Graphical Representations and Software Engineering

Robert Gabriel Stefan Jähnichen
GMD FIRST (German National Research Center for Computer Science)
Hardenbergplatz 2, W - 1000 Berlin 12, GERMANY

Abstract

This paper reports on ongoing activities to generate graphical representation manipulating user interfaces from very high level specifications. First, an overview of graphical notations that are relevant for software engineering is given. Second, the capabilities offered by the current version of the generator G^2F [4] are presented. Third, the paper shows how software technology techniques in turn have been applied to extend G^2F by the power of attribute grammars.

1 Examples of graphical representations

Graphical representations of information play an important role in software engineering. They help to display the underlying data in a user-friendly way. Many of the examples for kinds of data that lend themselves naturally to graphical presentation come from the domain of parallel and distributed programming:

- Multi-processor systems: graphical representations display processor connections, and process scheduling and communication[14].
- Networks: gauges and meters can be used as graphical abstractions of the state of computer networks [2].
- Petri Net Animation: graphical representation and animation techniques can be used to display the evaluation of Petri networks[11] [17].
- Process control applications: the structure and state of systems (e.g. a clinical neurophysiological monitoring system[3]) can be visualized and animated.

Examples more related to sequential programming are

- Program Structure: the module structure and use dependencies of large programs can be represented graphically[16].
- Program debugging environments: these show graphical views of the program data state, showing the modifications in the data state as the program runs [18][20][12].

Further examples for the use of graphical representations in software engineering come from the area of formal program development. Formal program development comprises the complete specification of the properties of programs, the application of formal methods like VDM or Z [13,24], and the conduction of complete proofs of conditions that have to be met during the process of software construction. Latest developments are meta-calculi designed for the expression of all these different kinds of formal objects in a uniform language. An example is the meta-calculus Deva [22,23] developed in the ESPRIT project 510 ToolUse [8].

2 G^2F, a generator of graphical user interfaces

G^2F is a generator of two-dimensional formulae manipulating structure-oriented editors. These editors can be used as user interfaces to formal systems by defining a set of interaction patterns [7]. Examples for formal systems where a tool like G^2F can be helpful are algebra systems, theorem provers or formal software development environments.

The construction of a generic tool like G^2F is easily motivated. First, graphical presentation of information is often easier to grasp and more adequate for human beings. Second, two-dimensional representations can help to simplify both use and understanding

0-8186-2755-7/92 $03.00 © 1992 IEEE

of formalisms. Third, there exist many formal frameworks which could be improved by user interfaces supporting framework-specific notations. And, last but not least, hand-coding of graphical interfaces is very time-consuming [19].

The most important design decision taken concerns the way the layout of two-dimensional formulae is specified. G^2F facilitates direct manipulation style definitions: first, a prototype of the envisaged formula is drawn. Then a couple of layout modes are selected [4]. This information is used to derive layout constraints for the formula drawn. A detailled description of this derivation process is in [5].

For the definition of the complete structure editor all graphical definitions are structured in a context-free grammar like style. Arguments of formulae correspond to nonterminals and determine the class of graphical objects which are allowed to be substituted for a particular argument in the structure editor.

Figure 1 shows what the definition of the layout of a quotient looks like. One of the layout modes used is *corner-relative* addressing. The class of admissible formulae as numerator is *Formula*. The little arrows indicate that the arguments shall be centered against one another.

Every graphical object edited is implicitely tree structured according to the formula grammar defined. In addition a unique textual representation satisfying the LALR(1) condition corresponds to every editable formula. This feature forms the basis for using the editors as interfaces to formal systems. It also provides an alternative to structure-oriented editing, because the textual equivalent of a formula can be inserted as well.

The deduced layout constraints are used to generate C procedures for drawing formulae, calculating their dimensions and constructing various kinds of menues in the resulting editor. All grammar information is handled by scanner and parser generators also producing C-code. The generated procedures are compiled and linked together with code common to all editors, which realizes standard operations of structure editors for edition and search commands, for instance.

The current version of G^2F uses the X-window system 11.3 and runs on UNIX workstations.

3 Mapping G^2F definitions on attribute grammars

This section is dedicated to the description of the mapping of the G^2F direct manipulation style definitions on attribute grammars. The reader also gets an impression of the specification language used in the tools Ast [10] and Ag [9]. The resulting attribution rules are modular and still relatively easy to understand. These two properties are very important when the extension of the generated definitions or their modification is concerned.

The generated attribute grammar consists of

- rules defining the abstract syntax of an application
- attributes describing offset and dimension of an object in the editor, and
- computation rules for these attributes

The following example shows the grammar generated for a nonterminal *formula* with the three right hand sides *sum, product* and *quotient.*

```
formula = <

    sum = son1:formula
    son2:formula .
    product = son1:formula
    son2:formula .
    quotient = son1:formula
    son2:formula .

> .
```

The syntax exploits the analogy of nonterminals (*formula*) to types. It gives additional names (*son1, son2*) to nonterminals, which are analogous to variables in functions. The identifiers (*sum, product, quotient*) associated with each rule are the function names in this analogy. Hence, the structure of an abstract syntax tree can be described by nesting the functions the signature of which is implicitely defined in the example.

In terms of abstract syntax trees the example defines three alternatives for a node with type *formula*. Each of these alternatives has two children *son1* and *son2* with nodetype *formula*.

The generation of the computation rules for the attributes describing the editor layout is triggered by the direct manipulation style definition made: the defined lines, linebreaks, indentations and layout modes determine the computation of these attributes.

We only give a flavour of the attribution rules generated. The rules are the ones resulting from the definition of the layout of a quotient shown in figure 1. They define how the the attributes *offsetx, offsety,*

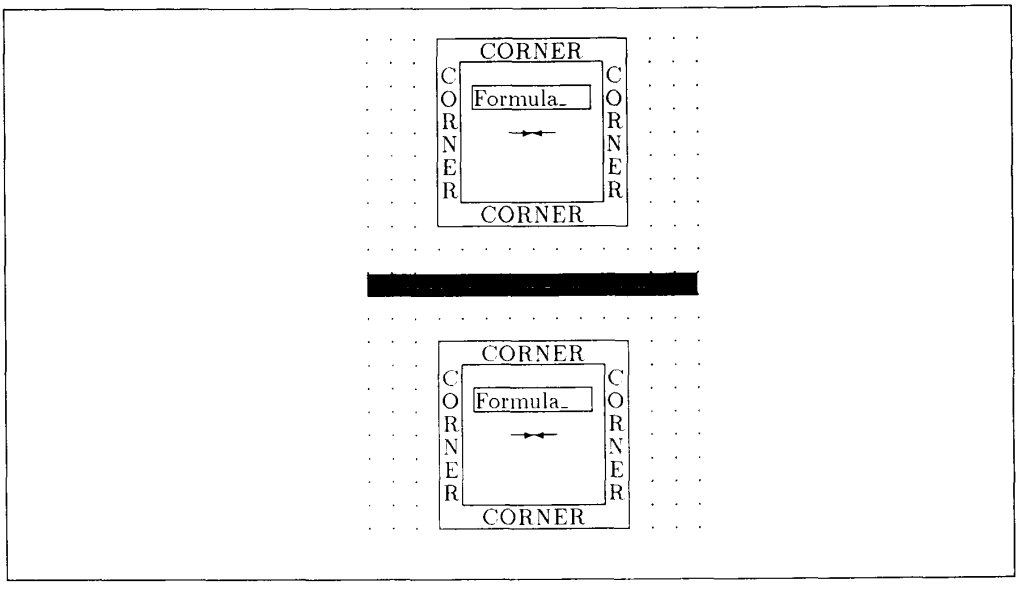

Figure 1: The specification of the layout of quotients with G^2F

```
son1:offx := offx + 1 + (max(son1:dimx,son2:dimx) - son1:dimx)/2;
son2:offx := offx + 1 + (max(son1:dimx,son2:dimx) - son2:dimx)/2;
son1:offy := offy;
son2:offy := son1:offy + son1:dimy + 5;
dimx := max(son1:dimx,son2:dimx) + 6;
dimy := son1:dimy + son2:dimy + 5;
```

Figure 2: Generated rules for calculating attributes

dimx and *dimy* of a node with type *quotient* have to be computed.

The first two rules (see figure 2), e.g., express the constraint that the arguments are centered against one another. The x-coordinate of the left-upper corner of the numerator (attribute *son1:offx*) depends on the offset of the overall formula (attribute *offx*) and the dimensions of the arguments. While the attributes containing the overall offset are inherited, the attributes providing the dimensions of the arguments are synthesized. This design decision reflects the limitation of the expressive power of the G^2F direct manipulation style definitions.

The drawing of text and lines in the generated editor is not reflected in the attribution rules generated as it was first planned and suggested in [6]. For reasons of clarity a separate procedure is generated, instead, which computes the coordinates of lines and text squares in dependency of the current attribute values in the structure tree.

We only wanted to give an impression of the resulting attribute grammar. A complete and much more detailled description of the mapping and the overall work is in [1].

4 Two-layered generation

This section shows how G^2F can be used to benefit from the combination of direct manipulation style definitions and attribute grammars in the construction of user interfaces.

Figure 1 already gave an example of the specification of a graphical notation.

As described in the foregoing section, modules of attribute grammar specifications in the Ast/Ag input language are generated from a set of definitions made with the extended G^2F. The entirety of these modules is then processed by Ast and Ag to provide the corresponding attribute evaluation and tree manipulation routines. The generated C-procedures are compiled and linked with standard code common to all generated editors. Figure 3 summarizes these explanations of the generation process.

The two-layered-ness of the generation process enables the extension or modification of the modules generated on the level of attribute grammars.

Additional modules of attribution rules could be added to define context sensitive properties of an application. The usual examples from language-based environments are rules for scope and name analysis, operator identification or rules for type checking and type coercions.

Modifications will help to overcome limitations of the G^2F direct manipulation style definitions. One example for such a change is the definition of the layout of indexed variables in mathematical applications: for the time being it has been impossible to specify in the drawing that the size of the index should be smaller than the size of the variable.

5 Conclusions and perspectives

We showed the relevance of a great variety of graphical notations to the domain of software engineering and to distributed and parallel computing in particular.

These observations have been the driving forces for the construction of the generator of user interfaces described in this paper. We presented its design, implementation and use.

G^2F multiplies the advantages of direct-manipulation style definitions by the power of attribute grammars: graphical notations can be defined in a very easy way, more or less by drawing examples. The attribute grammar automatically generated from these definitions is modular and still comprehensible. Thus, additional modules of attribute grammars can easily be added.

Another possibility is the modification of the layout rules generated. This is a way to overcome current limitations in the expressive power of the actual direct-manipulation style definitions of G^2F.

The extended G^2F system uses all compiler front end construction tools developed at the GMD in Karlsruhe. It is implemented in C, uses the X-Window system 11.4 and runs under the UNIX operating system.

Future work will primarily deal with improvements of the G^2F direct manipulation style definitions. The extensions will be driven by the requirements resulting from the application of G^2F in formal program development environments like the one existing for Deva [8].

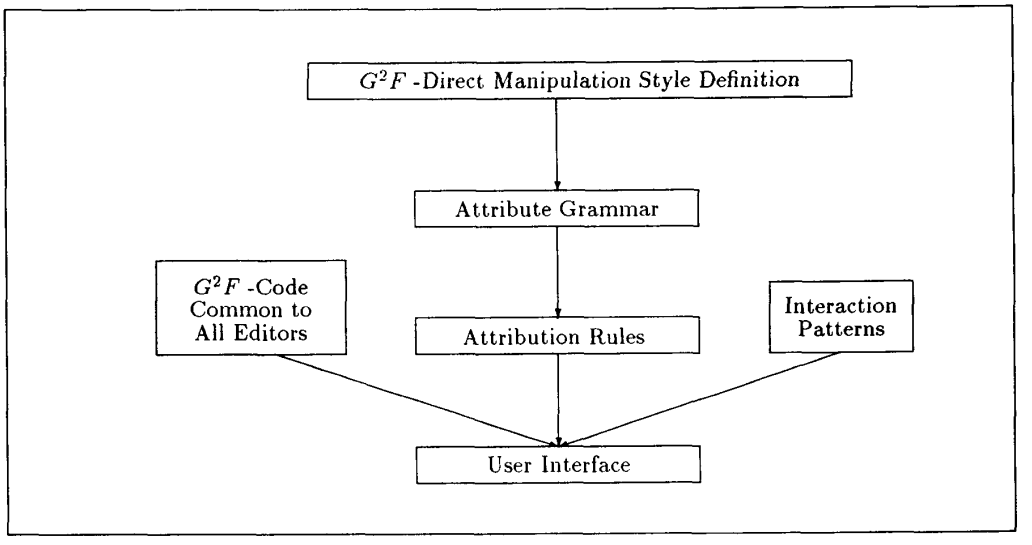

Figure 3: Overview of the generation process in G^2F

References

[1] M. Besser. Die Kombination von Metawerkzeugen zur Erzeugung strukturorientierter Editoren für Text und Graphik, December 1990.

[2] E.J. Cameron, B. Gopinath, P. Metzger, and T. Reingold. Infoprobe - a utility for the animation of ic* programs. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.

[3] T. F. Collura, E. C. Jacobs, R. C. Burgess, and G. H. Clemm. User-interface design for a clinial neurophysiological intensive monitoring system. In *Human Factors in Computing Systems, CHI 1989 Proceedings*, pages 363–368, May 1989.

[4] R. Gabriel. The automatic generation of graphical user-interfaces. In *System design: concepts, methods and tools*, pages 330–339. IEEE Computer Society Press, April 1988.

[5] R. Gabriel. A formalism for the definition of graphical formulas. In *ACM SIGSMALL symposium on personal computers*, pages 28–36. Association for Computing Machinery, May 1988.

[6] R. Gabriel. Structured definition of graphical layouts. In *20. GI-Jahrestagung*, pages 362–370. Informatik-Fachberichte 222, Springer Verlag, October 1989.

[7] R. Gabriel and R. Bock. *G^2F User Manual, Version 1.0.* Gesellschaft für Mathematik und Datenverarbeitung, March 1990.

[8] R. Gabriel and S. Jähnichen. ToolUse: a uniform approach to formal program development. *Technique et Science Informatiques*, 9(2):166–174, 1990.

[9] J. Grosch. AG - an attribute evaluator generator. Technical report, Gesellschaft für Mathematik und Datenverarbeitung, 1989.

[10] J. Grosch. AST - a generator for abstract syntax trees. Technical report, Gesellschaft für Mathematik und Datenverarbeitung, 1989.

[11] G.S. Hura, M.A. Costarella, C.G. Buell, and M.M. Cvetanovic. PNSOFT: A menu-driven software package for petri-net modelling and analysis. In *IEEE 1988 International Conference on Computer Languages*, pages 41–47, 1988.

[12] S. Isoda, T. Shimomura, and Y. Ono. VIPS: A visual debugger. *IEEE Software*, 8(3):8–19, March 1987.

[13] C. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1986.

[14] C. Kilpatrick, K. Schwan, and D. Ogle. Using languages for capture, analysis and display of performance information for parallel and distributed

applications. In *IEEE 1990 International Conference on Computer Languages*, pages 180–189, March 1990.

[15] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[16] Mark Moriconi and Dwight F. Hare. Pegasys: A system for graphical explanation of program designs. In *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 148–160, July 1985.

[17] B. Müller and G.-R. Friedrich. Using petri nets in a test environment for fms control software. In *Proceedings of the VIIth Annual Bilateral Workshop on Information in Manufacturing Automation (GDR-Italy)*, 1989.

[18] B. A. Myers. Incense: a system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[19] B.A. Myers. Creating user interfaces by demonstration. Technical Report CSRI-196, Computer Systems Research Institute Toronto, May 1987.

[20] S. P. Reiss and J. N. Pato. Displaying programs and data structures. In *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, pages 391–401, January 1987.

[21] B. Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, August 1983.

[22] M. Sintzoff, Ph. de Groote, M. Weber, and Jacques Cazin. Definition 1.1 of the generic development language DEVA. Technical report, Université Catholique de Louvain, Belgium, December 1989.

[23] M. Weber. *A Meta-Calculus for Formal System Development.* PhD thesis, University of Karlsruhe, 1990.

[24] J.P. Woodcock. Using Z. Technical report, Programming Research Group, Oxford University Computing Laboratory, 1988.