A Demonstration of the Grrr Graph Rewriting Programming Language

Peter J. Rodgers and Natalia Vidal

Computing Laboratory, University of Kent, UK
{P.J.Rodgers, N.Vidal}@ukc.ac.uk

Abstract. This paper overviews the graph rewriting programming language, Grrr. The serial graph rewriting strategy is detailed, and key elements of the user interface are described. The system is illustrated by a simple example.

1 Introduction

The basic elements of the Grrr system are described in this paper. It allows graph data structures to be visualised and has a computationally complete declarative programming method. This paper concentrates on detailing the core rewriting strategy, other literature [4,5,6] describes the more complex features that have been added for ease of programming, changing the rewriting method or execution order. However, the core of Grrr remains a serial, deterministic rewriting strategy with a top down matching method.

Other graph rewriting systems use different variants on the graph rewriting method, and visualise programs and graphs in alternative ways. Examples of such systems are Good [3], Progres [7], MONSTR [1] and Δ-grammar programming [2].

In Grrr, graphs have labeled nodes and labeled directed edges. This allows simpler graphs, without labels, or loop free to be specified if required. There are several different node types which allows the data graph to be differentiated from information derived from the graph during execution.

The prototype is too slow and the interface is to clumsy for industrial usage, but the current system allows experimentation and proof of concept. There are several suggested application areas: database programming, graph drawing, associational relations and graph algorithm animation. These share a graph based, possibly visual view of data, that need complex calculations.

The next section contains a worked example, based on a very simple program. The final section details possible further work on the Grrr prototype.

2 Example

This section shows the execution method of Grrr by a simple example.

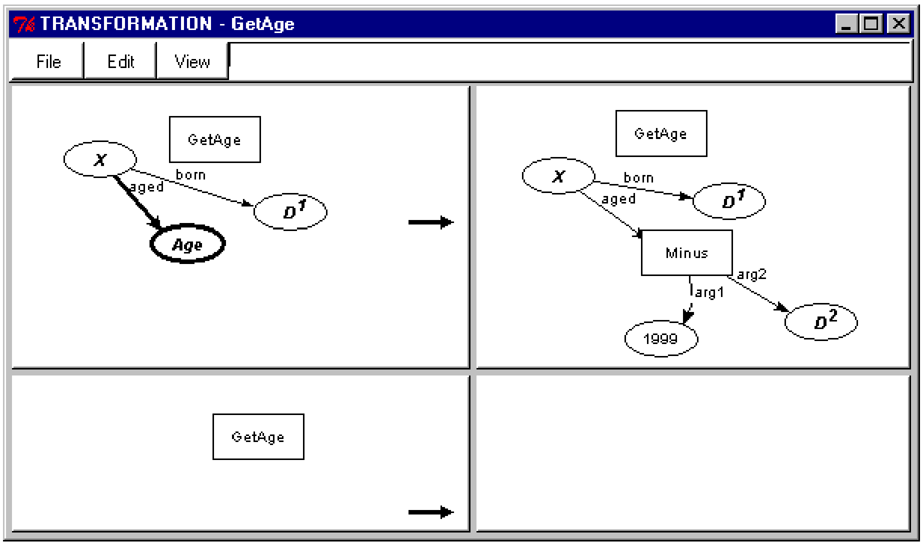

Fig. 1. A transformation window containing the transformation 'GetAge'. This transformation calculates the age of people given their birth date. The transformation has been simplified for clarity with the calculation dealing only with the year, and not with months or days, and the current year, '1999', is hard coded into the program

Fig. 1 shows a transformation window containing the transformation 'GetAge', which has two rewrites. The first rewrite tests for a person who has yet had their age calculated and calculates their age from the current year. The second rewrite, which is only called after the first fails to match, terminates recursion by deleting the initiating trigger node.

Every rewrite has a left hand side (LHS) and a right hand side (RHS). In a rewrite, the differences between the positive part of the LHS graph and the RHS graph indicate which nodes and edges are to be added and deleted in the host graph. The first rewrite does not remove any primitives, but it adds an edge 'aged' attached to a new trigger node 'Minus', the constant node '1999' attached to 'Minus' an edge 'arg1, a copy of the variable node 'D' attached to 'Minus' by 'arg2'. Here we use the convention that the labels of variables are shown with capitalised first letters, and constants are shown with lower case first letters. An exception are the rectangular trigger nodes, which are always constant, no matter what their label.

The 'Age' node and connecting 'aged' edge in the first LHS are both negatives (shown by thick lines), indicating that they must not be able to match for the LHS to match. Hence, the LHS will only match a person 'X' and a birth date 'D^1' if there has been no age calculated yet for the person. The superscript to 'D' is required because there are two instances of 'D' in the RHS, hence the programmer must specify which was the original, and which is the new copy. The copy, 'D^2', is used in the calculation to get the age.

The 'Minus' trigger calls a built in transformation. It requires two argument nodes attached by edges with labels 'arg1' and 'arg2', the label of the node attached to the second argument is taken away from the label of the node attached to the first, and a new node labeled with the result is created and is attached to the 'aged' edge. The trigger is deleted, as are the two argument nodes and edges.

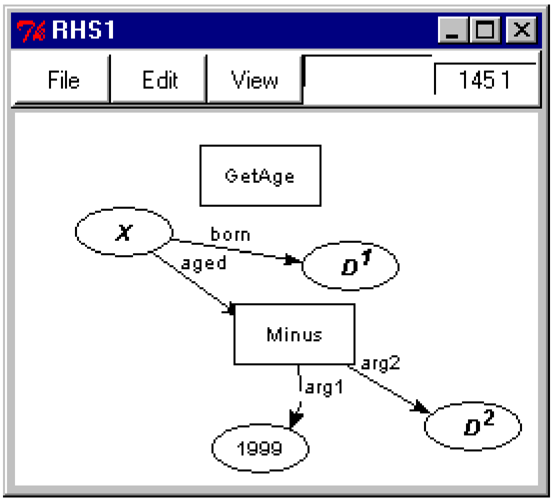

Fig. 2. The first RHS graph of 'GetAge' in a graph editing window. The numbers in the right hand corner indicate the current coordinates of the cursor

The transformation window does not allow graphs to be edited. It only has facilities to delete existing rewrites and create blank rewrites. To edit a LHS or RHS graph, a graph editing window has to be brought up by double clicking on a graph in the transformation window. A graph editing window with the first RHS of 'AddAge' is shown in Fig. 2. Numerous graph editing windows may appear on the screen at any time. The functionality these windows provide includes adding new nodes and adding new edges between existing nodes (edges are added after two nodes have been selected). Labels and node types can be changed. Groups of nodes and edges can be selected, and so deleted, cut, copied and pasted. When deleting or pasting, dangling edges are deleted. The syntax of LHS or RHS graphs is maintained by ensuring that invalid nodes or edges cannot be added to graphs. For instance, negatives cannot be added to RHS graphs, and more than one trigger cannot be added to LHS graphs.

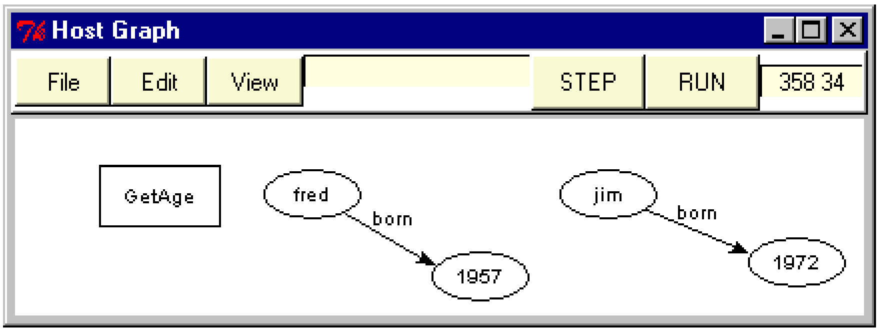

Fig. 3. The host graph window, with an example of using 'GetAge'

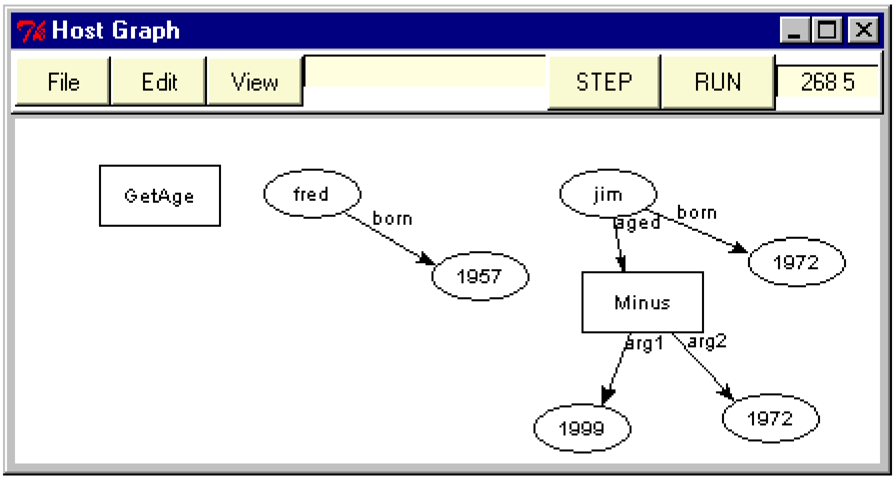

Fig. 4. The host graph after step 1

An example host graph window is shown in Fig. 3. This is the graph where rewriting occurs. It contains editing options much like the window of Fig. 2, in addition it has options to initiate rewriting: 'Step' and 'Run'. Step performs the next rewriting step only, whereas Run rewrites the graph until there are no trigger nodes remaining. A rewriting step consists of a single trigger node in the host graph initiating a single rewrite of the transformation with the same name as the trigger label. The rewrite changes only one subgraph in the host graph. The first host graph has only one trigger node, 'GetAge', so this is the one to be executed. The topmost LHS of the transformation is the first to be tested in the host graph. In this case a subgraph in the host will match, and so the rewrite is used.

There are in fact two possible matches: the node 'X' with 'fred' or 'jim', and the node 'D' with the respective years. Where there is a choice of subgraph to match the decision is made by an iterative sort of both the LHS graph and the host graph, and matching the highest valued subgraph. In this case 'jim' is the one to match, as 'jim' is ordered higher than 'fred' ('j' is higher in the alphabet than 'f'). The host graph is then changed as defined by the rewrite. The host graph after rewriting is shown in Fig. 4.

This first rewriting step adds nodes to the host graph, including the 'Minus' trigger node. The presence of this new trigger means there are now two triggers in the graph. As 'Minus' is newest, it is executed first. This newest first trigger initiation strategy means that higher level triggers can remain in the host graph whilst transformations that they call are executed. This allows programs to be structured in a hierarchical manner.

Minus is built in and calculates the difference between 1999 and 1972, creating a node with the result as its label, whilst deleting the nodes involved in the calculation. There are many built in transformations, taking various arguments. Some are atomic, in that they cannot be derived from other primitives in the system, others have been added for efficiency reasons. The result of executing 'Minus' can be seen in Fig. 5

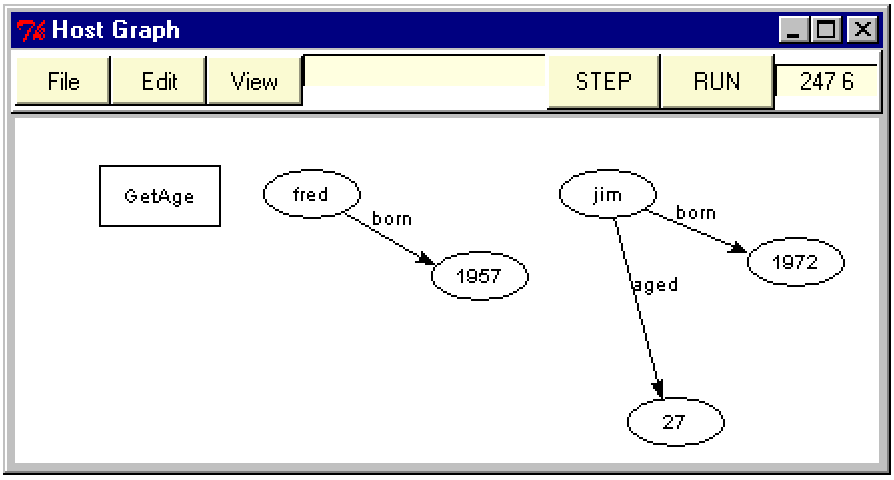

Fig. 5. After step 2

The only trigger in the host graph is now the original 'GetAge' trigger, it is executed and so will again cause the first LHS to be tested in the host graph. 'jim' will not now match with 'X', as '27' attached to 'aged' matches with the negatives. This means 'fred' is the only person that can match and so that part of the graph will be rewritten, assigning an age to the node. The host graph after both that execution step and the age calculation step is shown in Fig. 6.

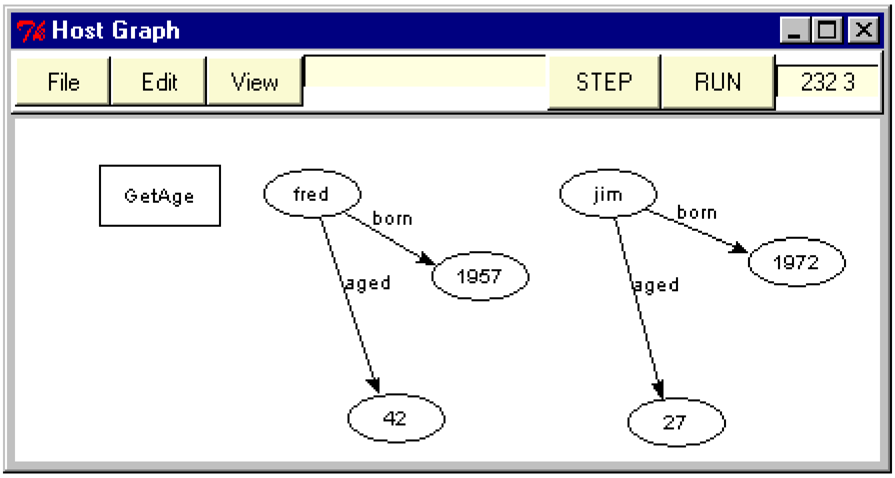

Fig. 6. After step 4

Both people in the host graph now have ages. The first LHS of 'GetAge' will now no longer match, because the negative edge and node will match the edges and nodes attached to both 'jim' and 'fred', hence the second LHS will be tried. This will match, as it looks only for a trigger node, and so the rewrite will occur. The rewrite simply deletes the trigger node, terminating the program as there are no more trigger nodes in the host graph, as shown in Fig. 7.

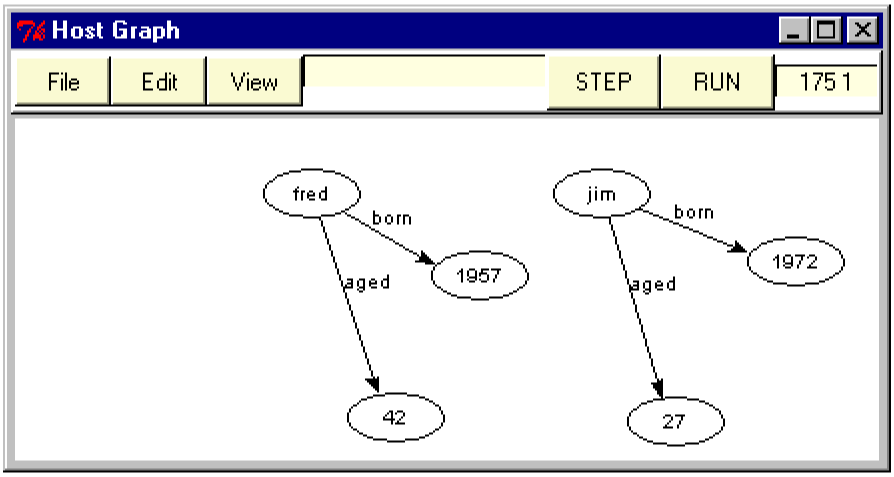

Fig. 7. The final host graph, after step 5

There are various ways of showing the execution in the host graph. The fastest method is to execute the program by the Run button, and see the final result after execution has finished. However, to aid debugging, it is possible to highlight nodes that have been matched, and see the continuous execution occurring as it happens in the window.

3 Further Work

This paper has worked through a simple example of programming with graph rewrites. Using similar techniques it is possible to create complex programs to alter visual representations of graph data. However, there is much work that might be done to augment Grrr, improve its efficiency and adapt it to new application areas.

The user interface needs improvement. Unlike text editing, graph editing needs specific, application based tools, particularly for systems such as Grrr, which rely on editing restrictions for syntactic correctness. Graph editing in Grrr can be improved by faster node and edge creation, improved cutting and pasting and changing the treatment of dangling edges. Changes are also needed to the visualisation of rewriting, and the addition of a good incremental graph drawing algorithm would improve the appearance of the host graph.

The programming language has no concept of libraries, encapsulation or other software engineering tools. The concept and design of such features will require more effort, but such additions should increase the portability, usefulness and attractiveness of the language.

Improving the efficiency of execution is always a goal of language designers. The current implementation could be much streamlined. Also, there are many possible optimisations of graph matching and graph rewriting that could be explored. Other optimisations that could be explored rely on knowledge about the application, and restrictions on graphs.

Improving execution efficiency should allow the scale of the graphs that are rewritten to be increased. As graphs get bigger the problems of visualising the graph also increases, and problems storing graphs have to be dealt with, as a graph database is required.

Further exploration in applications is always possible, with graphs widespread in computer science, particularly in areas such as networks, parallel computing and software engineering. The modifications required to meet the needs of such areas are possible interesting areas of research.

Acknowledgements

This work was partially supported by funding from the UK Engineering and Physical Sciences Research Council (EPSRC), grant reference GR/M23564.

References

1. Banach R.: MONSTR I -- Fundemental Issues and the Design of MONSTR. Journal of Universal Computer Science 2,4 (1996) 164-216.
2. Kaplan S.M., Goering S.K. & Cambell R.H.: Specifying Concurrent Systems with Δ Grammars. Proceedings of the Fifth International Workshop on Software Specification and Design. Society Press (1989). 20-27.
3. Paredaens J., Van den Bussche J., Andries M., Gyssens M. and Thyssens I.. An Overview of GOOD. ACM SIGMOD Record, 21,1. (March 1992) 25-31
4. Rodgers, P.J.: A Graph Rewriting Programming Language for Graph Drawing. Proceedings of the 14^{th} IEEE Symposium on Visual Languages, Halifax, Nova Scotia, Canada. IEEE Computer Society Press (1998) 32-39.
5. Rodgers P.J. and King P.J.H.: A Graph Rewriting Programming Language for Database Programming. The Journal of Visual Languages and Computing 8(6), 1997. 641-674.
6. Rodgers P. J. and Vidal N.: Graph Algorithm Animation with GRRR. In Agtive99: Applications of Graph Transformations with Industrial Relevance, Kerkrade, The Netherlands, September 1999. LNCS. Springer-Verlag.
7. Schürr A.: Rapid Programming with Graph Rewrite Rules. Proceedings USENIX Symposium on Very High Level Languages (VHLL), Santa Fe. October 1994. 83-100.