representing the parameter N receives a new data token. When N becomes null, the second test node will be executed. Since its predicate is always equal to true, its control-flow successor should be executed, but since there is none, this implies the end of the program. At this moment, the graph output parameter already contains the last computed value.

If we compare the two versions of the factorial calculation, we can deduce that the "for" iteration is simpler than the "while". But the "while" iteration would be appropriate for other situations, as in the HCF (Highest Common Factor) calculation.

One possible solution for HCF is as follows. Suppose A and B are the numbers, A is greater than B and their HCF is F. Then A-B and B will also have an HCF of F. If we use this fact, repeatedly replacing the greater of the two numbers by their difference, until the two numbers are equal, then this number will be the HCF. In the pseudo-code, this is equivalent to

\[
\begin{align*}
\text{while } A \neq B & \text{ do} \\
\text{if } A > B & \text{ then } A := A - B \\
\text{else } B & := B - A; \\
\text{return } A;
\end{align*}
\]
As for the factorial calculation, a group of cascading test nodes is used to evaluate the loop condition (Figure 6). A second group of cascading test nodes is also necessary to compare the values of A and of B, in each iteration. For this example the iteration is mainly managed by the data flow. We emphasize that the nodes within a cascade of test nodes are connected to each other by “hidden” control-flow arcs in such a way that whenever a test node evaluates to false, the test node just underneath it gets a control token.

Figure 6. Highest Common Factor calculation using While iteration

The terminals associated with the two input parameters of the first cascading test node group are non-triggering because we want the evaluation of the top test node to be carried out each time a new data token arrives on one of them. For the other nodes, we require that a new data token arrives on each input terminal to be able to activate the corresponding node. This leads us to use triggering input terminals.

At the beginning, the predicate \( A<>B \) of the first group of cascading test nodes is evaluated with the initial values of A and of B. Afterwards, it is evaluated each time one of its terminals receives a new value. If this evaluation gives true, two data tokens (respectively representing A and B) are sent to the second group that checks which of the two is greater than the other. If A is greater, the difference A-B is calculated and the result is sent to input terminal A of the first group of cascading test nodes. Otherwise, the difference B-A is calculated and the result is sent to the input terminal B. When the two values are equal, the value of A is transferred to the output parameter of the graph and the execution stops.

### 3.3. The "Foreach" iteration

In DIVA-cd, the “foreach” iteration is a by product of the semantics given to an arc which starts from a data output terminal with an associated array type, towards an input terminal that has the base type of that array associated with it. If there is only one processor, the destination node of this arc is executed as many times as the number of array elements, otherwise, a certain number of processes executing this node are carried out in parallel. This situation is called node replication in [9] (multiple execution of the concerned node).

Figure 7. Dot product calculation: (a) data-flow and control-flow, (b) graph / node parameter lists

The dot product calculation of two vectors illustrates this kind of iteration (see Figure 7). There are two data-flow arcs that have data (A and B), the type of which is array of integer, at their starting points and data (I1 and I2), the type of which is integer, at their ending points. The replicated node (which is the destination of these two arcs) calculates the product of two integers corresponding to an element of A and an element of B. Thus, its input parameters are of the integer type.

As one can see from these graphs (Figure 4, Figure 5, Figure 6 and Figure 7), the DIVA-cd visual language does not need any new element for the specification of the various control structures. All its expressive power lies in the semantics attached to the nodes, the arcs and the data terminals.

### 4. Related work

Several developments have been made in the visual programming domain to add various control structures to a data-flow language. These extensions can exhibit various degrees of complexity.

For the V visual programming language, which is a lan-
guage with a visual representation of dependencies between data and processes. Auguston and Delgado suggest a solution for iterative processing based on the notion of a conditional data flow switch, and a specialized iterative construct based on pattern matching for vectors and matrices [1].

To tackle sequential iterations in the VIPERS language [2], the presence of cycles within the program graph is admitted. VIPERS uses compact forms simulating loop behaviors (FOR block, FOREACH block and WHILE block).

A survey of 15 visual programming languages (including Prograph [12] and Cantata [10]) using the data-flow computational model, is given in [4]. According to this survey, some data-flow visual languages provide iteration in the form of cycles in the data-flow graph, and others use special control-flow constructs. The problem with the addition of special constructs is that if other control forms prove useful, new constructs should be created. In the DIVA-cd language, since the control-flow and data-flow paradigms are complementary, we do not need particular constructs.

5. Future work and conclusion

As explained in section 2, in the current implementation of our visual language, the control-flow view and the data-flow view are not visible simultaneously. This choice to separate the two views was taken to keep the graph more readable. We are still considering combining the two flows in a single view, using visual clues to differentiate between the control flow and the data flow.

The integration of the data-flow into the DIVA system allows the specification of data parallelism. It will be based on the semantics of the data-flow arcs [9]. We continue our work to further this research.

This article gave the visual formalism which we adopted in our DIVA-cd visual programming language to combine the control-flow and data-flow paradigms. This combination is simple and seamless. The programmer can work either on the data flow or on the processing sequence, and can switch between the two without any problem.

6. References


