A Constraint-Based Declarative Display Description Language for Visualization

by

Anthony J. Kusalik and **Gregory M. Oster**

Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan
Canada S7N 0W0

Abstract

An ongoing research project at the University of Saskatchewan is WAND, the design and realization of generally applicable visualization systems to aid in the monitoring and debugging of logic programs. The goal is systems which allow the rapid prototyping of graphics-based tracers for different (logic programming) languages, language implementations, and execution paradigms. The first phase of the project, the GLP system, has been completed, evaluated, and reported on earlier. It demonstrated the various merits of WAND. More recent work has been on a successor to GLP, incorporating ideas from the fields of logic grammars and constraint-based logic programming. A key component of this new system is the facility for describing the image which is to appear on the user's screen — for specifying what to display and how to display it. A constraint-based declarative display description language has been developed and implemented as part of this facility. This presentation describes this language and its current implementation, the ICOLA system, as well as outlining the overall project. A highlight of the language is its expressivity — the ease with which pictures can be described. As well, it does not require absolute positioning and permits pictures to be under-constrained.

1. Introduction

There is an unfilled need in logic programming for graphics-based visualization tools for program monitoring and debugging. Many systems still rely on a text-based user interface; for example, extensions of the 4-port box model [Byrd80]. Unfortunately, this type of model presents a second computation paradigm to users, which is often a source of great confusion. Also, a textual interface does not take advantage of the power and ease of a graphical one.

Some systems and implementations do provide visual execution tracing tools, for example DEWLAP [DeCl86], WamTrace [DiLu87], TPM [EiBr88], MUST [SvSu90], VISTA [Tick92], and VisAndOr [Car+92]. The characteristics of such tools are typically preset, and not readily changed. The chosen information content, display style, set of representational forms, or suite of features may be insufficient or inappropriate for a particular type of user, user's objective, or program's behavior. Also, the visualization facilities cannot be used with another language system.

The lack of graphics-based visualization tools is particularly acute in the case of new languages and implementations. Adding visual tracing or debugging capabilities to a language system is typically difficult and time-consuming. Hence, it is usually undertaken only after the language is stable, and characteristics of the interface can all be set. However, visualization tools are often highly desirable earlier, during stages of language design and initial implementation. Then they can be used to analyze, correct, and modify language constructs, language semantics, implementation code, and end-programs.

There has been little research into what makes a good visualization aid for logic-programming languages. It is not known, for example, what facilities and forms are best suited for different types of sequential language, OR-parallel languages, committed-choice languages with dataflow, constraint-based languages, etc. One reason for this is because of the difficulty and effort necessary in devising multiple interfaces having a variety of features and forms.

A solution to this combination of problems is provided by the WAND system design. WAND aims to provide generic logic programming monitoring tools, suitable for developing specific tracing or debugging systems. The tools are to be independent of a particular (logic programming) language or implementation. Customization is to be realized by high-level mechanisms. Effort necessary to achieve a specific visualization tool must be of the same order-of-magnitude as that necessary to write a meta-interpreter for the target language. Resulting visualization systems are to be reasonably efficient, showing efficiency expected for a prototype system.

Characteristics of the WAND design are as follows. The user interface is provided by a generic, stand-alone, graphics-based visualization program (tool). To create a custom tracer or debugger only three additional components need be supplied: a meta-interpreter (or equivalent) producing "raw" trace data; a transformation specification mapping the trace information into database updates; and a configuration specification controlling what information is displayed and the form that information takes. These three components are typically much simpler to compose than an entire visualization subsystem. By altering them, the models of execution and visualization, and the capabilities provided to a user, can all be tailored. Monitoring can be concurrent with program execution, or performed off-line, driven by a data-file.

The WAND design maximizes flexibility and power, while minimizing complexity and programming effort. Appendix A illustrates the design architecture, and indicates the potentials for tailorability.

A specific instance of the WAND design, GLP, has been developed and reported on earlier [KuOs93]. GLP provides a macro-based transformation language and a functional configuration specification language. It was written in the C programming language, and uses X-Windows user interface capabilities. Several different tracers for Prolog have been developed using GLP. These include tracers providing OR-tree, AND-tree, and AND/OR-tree views (models) of a Prolog computation [KuOs93]. The working tracers substantiate the design of GLP, and show that the goals of WAND can be satisfied. Two of the important lessons learnt from the GLP effort are that absolute positioning capabilities are not necessary, and that a simple inference system within the visualization system is useful.

Unfortunately, deficiencies in the type scheme of GLP's configuration specification language were discovered, as well as problems with the implementation. Instead of "fixing" the old system, however, it was decided to devise a new one based on the knowledge and experience gained. A different paradigm was chosen for the configuration specification language, one similar to production rules and logic grammars (e.g. DCGs). The remainder of this paper describes GLP's successor, in particular the constraint-based declarative graphics language to be used in implementing the new specification language. Section 2 outlines the structure of the Phase II system. The declarative graphics language in discussed in Section 3, and examples given. Section 4 describes the current implementation of this graphics language. Conclusions and areas of further work are given in Section 5.

2. Phase II

The framework for the next system within the WAND scheme is as follows. The Phase II system is being implemented using Prolog with graphics extensions. Database and inference capabilities can thus be provided by the underlying Prolog system. The configuration specification language is a "display description grammar", in the style of DCGs. While the basic structure for organizing text, the sequence, is adequate for DCGs, it is not sufficient for two-dimensional input or output. One mechanism for describing 2-dimensional layout is a combination of pictoral attributes and positional constraints [HeMa91]. The configuration specification language devised for Phase II begins with the capabilities and syntax of DCGs, but extends them with constraints and attributes.

2.1 The Picture Grammar

In the constraint-based, display description grammar for Phase II, terminals are basic graphical objects which can be drawn by the underlying system. Non-terminals are graphical objects defined in terms of terminals, non-terminals, positional constraints, and attributes. Labels are used to refer to objects (terminals or non-terminals). The grammar rules are used "backwards" — i.e. as production rules — with the arguments already instantiated based on the information regarding the execution already in the tool's database. Constraints, attributes, and labels can appear on either side of the production symbol.

An example of a rule for specifying an AND-subtree in a trace is given in Figure 1.

```
and_tree( t( Parent, Children, Clause ) ) ==>
        and_node( Parent ) <N>,
        collect(   Child,
                   ( and_edge( Clause ) <E>,
                     and_tree( Child ) <St>,
                     where E connects N to St ),
                   Children ) <C>,
        where horizontally_aligned C.
```

Figure 1: Production rule in the display description grammar

This states that the "AND-tree" (or subtree) corresponding to the execution state described by t(Parent, Children, Clause) is

- an "AND-node" showing the information in Parent, plus
- a collection of AND-subtrees, each corresponding to an element in Children,

where

- an "AND-edge" showing the information in Clause joins each AND-subtree to the parent AND-node, and
- all the subtrees are horizontally aligned.

There are many ways to implement a grammar such as the one above. Unfortunately, there may be a significant semantic gap between the grammar and the graphics extensions of the Prolog system used for the implementation. One way to ease the task of implementation is to map from the display description grammar to some intermediary language. This is the approach followed in Phase II of WAND. It is shown diagrammatically in Figure 2.

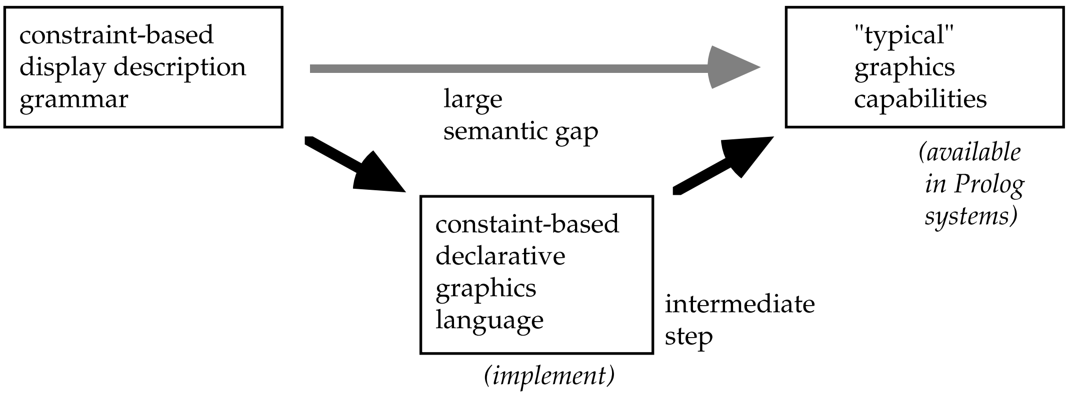


Figure 2: Implementation scheme for declarative display grammar

3. Declarative Graphics Language and ICOLA

As part of the Phase II effort, a constraint-based graphics language has been defined. This language is being used to implement the aforementioned constraint-based display description grammar (the configuration specification language for Phase II). This section gives an overview of the graphics language.

The most important characteristic of the new graphics language is that it is a declarative language. A user describes the contents of a display or picture using a set of simple statements or facts ("unit clauses") regarding it. A statement can declare the existence of a graphical object, a constraint between two objects, or an attribute of an object. For example,

```
object( root_node, oval ).
```

declares that an *oval* object, identified by the name root_node, exists in the picture.

```
constraint( above, [parent_node, child_node ] ).
```

is a constraint stipulating that the object identified as parent_node is above (in relative position) the object child_node. The fact that object halt_node has colour *red* is specified by an attribute

attribute(halt_node, color, red) .

The following are defined as basic objects (primitives) for the language: *box*, *oval*, *point*, *triangle*, *line*, *arc*, *text*, *bitmap* (an arbitrary image as a bitmap), and *complex*. Most objects are able to "stretch" to accommodate the constraints placed upon them. The exceptions are the fixed-sized objects, *point* and *bitmap*. Stretchable objects have a preset default size. A simplifying assumption for the implementation is that all objects except *line* and *arc* have a rectangular "bounding box" around them. This sometimes causes minor aesthetic problems with the display of objects such as triangles and ovals. For example, a connecting line does not extend completely to the edge of an oval.

Positioning is currently restricted to two dimensions, and constraints mainly deal with horizontal and vertical positional relationships. Defined constraints are of two types: *fundamental* and *aliased*. The fundamental constraints supported are: *above*, *left_of*, *touching*, *aligned*, *horizontal_distance*, *vertical_distance*, and *connects*. Aliased constraints are expressible in terms of one or more fundamental constraints. Examples of aliased constraints are *below* and *right_of*, which have simple duals, and *inside* which can be expressed as a set of *above* and *left_of* constraints. The transformation of aliased constraints is performed automatically by the system.

The language is typed in the sense that not all attributes are applicable to all types of objects. Also, a constraint may involve only certain kinds of objects. For example, the first item in a *connects* constraint must be an object of type *line*. A *text* object cannot be the second item of an *inside* constraint; i.e. another object cannot be "inside" some text.

To provide more precise control in positional relationships, "selector functions" are available. These can be applied to any object with a "bounding box", and specify that the top, bottom, left, or right of the object is to be involved in a constraint. For example,

constraint(aligned, left(node_A), right(failure_mark)).

specifies that the left edge of the object node_A is to be vertically aligned with the right edge of failure_mark. The selector functions are: *left()*, *right()*, *top()*, and *bottom()*.

Horizontal and vertical selectors can sometimes be inferred by the system, making description of a picture more convenient for the user. For example, the constraint

constraint(above, [box1, box2]).

intuitively means that, unless told otherwise, all of box1 is above box2. The latter is expressed by

constraint(above, [bottom(box1), top(box2)]).

Thus a system of defaults can be, and is, provided. In a preprocessing step the system applies such defaults.

The attributes currently defined in the language are: *colour*, *visible*, *style*, and *directed*. Obviously the last two attributes are only applicable to *line* and *arc* objects.

Specification of structured or hierarchical objects is facilitated by *complex* objects. A group of single objects can be declared to constitute one *complex* object. Any constraint involving, or attribute applying to, the *complex* object is taken as involving, or being applied to, each constituent object. In addition to its intended use, this feature can be used to abbreviate specifications.

The graphics language also includes some control directives. For example, *reset* causes the engine processing the picture description to reset its internal data structures. *Reset* should thus occur only at the beginning of a picture description. These directives can be ignored for understanding the described picture.

The language also permits under-specification of a picture. The implementation deals with this by applying defaults and heuristics. Typically, an infinite number of pictures satisfy a given picture description involving only relative layout specifications. ICOLA's heuristics and defaults are designed to generate an aesthetically pleasing final result in a majority of cases. However, output which the user may not have expected — but which is nonetheless "correct" in terms of the given specification — may still occasionally arise.

3.1 Example

Suppose we wish the following picture:

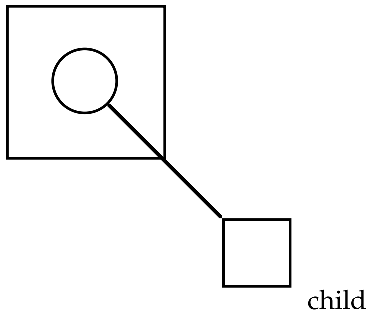


Figure 3: Sample picture

This can be specified using the picture description in Figure 4. How ICOLA renders the picture is shown later, in Appendix C.

```
object( parent, box ).
object( child, box ).
object( derivation_line, line ).
object( parent_component, oval ).
object( child_descr, text, “child” ).
constraint( inside, [ parent_component, parent ] ).
constraint( connects, [ derivation_line, parent_component, child ] ).
constraint( below, [ child, parent ] ).
constraint( right_of, [ child, parent ] ).
constraint( left_of, [ child, child_descr ] ).
constraint( aligned, [ bottom( child ), top( child_descr ) ] ).
```

Figure 4: Specification of picture in Figure 3.

The meaning of each statement in this specification should be intuitively obvious given the previous discussion and the picture in Figure 3.

4. Implementation

ICOLA, an engine able to process the declarative graphics language described in Section 3, has been designed and implemented. Appendix B graphically shows its high-level decomposition, and how the implementation fits the WAND architecture. The Prolog system used is SICStus Prolog V2.1 with GM graphics extensions. The constraint solver is a separate, linked module, written in C.

The Prolog components of ICOLA are responsible for preprocessing, postprocessing, user interface management, and overall control. The preprocessing stage parses input picture description statements, performs name lookup and dictionary operations, rewrites aliased constraints to fundamental ones, applies defaults, and translates the picture description to the internal form used by the constraint solver. Within the solver constraints are encoded as lattices. One lattice exists for top-to-bottom positioning, and one for left-to-right. The constraint solver determines the absolute coordinate positions and sizes for the objects in the picture, attempting to satisfy all constraints. It applies defaults where necessary; e.g. for under-constrained specifications. It can also check the constraint system for consistency as a separate preprocessing stage. The postprocessor in the ICOLA converts the resulting object-coordinate information into calls to the declarative display predicates of the graphics engine (GM). The latter then generates the display.

ICOLA can operate in either an “incremental” or “batch” mode. In both, the constraint system is checked for consistency as each statement of the picture specification is read. In incremental mode, the constraint system is also solved on each input. It is solved only after all input is read when in batch mode.

4.1 ICOLA Interface

An example illustrating the current user interface and its functionality is provided in Appendix C. The example uses the picture description of Figure 4. The picture of Figure 3 is indeed achieved.

5. Concluding Remarks

We have described ICOLA, an implementation of a constraint-based graphics language. This language is to be used in the successor to the GLP system. Highlights of the language are that it is declarative ("what you say is what you get!"), that it does not require absolute layout specifications, and that it is able to deal with under-constrained picture specifications.

Work is continuing on ICOLA. Areas of current effort include: completing the implementation of all constructs in the constraint-based graphics language; improving the implementation, exploring additional or alternative constraints and attributes, and analyzing other schemes for constraint solution.

The processing of under-constrained specifications, especially in conjunction with the application of defaults, has proven to be a problematic area. Often no exact solution is possible and inappropriate or incorrect strategies are extremely obvious to the user. Much effort has been expended to achieve "reasonable" behaviour, though the search for better heuristics continues.

Bibliography

[Byrd80] "Understanding the Control Flow of Prolog Programs", by L. Byrd, in ***Proceedings, Logic Programming Workshop***, ed. S-Å. Tärnlund, 1980, Debrecen, Hungary, pg. 127 – 138.

[DeCl86] "Graphical Display of complex information within a Prolog debugger", ***Intern. Journal of Man-Machine Studies*, 25**: 503-521, 1986.

[DiLu87] "A Graphical Tool for Observing the Behavior of Parallel Logic Programs", by T. Disz and E. Lusk, in ***Proceedings, 1987 Symposium on Logic Programming***, 1987, pg. 46 – 53.

[EiBr88] "The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming", by A M. Eisenstadt and A M. Brayshaw, in ***Journal of Logic Programming*, 5**(4): 277 – 342, 1988.

[Car+92] "Some Paradigms for Visualizing Parallel Execution of Logic Programs", by M. Carro, L. Gómez, and M. Hermenegildo, ***Proceedings, ICLP '93***, Budapest, Hungary, June 21-24, 1993.

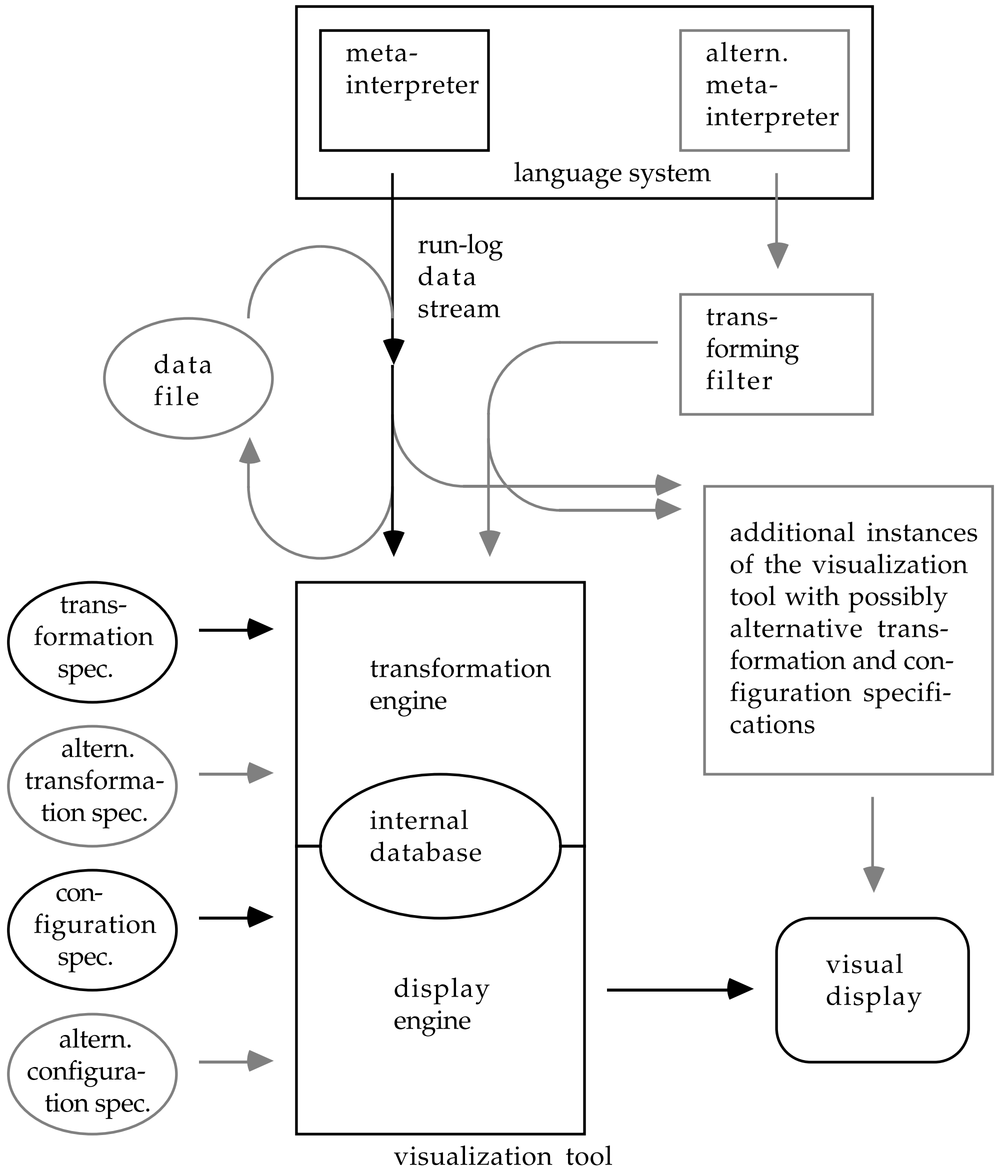
[HeMa91] "A Declarative Specification and Semantics for Visual Languages", by Richard Helm and Kim Marriott, in ***Journal of Visual Languages and Computing*, 2**:311-331, 1991.

[KuOs93] "Towards a Generalized Graphical Interface for Logic-Programming Development" by Anthony J. Kusalik and Gregory M. Oster, TR 93-2, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, February, 1993.

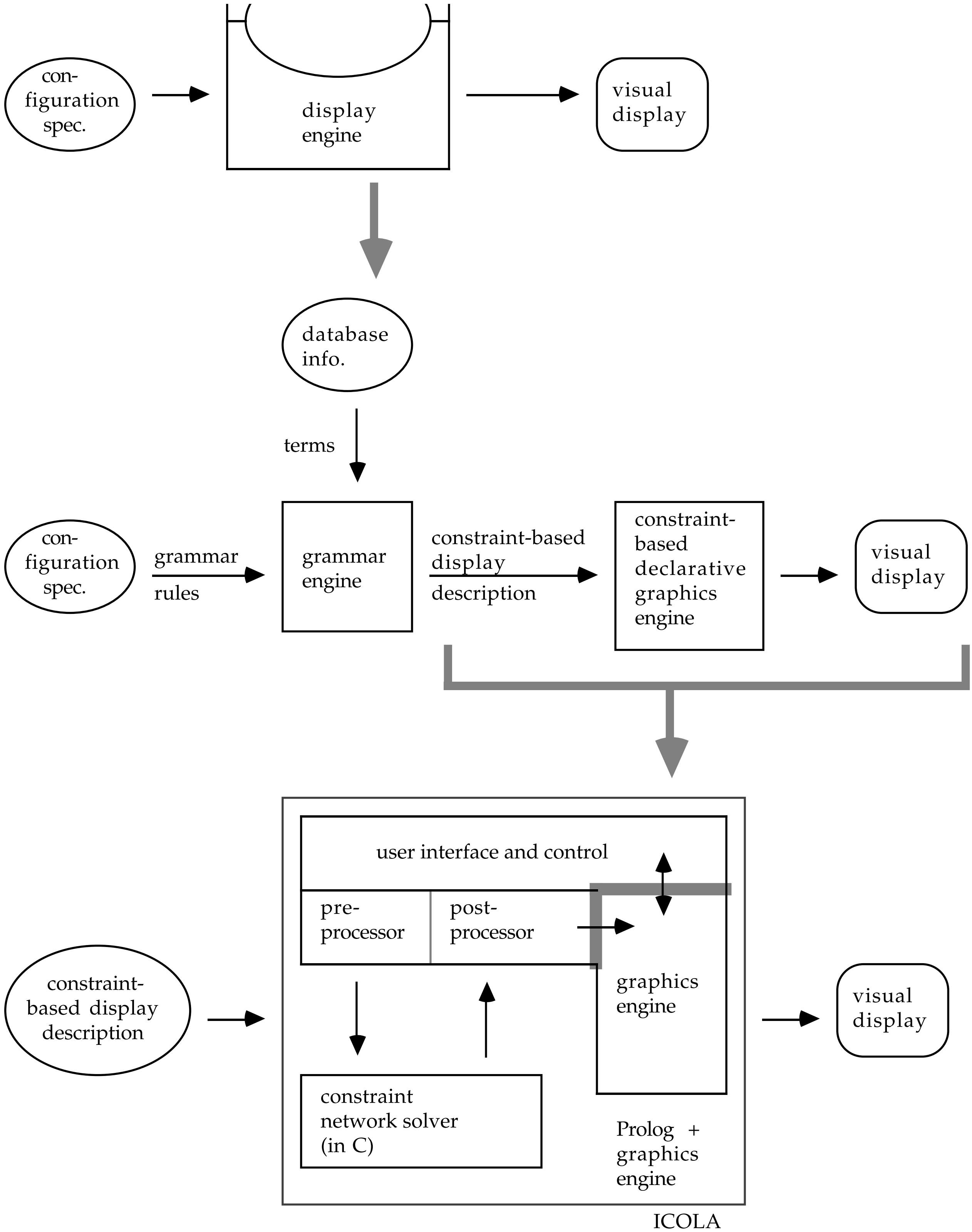
[SvSu90] "MuseTrace – A graphic tracer for Or-parallel Prolog" and "MUST User Manual", by C. Svensson and J. Sundberg, technical report T-90003, Swedish Institute of Computer Science, Kista, Sweden, 1990.

[Tick92] "Visualizing Parallel Logic Programs with VISTA", ***Proceedings, Intern. Conf. on Fifth Generation Computer Systems***, ICOT, Tokyo, Japan, June 1992.

Appendix A: Basic WAND design showing potential for tailorability



Appendix B: ICOLA implementation architecture



Appendix C: Sample ICOLA output using specification in Figure 4