Visual Execution and Data Visualisation in Natural Language Processing

Peter Rodgers, Robert Gaizauskas, Kevin Humphreys, Hamish Cunningham
Department of Computer Science, University of Sheffield, UK
{peterr,robertg,kwh,hamish}@dcs.shef.ac.uk

Abstract

We describe GGI, a visual system that allows the user to execute an automatically generated data flow graph containing code modules that perform natural language processing tasks. These code modules operate on text documents. GGI has a suite of text visualisation tools that allows the user useful views of the annotation data that is produced by the modules in the executable graph. GGI forms part of the GATE natural language engineering system.

1. Introduction

The current relationship between visual languages and natural language processing (NLP) is restricted to translating graphical languages into natural language [1] or visual representations of text processing languages [13]. We believe that there is a great deal of potential for expressing the execution of NLP systems visually. One reason for this is the modular nature of NLP algorithms, which mean that a data flow visual language is a natural way of representing NLP programs. There is also a great need for generic tools that allow the visualisation of data associated with textual documents after they have been analysed by NLP techniques.

This paper concentrates on the visual execution of NLP tasks using data flow techniques, and visualising the information that results. Specifically, the paper describes GGI – the GATE Graphical Interface. GGI is a tool for visualising the execution and data of programs integrated into GATE [5], a natural language engineering environment which aims to support researchers and developers of NLP systems and applications by supplying facilities for modular reuse of NLP software, management of large text collections, and visualisation of processing results (see Section 2).

While GGI provides a full user interface to GATE, including, for example, support for file management, there are two aspects of it that are of interest here. First, GGI provides an autogenerating, customisable, graph for controlling the execution of interdependent NLP modules. Second, GGI provides a class of generic visualisation tools for viewing the complex information computed about texts by NLP modules.

In GATE, execution of all modules is performed in an executable graph that is a simple form of data flow diagram in which the nodes are the modules or functions to be executed and the arcs represent data flows. We call this graph the *system graph*. The functions that form the nodes have a large computational granularity and are of comparable computational size to the functions seen in, e.g., ConMan [9].

This graph is less computationally expressive than is typically found in visual data flow languages [10, 18], as it contains no looping (iteration) or distributor constructs (by distributor construct we mean that the result of execution of an upstream module defines which downstream module is to be executed). However, this simplicity has benefits for the modular system development architecture that GATE aims to supply.

In particular, it is possible to autogenerate the data flow program (system graph) from the declaratively stated pre- and postconditions that each module in the GATE system must have. The preconditions define the data that must be present before a module can be run; the postconditions define the data that will be present after a module has been run. Together these permit the dynamic construction of the execution graph's arcs and mean that no 'hard-coding' of module connections is required. At run time actual data flow is mediated by a common database through which all modules intercommunicate and the execution graph conveys the state of the database to the user through the colouring of modules according to a traffic-light metaphor to indicate their executability.

The autogeneration procedure means that users do not need to take directly into account the other modules in the system (or unknown modules that might in the future be added to the system) when they integrate a new module into GATE. It thus helps to realise GATE's objective of providing a 'plug-and-play' architecture for natural language engineering. The executable graph is described in more detail in Section 3.

The second aspect of the GGI we describe below is the set of data visualisation tools it provides. The data produced from the execution of a module can be viewed directly from the system graph. Clicking on the module brings up the list of postcondition data types, i.e., the data that the module has created. Selecting one launches an appropriate results

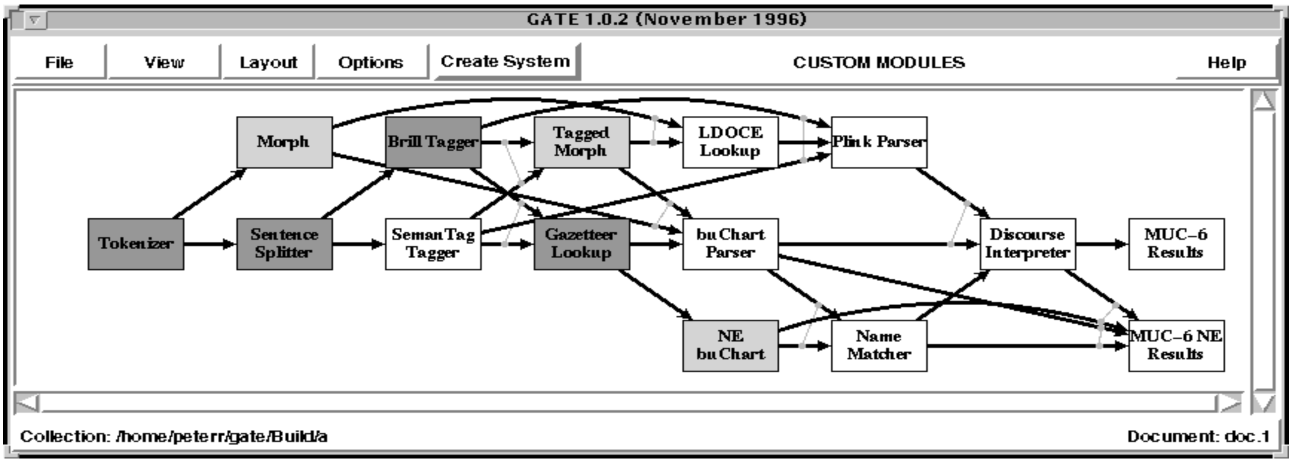

Figure 1. The GATE System Graph

viewer. These data visualisation tools range from the simple highlighting of annotations in the text to graph representations of parse trees and conceptual hierarchies represented as DAGs and are described in more detail in Section 4.

2. Background: GATE and NLP

GATE, and as a consequence GGI, is the result of recent developments towards code reuse in NLP. In particular GATE is based on an underlying database which conforms to the TIPSTER database standard [8]. This standard has been defined by the Architecture Committee of the ARPA-sponsored TIPSTER programme with the intention of providing a common framework for advanced text processing systems, such as information retrieval and information extraction systems. We have adopted this standard in GATE believing it to have even wider potential utility within natural language engineering, for example for machine translation, summarisation, and computer-assisted language learning.

In the TIPSTER model arbitrary *annotations* about documents are stored in a database separate from the text. An annotation consists of at least an *annotation type* and one or more *spans* (a pair of numbers that indicate a start point and an end point in the text). Further *attributes* (attribute-value pairs) can be associated with an annotation, including references to other annotations.

Figure 2 shows some annotations. It contains two kinds of annotation type: sentences and tokens. Tokens are the result of tokenization, a process that divides a text up into elements such as punctuation and 'words'. Associated with the token annotations are part-of-speech attributes which indicate the word class of each token. The sentence annotations have references to the tokens that are in them.

Integration of NLP modules in GATE is achieved by requiring modules to read and write their final annotations to a TISPTER database. A module is an interface to a resource which may be predominantly algorithmic (e.g. a parser) or predominantly data (e.g. a lexicon), or a mixture of both. Typically, a GATE module will be a wrapper around a pre-existing NLP module or database (hence, software reuse). It might seem that pipelining communication of executable modules through a database is inefficient. However, such is the nature of NLP tasks that any module might perform a large amount of computation which reduces the significance of the database overhead. Further, the objective of GATE is to promote code reuse between research groups and the construction of experimental systems for which efficiency is not the prime concern. Finally, the model maps naturally to client-server setup which easily supports distributed access to algorithmic and data resources.

The example in Figure 2 also illustrates how modules operate in GATE. The information shown there has actually been produced by three modules: a tokeniser module has produced the token annotations; a sentence-splitter module has determined sentence boundaries and added the sentence annotations and their associated constituent attributes; and a part-of-speech tagging module has added the part-of-speech attributes to the token annotations. These modules are shown in Figure 1. For the most part modules are components for building information extraction systems, systems designed to extract prespecified types of information from unstructured natural language text (such as newswire reports, journal articles, patents, e-mail, web pages, etc.) and place it into database-style structured representations, or 'templates'.

Most of these modules originated in the LaSIE system [6], our entry in the ARPA-sponsored MUC-6 information extraction system evaluation. These modules have been 're-used' in GATE by extracting them from LaSIE and writing simple wrappers around them to enforce communica-

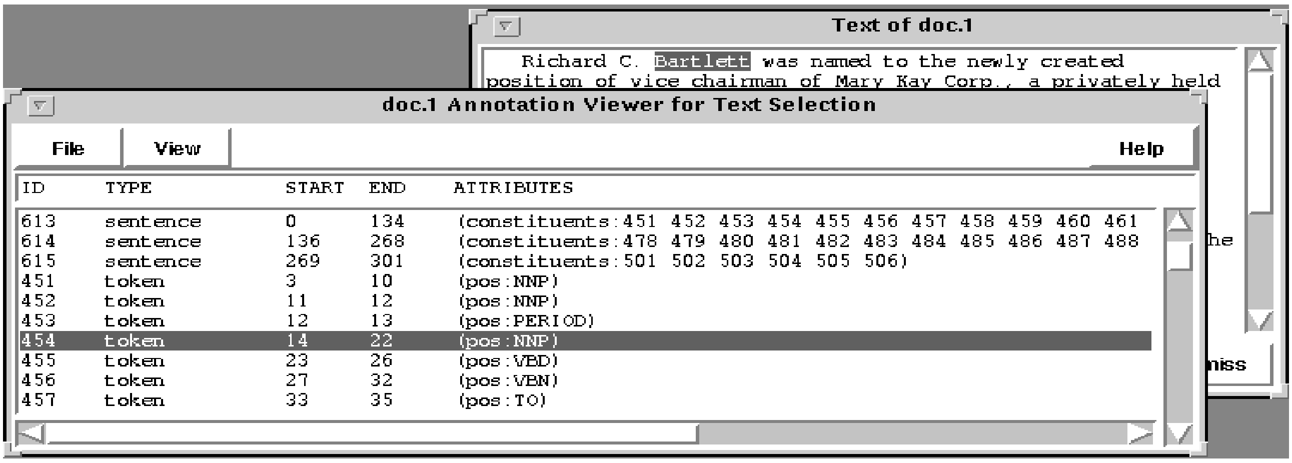

Figure 2. Some Annotations

tion through the GATE database.

GATE is a fully implemented system and is currently in use in a number of NLP research groups around the world. The GGI code is implemented in Tcl/Tk. GATE is implemented in C^{++}, however the modules may be written in any language. There are currently Perl, Prolog, Tcl, C, C^{++}, flex, and Common Lisp modules integrated into GATE.

3. Executable Graphs

A main purpose of the GGI is to allow execution of the modules within GATE. Section 3.1 describes the meaning of the primitives in the graph and how it is executed, Section 3.2 describes the method used to autogenerate the graph, and Section 3.3 discusses the method of creating manageable subgraphs.

3.1. Graph Syntax and Semantics

An example of a system graph is shown in Figure 1. A system graph is an executable graph, and is a simple data flow program. Modules are shown as nodes in the graph, with the data flow indicated by the arcs. Each incoming arc to a module indicates a dependency on results of previous processing. All modules at the source of arcs connecting to a dependent module must be run before the dependent module is executed, except where the incoming arcs are connected by lines, in which case the module requires the execution of only one of the modules at the other end of the arc (these arcs are then termed *or*-arcs). Thus, in the example graph of Figure 1, the `buChart Parser` module may only be run if the results of the `Gazetteer Lookup` module and either the `Tagged Morph` module or the `Morph` module are available. They in turn have earlier dependencies. The `Tokenizer` module has no dependencies and so begins execution. There are two modules with no downstream children: `MUC-6 Results` and `MUC-6 NE Results`, so either of these must produce an end result. However, because results from modules in the middle of the graph may be of interest to a NLP researcher, any module can be chosen as the final one that will be executed.

At any point in time, the state of execution of the system, or, more accurately, the availability of data from various modules, is depicted through colour-coding of the module boxes. Figure 1 shows a system window. Light grey modules (green, in the real display) can be executed. Modules that require input from others not yet executed, and so cannot be executed yet, are shown with a white background (amber, in reality). The modules that have already been executed are shown in dark grey (red).

The system graph can either be run in batch mode or in an interactive manner. To run in batch mode, the user selects a path though the graph and clicks on the final module. The current state of the graph, and the document (or collection of documents) currently undergoing execution is shown. The system ensures that the path chosen by the user is valid by only allowing a module to be selected if all its inputs have already been selected. Selected modules are executed in a data driven manner, with modules being executed as soon as their input data is available.

The interactive mode is designed for module developers. The modules under development can be executed as with the batch mode then the module or modules to be retried (after the underlying code or resources have been changed) can be reset by a mouse click. This clears the database of the postcondition annotations and allows the modules to be rerun.

The nature of the database (where each module produces a specific set of annotation types) means that it is possible to view partial results of execution without recourse to buffering intermediate data [19].

3.2. Autogeneration

The graph shown in Figure 1 is in fact the *custom graph*. This is the system graph that shows all the modules in the particular GATE environment. The custom window is automatically generated from the configuration information that is associated with each module, e.g., for the `buChart` module:

```
set creole_config(buchart) {
    title {buChart Parser}
    pre_conditions {
        document_attributes {language_english}
        annotations {token sentence morph lookup}
    }
    post_conditions {
        document_attributes {language_english}
        annotations {name syntax semantics}
    }
    viewers {
        {name single_span}
        {syntax tree}
        {semantics raw}
    }
}
```

The autogeneration algorithm creates data flow arcs from modules that have an annotation type in their postconditions to the other modules that have the same annotation type in their precondition. For example, `Gazetteer Lookup` has the annotation type `lookup` in its postconditions, so an arc connects it with `buChart Parser`, which has that annotation type in its preconditions. Arcs are not created between modules that operate on different languages, however in Figure 1, all the modules operate on English language documents. When more than one module has the same annotation type in its postcondition then it is assumed that either module may produce the required result, and so the two arcs are *or*-arcs and are connected by a line (both `Morph` and `Tagged Morph` produce the same annotation and so have *or*-arcs into `buChart Parser`).

The most computationally expensive part of autogeneration goes into discarding redundant arcs. Redundant arcs are those that connect an upstream module to a downstream module where it can be deduced that the preconditions of modules between the two given modules cover the annotation types that the arc represents. For example, the `Tokenizer` produces annotation types required by `buChart Parser`, but there is no need for a data flow arc between these modules as modules between them also require these annotation types.

The autogeneration facility allows easy integration of new modules into the GGI. Most NLP tasks can be expressed in the simple data flow techniques of this system, but it is currently not possible to integrate NLP tasks that require iteration.

Some modules have the same annotation type in both pre- and postconditions. These modify the result of previous computation and pass the data flow down stream. This kind of module, termed a *filter*, cannot be automatically positioned in the diagram, instead the user selects the position of filters from the arcs on which they may appear (arcs from modules that produce the annotation type the filter operates on). During execution filters are treated as normal modules.

3.3. Customising Graphs

The system graphs are displayed with the DAWG tool [14]. This is also used in the tree based visualisation tools described in Section 4. DAWG allows commands to be associated with nodes, hence it can be used for data flow graphs. It has a layout algorithm based on the method used by daVinci [4] to minimise arc crossing.

GGI suffers from the scaling problem [2], as the size of the custom graph quickly becomes unmanageable. This can be alleviated by creating new system graphs from specified subgraphs of the custom graph.

It is possible to group these derived system graphs together so that the user may chose from a selection of tasks at the top level of the GGI (not shown here for space reasons). Having chosen a task (e.g. parsing), an intermediate level display appears, presenting the user with a selection of icons, one for each of the one or more specific systems capable of performing the selected task (e.g. the buChart parser or the Plink parser). Once a particular system is selected, a final window appears displaying the appropriate system graph.

4. Visualisation of NLP Data

NLP data is wide ranging in scope but has specific characteristics that mean the problems with visualising large amounts of data [2] are less significant. This is because either the information is visualised as coloured markup on the text (meaning that the text can be displayed using traditional textual techniques [12]), or the information is grouped over small segments of text, such as paragraphs or sentences.

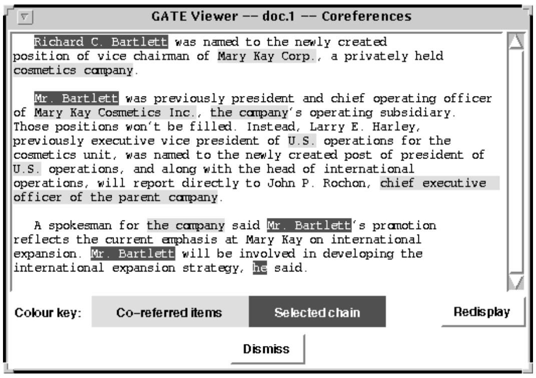

Figure 3. Multiple Span Viewer

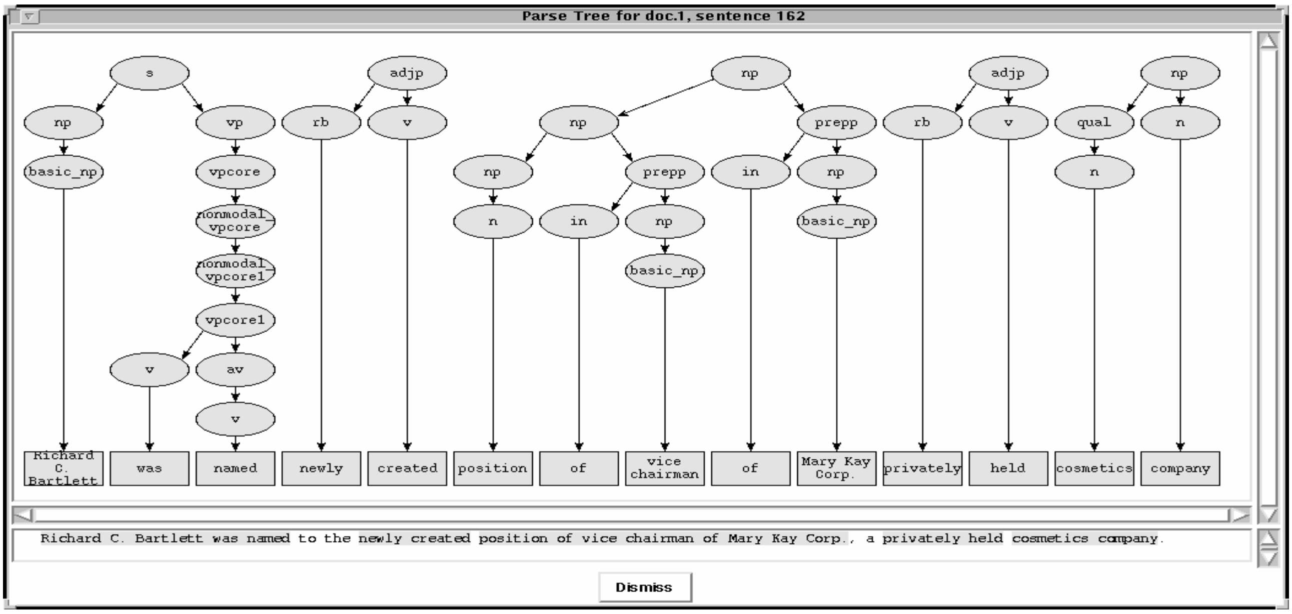

Figure 4. Tree Viewer

GGI has several viewers for the display of annotations. The viewer for each postcondition annotation is specified by the module configuration file, an example of which is given in Section 3.2. The viewers can be classified into those which display the text and overlay the annotations as colours or shades ('single span', 'multiple span', 'text-attribute'); and those that visualise a more complex relationship between annotations in an acyclic graph format ('tree'). Where no viewer is specified, a default annotation dump is displayed (similar to that of Figure 2). The configuration file for the `buChart Parser` module in Section 3.2 specifies that the 'name' annotation type is assigned the 'single span' viewer, 'syntax' the 'tree' viewer, and 'semantics' the 'raw' or annotation dump viewer. New viewers can be written where the default ones are not appropriate for new annotation types.

The 'single span' and 'text-attribute' viewers are fairly simple, assigning different colours to each annotation. 'multiple span' is more complex, as it is designed to view annotation chains. An annotation chain is a list of annotations specified by annotation references. The user chooses a highlighted part of the text, and all the other highlights that are part of the same chain are displayed. Figure 3 shows this viewer displaying the results of a coreference task. Coreference identifies elements of the text that are interpreted as referring to the same real world entity. For example, a person and a pronoun might be coreferential. In Figure 3 the user has chosen one of the highlights referring to 'Richard Bartlett'.

The 'tree' viewer containing 'syntax' annotations (produced by the `buChart Parser`) is shown in Figure 4. The parse trees currently integrated into GATE span at most a sentence, so that the tree size is always manageable.

The viewers are activated by first clicking with the mouse on the module in the system graph which reveals a menu of annotations, choosing an annotation brings up the appropriate viewer.

There is a certain amount of connectivity between these viewers, as it is possible to click on a node in the parse tree and have the area of text highlighted in a text display window, or it is possible to highlight areas of text and display the raw annotations that are contained within the highlighted span.

5. Concluding Remarks

We have described a tool called GGI that supports the visual execution of NLP systems that consist of multiple interdependent modules. The execution graphs in GGI are autogenerated from declaratively stated input and output specifications of the component modules, a feature which makes it easy to integrate new modules. The graphs are also customisable, permitting a user to define straightforwardly a new subgraph of interest. GGI also includes a range of tools for visualising the complex annotations that NLP modules may produce as a result of analysing texts.

GGI is proving to be an invaluable tool for the rapid development and integration of new NLP modules into a larger application. Feedback from users of the system has indicated that the graph-based execution model is appealing and that the visualisation tools are a great aid to researchers.

Further work will be driven by user feedback. Increasing the visual content of the system graph is a possibility. The discrete nature of the tasks would make iconic nodes a potentially useful addition because icons could ne used to group nodes that perform particular tasks, such as parsing or tagging. The labelling of arcs with the annotations that flow along them is also a possible future feature.

The data flow execution method presented here covers many NLP applications and allows the modules that form the nodes of the graph to be placed automatically. However, there are tasks that require more expressivity. The field of multilinguistic text analysis involves deciding what language a given document is written in. Allowing this sort of module would require the addition of distributor primitives. The current batch mode of execution might have to be modified because it may not be practical to choose a path through a graph when the path may branch.

Allowing iteration within the system graph would enable the NLP modules to be finer grained, and could allow NLP algorithms to be encoded with the data flow model. This would have a profound effect on the GGI model as presented here, complicating considerably the autogeneration process. The problems of procedural abstraction and the visual display of large graphs [7] would also have to be considered.

Finally, it is worth mentioning that there are other potential applications of visual languages to NLP. In particular, a visual approach to parsing seems promising as both the connection between parsing and graph grammars [11], and between graph grammars and visual languages [15, 16] has already been made. Another application area concerns graph-based semantic network representations which are widely used for knowledge representation by NLP systems [17]. Visual languages that examine the structure of graphs [3] could be used when manipulating such data.

Acknowledgements

The research reported here has been supported by a grant from the U.K. Engineering and Physical Science Research Council (Grant # GR/K25267).

References

[1] G. Bono and P. Ficorilli. Natural Language Restatement of Queries Expressed in a Graphical Language. In *ER'92-11th International Conference on the Entity-Relational Approach. LNCS 645*, pages 357–274. Springer-Verlag, 1992.

[2] M. Burnett, M. Baker, C. Bohus, P. Carlson, S. Yang, and van Zee P. Scaling Up Visual Languages. *IEEE Computer*, 28(3):45–54, 1987.

[3] I. Cruz, A. Mendelzon, and P. Wood. G+: Recursive Queries without Recursion. In *Proceedings of the 2nd Expert Database Systems Conference*, pages 645–666. Benjamin-Cummings, 1989.

[4] M. Fröhlich and M. Werner. Demonstration of the Graph Visualization System daVinci. In *Proceedings of DIMACS Workshop on Graph Drawing '94, LNCS 894*. Springer-Verlag, 1995.

[5] R. Gaizauskas, H. Cunningham, Y. Wilks, P. Rodgers, and K. Humphreys. GATE – an Environment to Support Research and Development in Natural Language Engineering. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-96)*, Toulouse, France, Oct. 1996.

[6] R. Gaizauskas, T. Wakao, K. Humphreys, H. Cunningham, and Y. Wilks. Description of the LaSIE system as used for MUC-6. In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*. Morgan Kaufmann, 1995.

[7] M. Gorlick and A. Quilici. Visual Programming-in-the-Large versus Visual Programming-in-the-Small. In *Proceedings VL'94 Tenth Annual IEEE Conference on Visual Languages, St.Louis*. IEEE Computer Society Press, 1994.

[8] R. Grishman. TIPSTER Architecture Design Document Version 2.2. Technical report, DARPA, 1996. Available at `http://www.tipster.org/`.

[9] P. Haeberli. ConMan: A Visual Programming Language for Interactive Graphics. *ACM Computer Graphics*, 22(4):103–111, 1988. ACM SIGGRAPH '88.

[10] D. Hils. Visual Languages and Computing Survey: Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing*, pages 69–101, 1992.

[11] E. Hyvönen. Graph Grammar Approach to Natural Language Parsing and Understanding. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI'83)*, volume 2, pages 671–674. Morgan Kaufmann, 1983.

[12] D. Jonassen, editor. *The Technology of Text*. Educational Technology Publications, 1982.

[13] J. Landauer and M. Hirakawa. Visual AWK: A Model for Text Processing by Demonstration. In *Proceedings VL'95 11th International IEEE Symposium on Visual Languages, Darmstadt*. IEEE Computer Society Press, 1995.

[14] P. Rodgers. DAWG (Displaying Annotations With Graphs) Developers Guide. Technical report, Department of Computer Science, University of Sheffield, 1997.

[15] P. Rodgers and P. King. A Graph Rewriting Visual Language for Database Programming. *The Journal of Visual Languages and Computing*, In press.

[16] A. Schürr, A. Winter, and A. Zündorf. Visual Programming with Graph Rewriting Systems. In *Proceedings VL'95 11th International IEEE Symposium on Visual Languages, Darmstadt*. IEEE Computer Society Press, 1995.

[17] J. Sowa, editor. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, San Mateo, CA, 1991.

[18] S. Steinman and K. Carver. *Visual Programming With Prograph CPX*. Manning Publication, 1996.

[19] A. Woodruff and M. Stonebreaker. Buffering of Intermediate Results in Dataflow Diagrams. In *Proceedings VL'95 11th International IEEE Symposium on Visual Languages, Darmstadt*. IEEE Computer Society Press, 1995.