# A Graph Rewriting Programming Language for Graph Drawing

P.J. Rodgers
Computing Laboratory, University of Kent, U.K. email: P.J.Rodgers@ukc.ac.uk

**Abstract**

*This paper describes Grrr, a prototype visual graph drawing tool. Previously there were no visual languages for programming graph drawing algorithms despite the inherently visual nature of the process. The languages which gave a diagrammatic view of graphs were not computationally complete and so could not be used to implement complex graph drawing algorithms. Hence current graph drawing tools are all text based.*

*Recent developments in graph rewriting systems have produced computationally complete languages which give a visual view of graphs both whilst programming and during execution. Grrr, based on the Spider system, is a general purpose graph rewriting programming language which has now been extended in order to demonstrate the feasibility of visual graph drawing.*

## 1. Introduction

The use of graphs extends throughout computing. Examples of application areas are neural networks; database modelling, software engineering, communication networks, and microprocessor design. Graphs are widely used because they enable complex interactions to be expressed visually. However, the visual representations quickly become unusable unless they are presented in a comprehensible manner. This has motivated the ongoing research in graph drawing algorithms, which attempt to automatically display graphs according to some aesthetic criteria.

The purpose of graph drawing is to give a comprehensible view of complex information by producing a visual layout of it in graph form. Hence, when programming graph drawing algorithms it would make sense to use a visual representation of the graph being manipulated. However, the current graph drawing tools use textual representations of graphs.

This paper describes Grrr, a visual graph rewriting language for graph drawing. The principle advantage of the system is to allow programmers to be closer to the final representation of the graph they are manipulating. Allowing an intuitive representation of data within a program means graph drawing algorithms can be coded more quickly and efficiently, and should encourage the development of novel algorithms.

There are a large variety of graph drawing algorithms, for various types of graph, such as hierarchical, planar and general graphs [2]. Graph drawing algorithms attempt to improve the appearance of graphs. A good drawing depends on the type of graph and the application domain. There are various desirable aesthetic criteria including: minimising the number of arc crossings, minimising the area the graph uses, or maximising the symmetry of the graph. Most graph drawing algorithms produce a final result that is a compromise on several aesthetic criteria.

There are many tools for prototyping graph drawing algorithms, discussed in Section 2.1. They all rely on some form of textual representation for the graph and use a textual language for expressing graph drawing algorithms. This clearly is at odds with the basic motivation for graph drawing, which is visualising complex data in a graphical manner. Textual representations fail to give an indication of the interconnected nature of graphs and can force undesired structure on them, which makes programming graph drawing algorithms more complex.

Graph drawing is an algorithmically complex task, and requires computational completeness. Computational completeness means that a programming language can be used to encode any algorithm encodable by a computer language. As graph drawing algorithms often make use of complex iteration techniques, it is important that any language designed as a general graph drawing language should be computationally complete.

Until recent developments in graph rewriting languages there was no visual language that both visually represented graph based data structures and was computationally complete. DOODLE [5] is a visual system for graph drawing, but it does not allow algorithms to be encoded, instead it is designed for specifying declarative constraints on existing algorithms. Other visual languages such as G+ [6] and VQL [14] have a diagrammatic representation of data graphs and use a notion of transitive closure, which allows simple recursion, but does not allow algorithms which need complex recursion to be encoded (such as the example algorithm given in Section 4). There are computationally complete dataflow visual languages [8], which use graphs to represent programs, but do not allow a visual representation of graph data structures.

Graph rewriting programming languages have been developed over the last decade. Based on graph grammars, these systems attempt to program by rewriting a host graph with user defined transformations. They share only the process of graph rewriting and that they are computationally complete visual languages. These languages differ widely in programming paradigm and application area. They are discussed in more detail in Section 2.2.

Graph rewriting languages often use graph grammars as a theoretical base, and graph grammars for graph drawing, have been proposed [3, 21]. These are specialised to certain sorts of graph (trees or DAGs) and allow a limited class of algorithm to be applied. This contrasts with programming languages for graph drawing which are often designed to work with general graphs and allow a large number of different algorithms to be specified.

Section 3 discusses Grrr and the changes made to it for graph drawing. The main alteration is the integration of geometric operators. These allow the programmer to query and change the position of single nodes and arcs in the graph, or to work on groups of primitives. The operations that are now built into Grrr and the issues concerning their implementation are discussed in Section 3.2. The syntax and semantics of Grrr that make it a practical programming language are discussed in Section 3.1. Also discussed are the features that make it a suitable system for producing complex algorithms, such as graph drawing algorithms.

An example of use, detailing the implementation of a graph drawing algorithm for hierarchical graphs is given in Section 4. Section 5 contains the conclusions and discusses possible further work to enhance this prototype system.

## 2. Background

### 2.1. Graph drawing tools

Graph drawing tools allow users to display specified graphs by programming their own graph drawing algorithms or accessing already coded algorithms. They are designed to allow algorithms to be expressed as easily as possible in textual languages, such as Java or C.

The manner in which the graph is represented as a data structure influences the ease in which certain types of graph can manipulated. When working with hierarchical graphs a term based representation can be used, such as that employed by daVinci [7], however this representation method has an implicit ordering of the nodes in the graph and so makes cyclic graphs hard to manipulate. Many other systems use a variant of the GML method [10] which has two discrete sets, one for the nodes and another for the arcs. The arcs have references to the source node and the destination node. This allows more general graphs to be manipulated, but any connection between nodes is difficult to derive by inspection of the graph representation. Another method is the adjacency method, where the arcs are specified by the relationship between node definitions. This allows the arcs to be placed in context, but because a node may be connected to any number of other nodes multiple redundant node definitions are required.

The most general graph drawing tools provide built in algorithms and user editing facilities as well as the capability to produce graph drawing algorithms. Graphlet [9] is one such publicly available tool. It uses the GML representation method. GDS [4] is a Java applet that supports graph editing and graph drawing algorithms. There are commercial systems such as Graphviz, a suite of graph drawing algorithms and graph editors, and GLT, which includes graph drawing and editing functionality, with incremental layout of dynamic graphs.

Other graph drawing tools have algorithm creation facilities but do not provide interactive graph editing. VGJ [13] is a Java based tool for graph layout and drawing without editing capabilities. DAWG [17], which uses a similar graph definition to GML, is a Tcl/Tk based display tool that allows graph editing facilities to be built onto it.

### 2.2. Graph rewriting programming languages

Graph rewriting programming languages compute by rewriting a host graph according to user defined transformations. A key advantage to this approach is the combination of computational completeness and visual view of both the graph being rewritten and the transformations that rewrite the graph. These are the only forms of visual language in which complex algorithms on graphs can be programmed.

Graph rewriting languages vary in several important aspects: the host graph that is to be rewritten may be any graph, or it may be restricted by disallowing duplicate nodes or arcs; there are several different methods for rewriting graphs given a set of transformations; and there are alternative ways that the user can specify the transformations. The consequence of this is that, although there are several systems that use graph rewriting, their programs work in very different ways.

GOOD [15] is a language for database programming. It rewrites a host graph that is restricted by not having duplicated node labels. This means that algebraic expressions are difficult to formulate, but allows GOOD to rewrite the graph in a parallel manner, that is, all the subgraphs of the host graph that match will be rewritten in one step (the alternative is to rewrite subgraphs in a serial manner, changing only one subgraph at a time). In GOOD, rewrites are initiated by special ‘trigger’ nodes in the host graph, which can be viewed as corresponding to function or procedure calls in textual programming languages. As with subgraph rewriting, all the trigger nodes in the graph are initiated in parallel. GOOD has a single graph method for specifying rewrites, so that all the graph items to be matched, deleted and added are placed on the same graph, but displayed differently (in bold, or with double outlines), which can lead to cluttered displays.

Progres [18] is designed for rapid prototyping of computer languages and environments [19]. The graphs rewritten are general. The host graph is rewritten serially in a non-deterministic backtracking manner, so that a target graph is specified, and where more than subgraph can be matched, one is chosen in a non-deterministic way. If the target graph can be found the execution ends, otherwise if it is not found then the system backtracks to the last alternate subgraph match. Progres does not have trigger nodes in the host graph, instead rewrites are initiated by a textual language. The rewrites contain two diagrams, a left hand side (LHS) for the graph to be matched, and a right hand side (RHS) which specifies the changes to be made to the host graph.

Visual Δ-grammar programming [11] was introduced as a method of writing concurrent programs. The method of specifying the productions is via a single diagram, where the items to be removed and added are given by their position in relation to a triangular structure. The system is visual, but the requirement to place added and deleted nodes and arcs in particular locations means that it is hard to maintain the appearance of the original data structure in the transformations. The graphs rewritten are general. Both trigger initiation and subgraph rewriting is in parallel, and to prevent cases where the rewritten subgraphs overlap, a conflict resolution algorithm is used. This is not given for a particular implementation, but it should select the maximum number of rewritable subgraphs. This leaves an important section of the programming paradigm unspecified.

Δ has interesting modifications to the rewriting process. Sections of the graph that are marked with a ‘Kleene *’ operator may match a number of times in the host graph. Several nodes may be marked with a ‘Fold’ operator. Such nodes may match with a single node in the host graph.

## 3. Grrr

Grrr is a development of the Spider graph rewriting programming language [16]. Spider is a prototype system for database programming. Modified, it forms the basis of Grrr, a general purpose programming language. The important changes are: true serial trigger initiation; removing any semantic differences between node and arc types; improvements to the algorithm animation facilities; and allowing new built in triggers to be easily integrated into the language, such as those for graph drawing, described in Section 3.2. Section 3.1 discusses the unmodified Grrr programming language.

### 3.1. Standard syntax and semantics

Grrr uses a two graph rewrite specification method integrated into a fully visual language. It has trigger nodes and a serial newest first trigger rewrite strategy. This combined with a serial subgraph rewriting technique means programs can be structured sensibly. It does not use a backtracking method for serial rewriting, rather it uses a deterministic subgraph matching strategy.

Grrr is deterministic in the method it uses for graph matching. This is achieved by a graph sorting method that takes into account the surrounding subgraph of each graph element. Once sorted, the graphs can be matched with a backtracking algorithm.

As with Spider, the triggers in Grrr are initiated newest first, which means they are applied serially. For example, Figure 4.1 shows the toplevel transformation ‘LayoutGraph’. The third rewrite calls ‘DoChildBCs’, shown in Figure 4.2. Now all the programming associated with ‘DoChildBCs’ will be completed before ‘LayoutGraph’ is called again. This means a nested, hierarchical structure can be given to programs. Under parallel rewriting, mutual recursion is required to enable one trigger to pass execution on to another, so the ‘Layout Graph’ trigger would have to be deleted at the same time as ‘DoChildBCs’ was initiated, and recreated again later as ‘DoChildBCs’ terminates.

Grrr has serial trigger initiation for triggers that are the same age. In this case the next trigger node to be initiated is determined by the ordering of the nodes. Previously, triggers in Spider were executed in parallel which could cause conflicts, forcing the restructuring of those programs.

There are other Grrr features that are designed to make it a more usable programming language: it has a LHS/RHS specification for the rewrites, and the rewrites within a transformation definition have a top-down order of matching, similar to that often used in textual functional and logical programming languages.

The data graph (that is the part of the host graph that holds application data) and the nodes and arcs that hold associated information (that is, information derived from the data graph and information concerning execution) can be separated using different node types. A node type specified in a rewrite will only match with that node type in the host graph, avoiding potential confusion. The example given in Section 4 prevents the nodes holding associated information about the data graph to be confused with nodes in the data graph itself by using circular node types for the data graph and oval node types for the associated information, such as the X and Y positions of the data nodes, or for tags to indicate whether data nodes have already been matched.

The types of primitive in the language is kept to a small number, but the addition of attractor nodes and negatives improves the expressive capabilities. Attractor nodes (shown with a shaded center) force arcs that have been left dangling by node deletion to be attached to another node. This allows the easy expression of node replacement and algebraic expressions. Negatives (shown with thick lines in the diagrams in this paper, but on the screen they are coloured red) allow the user to specify that a certain subgraph should not be matched.

Where duplicate labels appear in the LHS or RHS they must be identified by the user to avoid confusion. The identifier is an integer superscripted to the node label. An example of identification is shown in Figure 4.3, where the fourth rewrite contains identifiers for both 'P' and 'X'. The identifiers do not affect matching, but ensure that there is no ambiguity when duplicating or removing primitives.

Graph theoretic operations, sometimes called graph associations [1], are expressible in Grrr as a direct consequence of the paradigm that it uses. These operations allow paths through the graph to be found, or allow semantic distance to be calculated. These sort of operations are widely used in graph drawing. In textual languages these operations require new primitives to be added to the language.

Grrr algorithm animation is improved from the functionality in Spider. Previously, the user had a debugging feature that allowed the user to step through the rewriting in the host graph. In Grrr the functionality has been improved to animate algorithms by showing the changes in the host graph whilst the rewriting is progressing automatically. Chosen node types can be selected by the user to be hidden, so ensuring an uncluttered view of changes made to the graph. For example, in Section 3.2, the oval nodes that represent associated information might be hidden, so ensuring that only changes to the data graph can be seen.

### 3.2. Graph drawing facilities

To be a useable graph drawing language Grrr has been augmented with triggers that perform geometric operations, to measure the position of nodes in the graph and allow nodes and arcs to be placed in specific positions. These use either screen pixels or the size of nodes as units of measurement.

Some geometric triggers are atomic and so must be built in. These provide basic information, such as: the X Y coordinate position of nodes; the height and width of the nodes themselves; and the length of arcs. Correspondingly, to alter the appearance of the graph, there are atomic triggers to specify the X Y position of nodes, and the points at which non-straight arcs bend. The last of these requires a slight change to the rewriting method used by Grrr to give a high priority to initiating these triggers. The addition of the arc bend trigger temporarily alters the graph theoretic structure of the data graph, so that any rewrite which would normally match the repositioned arc would not do so with such a trigger in place, hence arc bends must be initiated before any other trigger.

Other operations for graph drawing can be derived from the above triggers, but to improve efficiency, many of these are built in. Enclosure operations in particular are inefficient to derive. These include finding the nodes within a particular bounding rectangle, or the nodes within a certain radius of a centre point. To derive these, each node in the graph would have to be examined, and the results processed. Instead of this heavy computational overhead they are hard coded in Grrr. Their analogs, finding a particular area, rectangular or circular, that encloses a set of nodes, are similarly built in.

There are further built in triggers. Finding the distance between two nodes is a common operation, and a hard coded trigger to perform this operation reduces execution time. For other operations it is not so clear that the balance of time spent on implementation is worth the saving in execution time. If it becomes clear that certain operations, such as finding the angle between two arcs, or placing a set of nodes in a straight line are becoming bottlenecks, then they can be added as built in triggers.

Graph drawing algorithms often work on higher level units than pixels, and operations on units based on node size are present, so that the node width is one unit in the X direction and node height one unit the Y direction. The implementation of these triggers shares a large amount of code with the triggers that operate on pixels, given above.

The use of geometric triggers can be seen in the 'ClosestPoint' transformation of Figure 4.3. 'XPixelPlace' can be seen in the fourth RHS. This puts the node attached by a 'move' arc at the X pixel coordinate specified by the node attached to a 'point' arc. There is a 'YPixelPlace' for the placing nodes at a Y pixel coordinate. The position in terms of node width and height can be specified by 'XIntPlace' and 'YIntPlace', the latter is used by the 'LayoutY' transformation, not shown due to space reasons. The corresponding triggers which find the coordinates of a node are 'XPixelCoord', shown in Figure 4.3 and 'YPixelCoord'. These produce a new attractor node containing the coordinate of the node attached to an 'arg' arc. 'XIntCoord' and 'YIntCoord' find the X and Y coordinates in terms of node width and height, rounding to the nearest integer. 'NodeWidth' can be seen in the fourth RHS. This returns the width, in pixels, of the node attached by an 'arg' arc. The result is an attractor node. 'NodeHeight' finds the height, in pixels of the 'arg' node.

The transformation shown in Figure 4.3 also makes use of 'BBox', which finds the bounding rectangle of one or more nodes attached to it by 'arg' arcs, and returns the coordinates as pixels attached to the specified result node. The values are attached to 'min X', 'min Y', 'max X' and 'max Y' arcs, giving the rectangle in terms of the bottom left point and the top right point. This method of returning a result means that other related triggers can easily make use of the information returned, in particular in this figure, the trigger 'OverlapBox' uses the result. It finds the nodes wholly or partially contained within a specified rectangle, given in the format above. The 'InBox' trigger is not shown, but returns the nodes contained wholly in a rectangle. The corresponding commands for circular areas, again not shown, use a center point specified by 'X' and 'Y' pixel coordinates and an integer pixel 'radius'.

## 4. Graph drawing example

In this section, graph drawing in Grrr is illustrated using a directed acyclic graph (DAG) drawing technique based first on finding the Y location of nodes according to their position in the hierarchy. Then the X coordinates are specified using multiple passes though the graph, alternately down the hierarchy sorting on child barycenters, then up sorting on parent barycenters. This is a visualised (and simplified) version of the approach used by Sugiyama et. al. [20]. The Grrr system is not restricted to DAGs, or indeed hierarchical or centred graphs. Any circular, general graph may be manipulated. This example is given because it uses complex recursion, and it highlights some interesting programming and layout features of Grrr.

Figure 4.1, Figure 4.2 and Figure 4.3 show some transformations in the program that lays out a hierarchical graph. Figure 4.1 shows the top level transformation 'LayoutGraph'. This is the entry point into the program. Figure 4.4 shows the host graph at the start of execution where the trigger node 'LayoutGraph' is attached to a node indicating the number of passes through the graph. In this case 3 passes will be made. The data graph contains unlabelled (circular) nodes and unlabelled arcs to improve the clarity of the diagram. However, the manner in which the transformations are written would mean that any labels could be used for data graph nodes and arcs. The diagrams showing the host graph indicate the number of steps that have been executed from the initial host graph. Each step is a single graph rewrite.

'LayoutGraph' controls the execution order of the lower level transformations by using flag nodes. The transformation has four rewrites. Each rewrite has an LHS and an RHS. On each application of the transformation matches are attempted on each LHS in turn, starting with the topmost LHS. The difference between the LHS and RHS specifies which nodes and arcs in the host graph are to be added or deleted. The first LHS matches when the node attached by a 'passes' arc is '0', however at the start of execution the number of passes is '3', so the first rewrite is not used. The second LHS matches in the case where there is no 'done Y' flag. This is because the 'done Y' node and connecting arc are bold, indicating that they are negative, and so there will be a match if that part of the LHS graph cannot be found in the host graph. It prevents 'LayoutY' executing more than once because the 'done Y' flag is created by this rewrite. The 'X' node is a variable, indicated by an italic font. In this example the start of variable labels is capitalised. The third and fourth rewrites toggle between calling 'DoChildBCs' and 'DoParentBCs', as the 'done Child' flag is alternately created and deleted by these two rewrites.

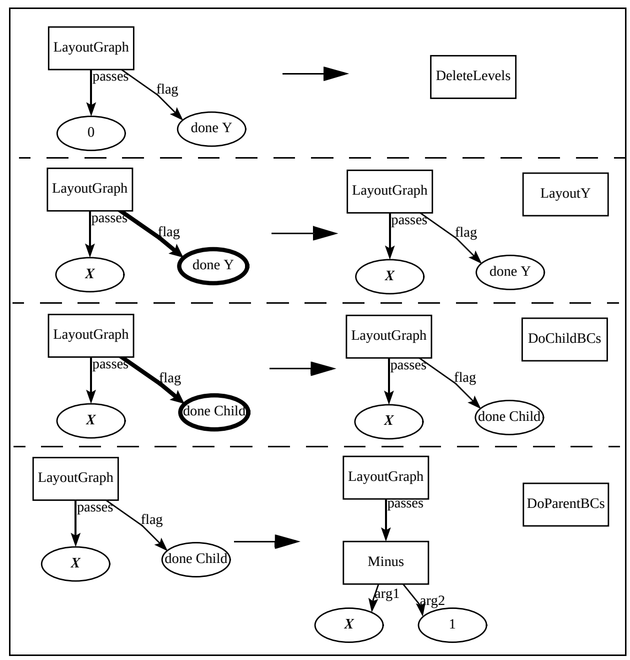


**Figure 4.1. The transformation 'LayoutGraph'**

Each time the fourth rewrite is applied, the number of passes is reduced by one. This means that eventually the applications of 'LayoutGraph' will reduce it to '0', when the next application of 'LayoutGraph' will mean the LHS of first rewrite will now match. This first rewrite of 'LayoutGraph' is the terminating case, deleting the 'LayoutGraph' trigger from the host graph and replacing it with a tidying trigger, **'**DeleteLevels'**,** which removes the nodes holding information from the graph and leaving only the repositioned data graph, as shown in Figure 4.6.

Figure 4.2 shows the transformation 'DoChildBCs' which iterates through the nodes in the data graph in the order of the Y level of the nodes placing them at the average of the X coordinates of their children. For reasons of space other transformations are left out of this paper. In particular, missing is the code for ‘LayoutY’, which sets the Y levels of nodes and places them appropriately, and ‘DoParentBCs’ which is similar to ‘DoChildBCs’.

The trigger that initiates ‘DoChildBCs’ is created by the third rewrite of ‘LayoutGraph’. Because Grrr uses a newest first execution strategy, this new trigger, and all triggers that are created as a result of its application will complete execution before ‘LayoutGraph’ is called again.

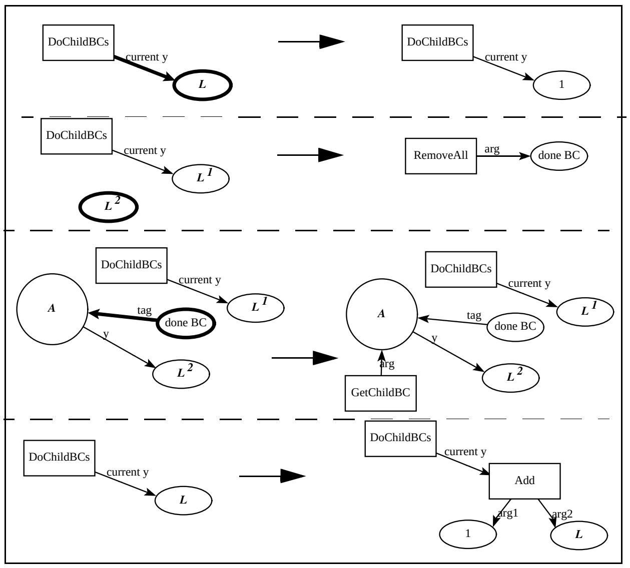


**Figure 4.2. The transformation ‘DoChildBCs’**

The first rewrite is called when there is no node attached to a ‘current y’ arc. The rewrite creates the ‘current y’ arc and attached node which has a value set to the top of the hierarchy, ‘1’. The second rewrite is the terminating case which calls a tidying transformation ‘RemoveAll’. It matches when there is no level corresponding to the current level, hence the bottom of the hierarchy must have been passed. The third transformation matches a node at the current level and calls ‘GetChildBC’, which is not shown. It calculates the barycenters and attempts to place the node sensibly, using the ‘ClosestPoint’ transformation, described below. The fourth rewrite increments the current level. If the increment is more than any current level then the next application of the transformation will mean the second LHS matches, and the trigger will be deleted.

Figure 4.3 shows the transformation ‘ClosestPoint’**.** This illustrates some of the built in geometric triggers. It is designed to find a reasonably close X coordinate for the argument node which does not overlap with any other nodes. It does this by alternately moving the node right then left with bigger steps until a clear location is found. This transformation also illustrates the high level programming possible with Grrr, because there is a large amount of processing performed by the five rewrites.

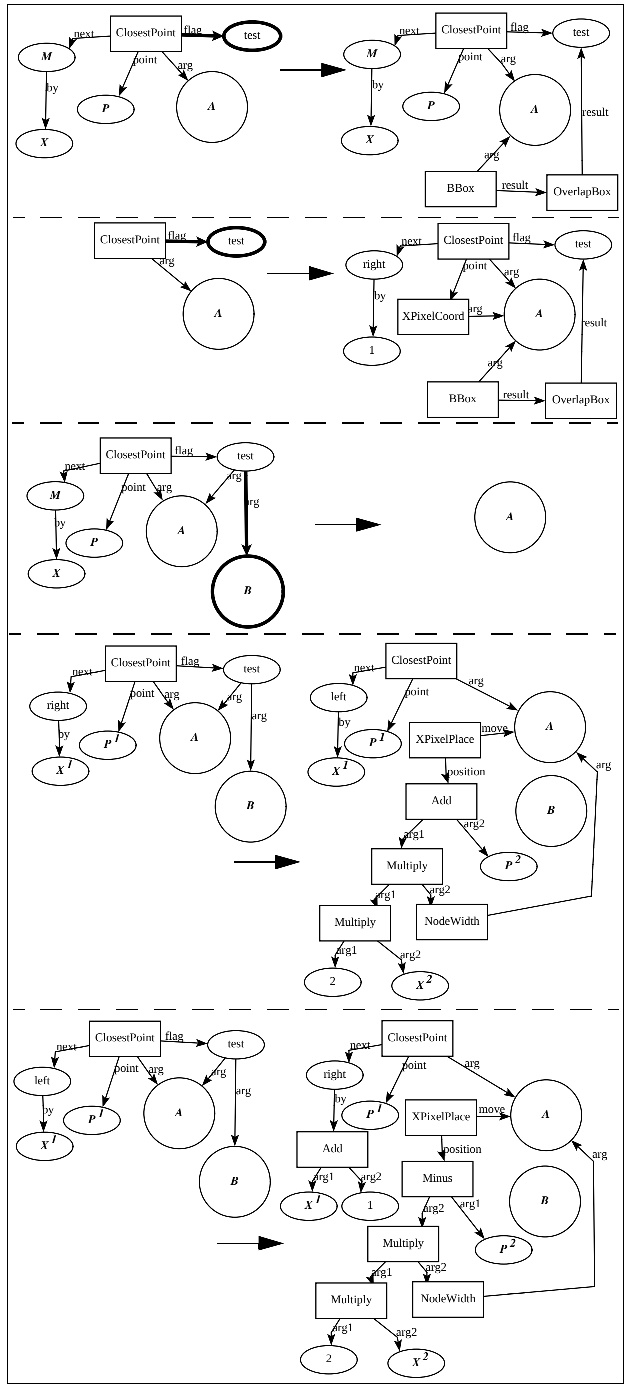


**Figure 4.3. The transformation ‘ClosestPoint’**

‘ClosestPoint’ is designed to be called with only the node to be positioned attached, so the second rewrite is the first to be called, as the first requires ‘point’ and ‘next’ tag nodes to be attached. The RHS creates a ‘test’, a ‘right’ and a ‘1’ node. It also creates three geometric triggers, ‘XPixelCoord’, ‘OverlapBox’ and ‘BBox’. The triggers are the same age but ‘XPixelCoord’ is executed first according to the serial trigger resolution algorithm. It returns the value, in pixels of the X position of the argument node. This value becomes the node attached to the ‘point’ arc, and is used in later calculations. ‘OverlapBox’ cannot yet be called as none of its RHSs will match, it requires associated coordinates, hence ‘BBox’ will be the next executed. This returns, attached to the node connected by the ‘result’ arc, the bounding box of the nodes attached by ‘arg’ arcs. Here there is only one such node. The point in execution after ‘BBox’ is called is shown in Figure 4.5. This figure also shows the effect of the execution of ‘LayoutY’, with the nodes in their correct level in the hierarchy, and with the tags giving the level attached.

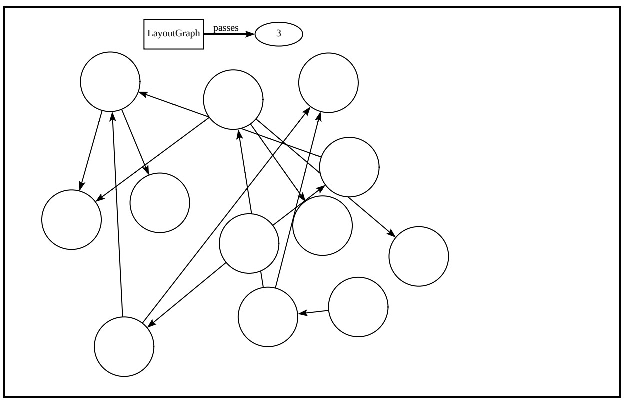


**Figure 4.4. The initial host graph (Step 0)**

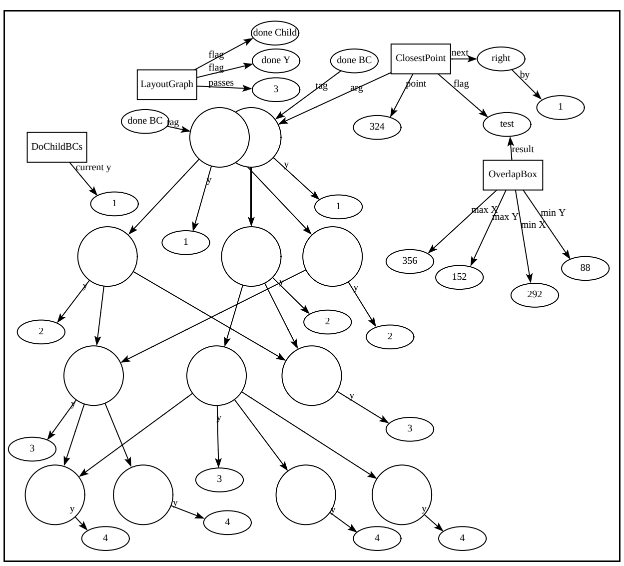


**Figure 4.5. The host graph after Step 167**

Now ‘OverlapBox’ can match, as it has the required coordinates, and this returns attached to the node connected by the ‘result’ arc the nodes in the graph which are wholly or partially contained within the given coordinates. If there is only one (which will be the argument node, hence it does not overlap another node) then the third LHS of ‘ClosestPoint’ will match next, and recursion will terminate. If there is more than one, then depending on whether the next move is left or right, the fourth or fifth rewrites will be next used. In this case, two steps after the host graph in Figure 4.5 it is the fourth rewrite. The RHS of the fourth rewrite has an algebraic expression which calculates the new location of the node, according to the previous number of moves, the original position and the node width (found using another geometric trigger, ‘NodeWidth’). The fourth rewrite deletes the ‘test’ tag, so that the next application of ‘ClosestPoint’ will mean the first rewrite is used, which sets up the graph for the next test of the position of the argument node. The fifth rewrite is similar to the fourth, but increases the amount by which the node is moved.

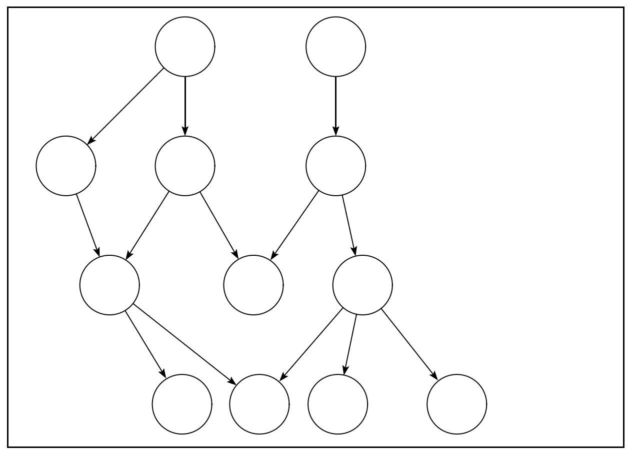

**Figure 4.6. The final host graph (Step 1584)**

Execution finishes after 1584 rewriting steps, resulting in the host graph as shown in Figure 4.6. Transformations delete their associated triggers when they complete recursion and tidying transformations have deleted all the associated data, leaving only the data graph as the final result.

## 5. Conclusions and further work

This paper has demonstrated the feasibility of visual graph drawing by giving an example of a concise, but powerful hierarchical graph drawing algorithm which makes use of the new geometric operators and previously available implicit graph theoretic operations.

This paper has also shown a novel application area for graph rewriting languages, an area for which these systems are uniquely suited by virtue of their visual representation of graphs and computational completeness.

With enhancements it is conceived that Grrr will be a practical tool for the graph drawing community. The major further work required to turn Grrr into a serious development tool are improvements in user interface and execution efficiency. The former requires resources, but the latter is more interesting in research terms. Grrr uses a general but sometimes expensive serial graph matching algorithm. In practice most graph matching operations can be optimised, either at compile time by converting rewrites to a more efficient form, or during run time by using intelligent searches through the host graph. Such efficiency gains should be achievable without altering the Grrr programming method.

Further programming features could be added to Grrr. Because the semantics have been maintained at a simple level there are various enhancements that the example program highlights. Flags are used to control the order of execution of some lower level transformations. A more elegant method might be to add a system of ensuring specified rewrites are executed only once. Also widely used are tags, and the situations in which tags are applied could be dealt with by specifying nodes in LHS graphs that should be matched only once by a particular trigger node application.

The extensions that enable graph drawing in Grrr have been designed to integrate into the language without greatly affecting the semantics. Manipulation of graphs might be more intuitive if the operations were directly embedded into the rewriting process. For example, the repositioning of a node could be specified by its position in the RHS relative to the LHS. New forms of arc are also possible. These might alter the position of the connected nodes (bringing them closer, or separating them). Grrr would then become more expressive at the cost of simplicity.

Graph drawing is a large field, and work is in progress to discover which classes of algorithm are amenable to visual graph drawing and to compare the effort required for producing visual algorithms against producing their textual duals. It is also conceivable that this new visual approach to graph drawing will lead to the development of novel algorithms.

There are graph drawing requirements other than discussed in this paper. In particular, it might be useful to add operations that allow automatic measuring of aesthetic criteria that is not easy to derive from current triggers. This includes measuring arc crossing in a graph and the symmetry of a graph. Work in the area of constraints on graph drawing [5] might also be applicable to this system.

There are a number of computational tasks related to graph drawing, such as planarity testing. Interestingly, this does not require geometric operations, only graph theoretic operations. Implementing planarity testing and other related tasks into Grrr is an area of further work.

## References


1. R. Ayres and P.J.H. King. Extending the Semantic Power of Functional Database Query Languages with Associational Features. *Congres INFORSID 1994*. pp. 301-320. 1994.
2. G. Di Battista, P. Eades, R. Tamassia and I.G. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. *Computational Geometry: Theory and Applications, 4*. pp. 235-282, 1994.
3. F.J. Brandenburg. Designing Graph Drawings by Layout Graph Grammars. *Graph Drawing '94. LNCS 894.* Springer-Verlag. pp. 266-269. 1995.
4. S. Bridgeman, A. Garg and R. Tamassia. A Graph Drawing and Translation Service on the WWW. *Graph Drawing '96. Berkeley, CA, USA, LNCS 1190.* Springer-Verlag. 1996
5. I.F. Cruz. Expressing Constraints for Data Display Specification: A Visual Approach. *Principles and Practice of Constraint Programming, eds. Vijay Saraswat and Pascal Van Hentenryck*. The MIT Press, pp. 443-468, 1995.
6. I.F. Cruz, A.O. Mendelzon and P.T. Wood. G+: Recursive Queries Without Recursion. *Proceedings of the 2nd Expert Database Systems Conference*. Benjamin-Cummings. pp. 645-666. 1989.
7. M. Fröhlich and M. Werner. Demonstration of the Interactive Graph-Visualization System daVinci. *Graph Drawing '94. LNCS 894.* Springer-Verlag. pp. 266-269. 1995.
8. D. Hils. Visual Languages and Computing Survey: Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing 3(3)*, pp. 69-101. 1992
9. M. Himsolt. The Graphlet System. *Graph Drawing '96. LNCS 1190.* Springer-Verlag. pp. 233-240. 1996
10. M. Himsolt. GML: A Portable Graph File Format. *Technical Report*, Universität Passau, 1997.
11. S.M. Kaplan, S.K. Goering and R.H. Cambell. Specifying Concurrent Systems with Δ-Grammars. *Proceedings of the Fifth International Workshop on Software Specification and Design*. Society Press. pp. 20-27. 1989.
12. E. Koutsofios and S.H. North. Drawing Graphs with DOT. *User Manual*. AT&T Bell Laboratories. 1993.
13. C. McCreary. Visualizing Graphs with Java (VGJ) Manual. *Available from http://www.eng.auburn.edu/department/cse/research/graph_drawing/manual/vgj_manual.html*. 1997.
14. L. Mohan L. and R.L. Kashyap. A Visual Query Language for Graphical Interaction With Schema-Intensive Databases. *IEEE Transactions on Knowledge and Data Engineering, 5, 5*. pp. 843-858. 1993.
15. J. Paredaens, J. Van den Bussche, M. Andries, M. Gyssens and I. Thyssens. An Overview of GOOD. *ACM SIGMOD Record, 21,1*. pp. 25-31. March 1992.
16. P.J. Rodgers and P.J.H. King. A Graph Rewriting Visual Language for Database Programming. *The Journal of Visual Languages and Computing 8(6)*. Academic Press. pp. 641-674. December 1997.
17. P. Rodgers, R. Gaizauskas, K. Humphreys and H. Cunningham. Visual Execution and Data Visualisation in Natural Language Processing. *Proceedings of the VL'97 IEEE Symposium on Visual Languages*. pp. 342-347. 1997.
18. A. Schürr. Rapid Programming with Graph Rewrite Rules. *Proceedings USENIX Symposium on Very High Level Languages (VHLL), Santa Fe*. pp. 83-100. October 1994.
19. A. Schürr. BLD - A Nondeterministic Data Flow Programming Language with Backtracking. *Proceedings of the VL'97 IEEE Symposium on Visual Languages*. pp. 398-405. 1997.
20. K. Sugiyama, S. Tagawa and M.Toda. Methods for Visual Understanding of Hierarchical Systems. *IEEE Trans. on Systems, Man and Cybernetics, SMC-11(2)*. pp. 109-125. 1981.
21. G. Zinßmeister and C.L. McCreary. Drawing Graphs with Attribute Graph Grammars. *Graph Drawing '94. LNCS 894.* Springer-Verlag. pp. 266-269. 1995.