

Automatic Generation of Intelligent Diagram Editors

SITT SEN CHOK and KIM MARRIOTT
Monash University

The *intelligent diagram* is a recent metaphor for diagramming in which the underlying graphic editor parses the diagram as it is being constructed, performing error correction and collecting geometric constraints that capture the relationships between diagram components. During diagram manipulation a constraint solver uses these geometric constraints to maintain the diagram's semantics. We introduce the Penguins system. This automates the development of graphical editors that support the intelligent diagram metaphor. It takes a grammatical specification of a particular diagram language and generates an editor specialized for the creation, manipulation and parsing of diagrams in that visual language. We extend previous research in this area by allowing more expressive grammars, performing automatic error correction, and detailing how efficient incremental parsing has been achieved. We also provide an empirical evaluation of the system. This shows that the system can be used to generate customized editors for a wide variety of diagram languages, ranging from state transition diagrams to mathematical equations, with real-time incremental parsing and error correction.

Categories and Subject Descriptors: I.3.4 [Computer Graphics]: Graphics Utilities—*Graphics editors*

General Terms: Algorithms

Additional Key Words and Phrases: Constraint multi-set grammars, constraint solving, diagram interaction, diagram parsing, intelligent diagram, pen-based computing

1. INTRODUCTION

Throughout history humans have used diagrams of various kinds for communication. Such visual languages are used in most disciplines and range from an architect's initial design to precise technical communication using rigorously defined notation, such as musical notation, state transition diagrams, mathematical notation or street maps. For eons diagramming was done manually

Preliminary versions of this paper appeared in Chok and Marriott [1995a, b; 1998].

Author's address: Kim Marriott, School of Computer Science and Software Engineering, Monash University, Wellington Road, Clayton Campus, Victoria 3800, Australia; email: Kim.Marriott@infotech.monash.edu.au.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2003 ACM 1073-0616/03/0900-0244 \$5.00

with pen and paper but with the introduction of computers a new era in diagramming support has evolved.

Two standard interaction models for diagramming software are uninterpreted diagramming and the syntax-directed model. The uninterpreted diagramming model is widely used and provides a single, simple framework for general purpose diagramming. An example is `xfig`. The order in which a diagram is drawn is not important, providing complete freedom during diagram creation. For example, a labelled arc can be drawn by either first drawing the arrow or the text label. Although the uninterpreted diagramming model is popular it has two large drawbacks. First, it does not check that the diagram is syntactically correct. Second, it does not automatically preserve the semantics of the diagram during manipulation. Thus when the arrow component of a labelled arc is moved, the associated text will remain behind unless the user explicitly adds a constraint linking the textual label with the arrow.

The syntax-directed diagramming model, on the other hand, provides structural guidance during diagram construction and enforces a top-down approach to diagram construction. In this model, users typically have to select the appropriate structure template containing empty slots, and then fill in the details. An example of such a syntax-directed editor is the mathematical equation editor used in Microsoft Word. This model has two main drawbacks. First, users are restricted in the order in which they can develop their diagrams. It is impossible for the user to construct incorrect syntactic structures for intermediate diagrams as the editing process is guided strictly by a sequence of structural templates. Permitting incorrect intermediate diagrams is important since it is natural to have syntactically incorrect diagrams in the editing process leading to the final syntactically correct diagrams [Rekers and Schürr 1995; Serrano 1995]. Second, diagram manipulation is not possible unless complex transformation rules mapping one template structure to another are embedded into the system [Minas and Viehstaedt 1995].

Because of the inherent limitations of both the uninterpreted diagramming and the syntax-directed diagramming models, a new metaphor for diagramming, the *intelligent diagram* has emerged [Weitzman and Wittenburg 1993; Chok and Marriott 1995a; Gross and Do 1996; Chok and Marriott 1998]. This model combines the best features of the uninterpreted diagramming and syntax-directed diagramming models. It allows the diagram to be constructed in free form and in any order. The underlying graphic editor parses the diagram as it is being constructed, performing error correction and collecting geometric constraints that capture the semantics of the diagram. During diagram manipulation objects in the diagram can be moved or resized while the constraint solver maintains the semantics by preserving the geometric constraints between the diagram components.

Although the intelligent diagram metaphor is promising, two issues have to be addressed before this promise can be realized. The first issue is whether the metaphor is practical. For it to be practical, we require real-time incremental parsing of diagrams combined with error correction. We also require real-time constraint solving during direct manipulation of the diagram. An added complication is that the underlying geometric constraints must be sufficiently

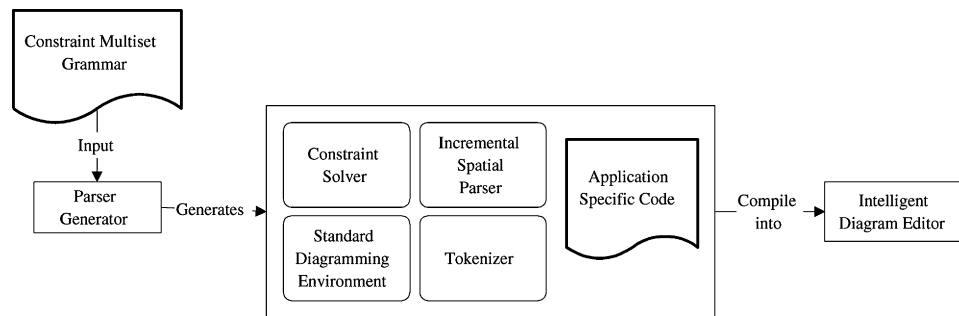


Fig. 1. An overview of the Penguins system.

expressive so as to allow the semantics of “real” diagram languages to be captured. In particular they should provide touching and containment. The second issue is that we must provide support for application programmers who wish to develop diagramming environments based on this metaphor. It is unrealistic to expect an application programmer to develop an intelligent editor for each new language from scratch.

To address these two concerns we have developed the Penguins system. This facilitates the development of software that supports the intelligent diagram metaphor by automating most of the software development process, much like tools such as `lex` and `yacc` automate the compiler construction process. Figure 1 shows the main components of the Penguins system.

The generated diagram editor supports the creation, manipulation and interpretation of diagrams in the particular visual language whose high-level specification is provided by the application programmer. The specification language is based on *constraint multiset grammars*. From the specification, the *parser generator* generates an incremental *diagram parser* or *spatial parser* that is incorporated into the standard *diagramming environment*. This diagramming environment provides standard graphic primitives, such as lines, circles, texts and arrows. A *constraint solver*, QOCA, is incorporated into the standard diagramming environment to provide the constraint solving mechanisms necessary for geometric error correction and diagram manipulation. Certain modules in the generated program can be extended to cater to application specific computation. New terminal types can be added to provide language specific graphic primitives. The generated C++ code, together with the application specific routines are compiled to give the final intelligent diagram editor.

Not only is the intelligent diagram metaphor suitable for the standard mouse and keyboard interface, it also seems ideally suited to the emerging technology of pen-based computers in which a flat visual display device is combined with a transparent digital tablet and a pen-like pixel selector. It allows the user to draw diagrams naturally, almost exactly as they would on paper, with the benefits that the drawn components are automatically beautified and behave intelligently during direct manipulation. In order to provide support for such free-hand drawing from a pen as an alternative input method, the Penguins system also provides a *tokenizer*. This maps handwritten input gestures to the

appropriate graphic primitives using the standard technique of least-squares curve fitting.

Using the Penguins system we verify that the intelligent diagram metaphor is practical by building six example diagram editors for visual languages ranging from state transition diagrams to mathematical expressions and demonstrating that they provide real-time parsing and constraint solving during direct manipulation. This has required us to develop novel interactive parsing algorithms as well as constraint solving algorithms for linear arithmetic constraints. This also establishes that it is possible to automatically generate a diagram editor supporting the intelligent diagram metaphor from a high-level specification of the visual language.

2. RELATED WORK

Two other research groups have also investigated the intelligent diagram metaphor. Gross [Gross 1994a, 1994b; Gross and Do 1996] has developed a pen-based system that supports the incremental recognition of components of informal sketches made during the design process. Weitzman and Wittenburg also consider design support, incrementally recognizing document or diagram structure and enforcing the induced semantic constraints during manipulation [Weitzman and Wittenburg 1993, 1994, 1996], although here the input device need not be a pen. The Penguins system extends these by allowing more expressive constraints, such as inequalities, and a more powerful underlying grammatical formalism providing existential and negative constraints. For example, we believe it is unlikely that these other systems could be used to recognize N -ary trees or message sequence charts. Nor do these systems have a formal method for handling error correction. We also provide a detailed empirical evaluation and a detailed description of our incremental parsing algorithm.

Our approach to error correction can be regarded as a continuation of earlier work on constraint-based beautification of diagrams. For instance, that of Pavlidis and Wyk [1985]; Kurlander and Feiner [1992, 1993] and Igarashi et al. [1997]. The primary difference is that we use a grammar to guide error correction. This means that error correction can be context and syntax dependent.

A number of researchers have looked at automatic construction of syntax-directed diagram editors from a visual language specification [Üsküdarlı 1994; Minas and Viehstaedt 1995]. In this diagramming model the visual language grammar guides the top-down construction of the diagram. As previously noted, one drawback of this approach is that it is impossible for the user to construct incorrect syntactic structures for intermediate diagrams as the editing process is strictly guided by a sequence of structural templates.

There has also been considerable research into automatic generation of static parsers from a visual language specification. The most general approaches are either based on graph grammars (for instance Ferrucci et al. [1991]; Tucci et al. [1994]; Rekers and Schürr [1995]; Zhang et al. [2001]) or attributed multiset grammars (for instance Golin [1991]; Wittenburg [1992]; Golin and Magliery [1993]). The constraint multiset grammar formalism was introduced in Helm

and Marriott [1991]; Helm et al. [1991] and Marriott [1994] and is an example of an attributed multiset approach. However, during error correction and semantics preserving diagram manipulation the behavior is more graph grammar-like since the actual attributes are ignored but the relationships between the diagram components are preserved. For a more complete survey of visual language specification and parsing algorithms, see Marriott et al. [1998b].

Other related work includes that of Vander Zanden [1988, 1989], who introduces constraint grammars, a type of attribute grammar. They are intended to specify the part-whole relationships in application data, such as a binary tree, and constraints are used to specify the appearance of the data in a GUI and to ensure that the visual representation always remains in accord with the underlying data structure. In some sense this is the dual problem to the one addressed here since the underlying data structure corresponds to a parse tree so in Vander Zanden's work this is given and the main problem is how to display it. Since almost hierarchical (data) structures are allowed this corresponds to allowing existential quantification in the grammar. Some analogue of parsing is provided by transformations that can be interactively applied by the user to edit the visual representation and underlying data structure. They are essentially unrestricted production rules so if transformations corresponding to production rules in a constraint multiset grammar were provided the user could in theory interactively build a parse tree by selecting objects and applying a reduction one at a time. Of course unlike our system this is not automatic. There is no analogue to negative constraints in the grammar, and the constraint-solving technology is limited. Also there is no automatic error correction.

3. CONSTRAINT MULTISSET GRAMMARS

As we have indicated, the Penguins system uses *constraint multiset grammars* for the specification of the visual language. These provide a high-level and declarative framework for the definition of visual languages and arose out of early work in logic based formalisms for diagram specification [Helm and Marriott 1986, 1990]. An important reason for using a logic or constraint based formalism is that the same grammar can be used to both generate and to recognize diagrams in the language as constraints are inherently bi-directional. As we shall see constraints also allow for geometric error correction in parsing and semantics preserving manipulation in the graphics editor.

Productions in a constraint multiset grammar have the form

$$P ::= P_1, \dots, P_n \text{ where } C$$

indicating that the non-terminal symbol P can be recognized from the multiset of symbols P_1, \dots, P_n whenever the attributes of all symbols satisfy the constraints C . The constraints enable information about spatial layout and also semantic information to be naturally encoded in the grammar. Terminal types in constraint multiset grammars refer to graphic primitives, such as *line* and *circle*, instead of the usual tokens. Each *symbol type* has a set of zero or more attributes, typically used to describe its geometric properties. A *symbol* is an instance of a symbol type. A *sentence* is a multiset of terminal symbols, that is, tokens. The declarations for each symbol type must be given in the grammar.

In each grammar, there is a distinguished non-terminal symbol type called the *start* type.

As a running example for a constraint multiset grammar specification, consider the language of state transition diagrams. A typical example is shown in Figure 13(e). The symbol type declarations for this grammar are:

```
declare symboltype {
  arc(start:point,mid:point,end:point,label:string):nonterminal;
  startArc(start:point,mid:point,end:point):nonterminal;
  state(mid:point,radius:integer,label:string,kind:string):nonterminal;
  transition(start:string,tran:string,end:string):nonterminal;
  transitions(set:multiset<transition>):nonterminal;
  states(set:multiset<state>):nonterminal;
  std(ss:states,ts:transitions):starttype;
}
```

For instance, the first declaration specifies that the symbol type *arc* is a nonterminal symbol and that it has four attributes: its start-point, its mid-point, its end-point and its label.

The simplest production in the grammar for state transition diagrams is that defining the *arc*. It specifies that an *arc* in a state transition diagram is composed of a text label *T* and an arrow *A*, and that the mid-points of *T* and *A* must be the same:

```
R:arc() ::= A:arrow,T:text where (A.mid == T.mid) {
  R.start = A.start;
  R.end = A.end;
  R.mid = A.mid;
  R.label = T.label;
}
```

One potential problem with incrementally parsing diagrams in an environment that supports the intelligent diagram metaphor is that of ambiguity when recognizing sub-diagrams in a partial diagram. In the case of state transition diagrams, a *start arc*—an arrow to a state indicating that the state is the start state—is very similar to an *arc*. The only difference between these two structures lies in the additional text label forming part of the arc structure. This poses potential ambiguity in parsing. For example, if an arrow is drawn, should it be recognized as a start arc or an arc? In static parsing such ambiguity may be resolved by considering all possible maximal interpretations and then choosing the interpretation that explains all elements in the diagram. In our context such ambiguity is bad for at least two reasons. First, it makes parsing considerably more expensive since multiple parse trees must be constructed. Second, and even more importantly, it makes it extremely difficult to provide immediate feedback about what has been recognized so far during incremental diagram construction since the ambiguity may only be resolvable once the complete diagram has been drawn.

Constraint multiset grammars allow the use of *negative constraints* to remove ambiguity from the grammar.

Returning to our example we can remove the ambiguity between start arcs and normal arcs by using a negative constraint to specify that for a start arc there must be no text label R with the same mid-point as the arrow A:

```
S:startArc() ::= A:arrow where (
  not exist R:text where (R.mid == A.mid)
) {
  S.start = A.start;
  S.end = A.end;
  S.mid = A.mid;
}
```

As another example of the use of negative constraints we give the productions for final, normal and start states. A *final state* is made up of two circles C1 and C2 and text T satisfying the following geometric relationships. The first is that the mid-points of the circle C1 and the circle C2 are the same, the second specifies that the mid-points of the circle C1 and the text T are the same while the third ensures that C2 is the outermost circle. The production defining a final state is:

```
S:state() ::= C1:circle,C2:circle,T:text where (
  C1.mid == C2.mid &&
  C1.mid == T.mid &&
  C1.radius <= C2.radius
) {
  S.mid = C1.mid;
  S.radius = C2.radius;
  S.label = T.label;
  S.kind = "final";
}
```

A *start state* is composed of a labelled circle, C and T, with a start arc A pointing to it. To avoid ambiguity between the production for a start state and the production for a final state, a negative constraint is required. This negative constraint specifies that there must not be another circle M with the same mid-point, since if such a circle exists, then a final state should be recognized instead. The following shows the defining production for a start state.

```
S:state() ::= C:circle,T:text,A:startArc where (
  T.mid == C.mid &&
  OnCircle(A.end,C.mid,C.radius) &&
  not exist M:circle where (M.mid == C.mid)
) {
  S.mid = C.mid;
  S.radius = C.radius;
  S.label = T.label;
  S.kind = "start";
}
```

A *normal state* is similar to a start state, except that it does not have a start arc pointing to it. Likewise it is similar to a final state, except that it only has

one circle rather than two concentric circles. Thus there is potential ambiguity. The problem is that a labelled circle that is rightfully part of a start state or a final state, may be recognized as a normal state instead. To remove such ambiguity, negative constraints can be used to specify that there must not be a start arc pointing to the labelled circle and that there must not be another circle with the same mid-point.

```
S:state() ::= C:circle,T:text where (
    not exist M:circle where (M.mid == C.mid) &&
    not exist A:startArc where (OnCircle(A.end,C.mid,C.radius)) &&
    T.mid == C.mid
) {
    S.mid = C.mid;
    S.radius = C.radius;
    S.label = T.label;
    S.kind = "normal";
}
```

Constraint multiset grammars also allow *existential quantification*. Expanding the constraint multiset grammar formalism in this way essentially makes it context-sensitive. It has been proven in Marriott and Meyer [1997] that some sort of context-sensitivity is required for the specification of graph-based visual languages. The idea of existential quantification of variables is to allow the quantified variables to refer to symbols that have already been reduced as well as to symbols in the current sentence. Consider the following production, which specifies a *transition* in a state transition diagram. The variables S1 and S2 are existentially quantified. This means that they can be assigned symbols from either the current sentence or from the previously reduced symbols.

```
T:transition() ::= A:arc where (
    exist S1:state,S2:state where (
        OnCircle(A.start,S1.mid,S1.radius) &&
        OnCircle(A.end,S2.mid,S2.radius))
) {
    T.start = S1.label;
    T.tran = A.label;
    T.end = S2.label;
}
```

In order to accord with the multi-set semantics of constraint multiset grammars, different existentially quantified variables must be bound to different symbols. For instance, in the above example S1 and S2 must refer to two different states. Thus, to handle transitions from a state to itself we need an additional production:

```
T:transition() ::= A:arc where (
    exist S:state where (
        OnCircle(A.start,S.mid,S.radius) &&
        OnCircle(A.end,S.mid,S.radius))
)
```

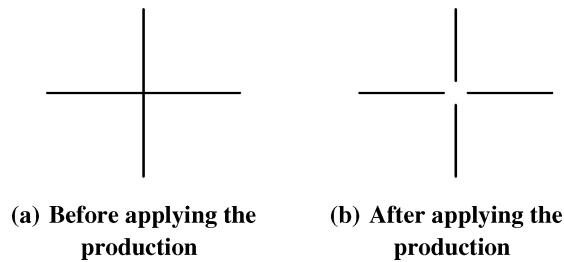


Fig. 2. Applying the example unrestricted production.

```

) {
  T.start = S.label;
  T.tran = A.label;
  T.end = S.label;
}

```

In many graph-like visual languages it is useful to be able to collect symbols of the same type into a multi-set. For instance, in the case of state transition diagrams, we wish to collect all of the states and all of the transitions. *Collection productions* support this activity. The following productions collect, respectively, all of the states and transitions in the diagram.

```

SS:states() ::=* all S:state where (true) {
  SS.set.Add(S);
}

TS:transitions() ::=* all T:transition where (true) {
  TS.set.Add(T);
}

```

The final production in our example grammar is the defining production for the start symbol type `std`. It specifies that a state transition diagram is composed of states `SS` and transitions `TS` and must have a start state `S`.

```

F:std() ::= SS:states,TS:transitions where (
  exist S:state where (S.kind = "start")
) {
  F.ss = SS;
  F.ts = TS;
}

```

Actually, the implementation of constraint multiset grammars in the Penguins system is more powerful than the example of state transition diagram suggests. Productions may also be *unrestricted*, that is more than one symbol may occur in the left-hand-side of a production rule. This allows, for instance, two crossed lines to be recognized as four non-crossing lines. This production is visually illustrated in Figure 2. However, the grammar writer must be careful with the use of such productions as they may lead to slow parsing and even to nontermination.

```

L1:line(),L2:line(),L3:line(),L4:line() ::=
  LA:line,LB:line where (Intersect(LA,LB)) {
    L1.start = LA.start;
    L1.end = Intersection(LA,LB);
    L2.start = Intersection(LA,LB);
    L2.end = LA.end;
    L3.start = LB.start;
    L3.end = Intersection(LA,LB);
    L4.start = Intersection(LA,LB);
    L4.end = LB.end;
  }

```

The `Intersect` constraint specifies that the lines `LA` and `LB` intersect. The computation function `Intersection` determines the point of intersection of the two lines, used to compute the end-points of the four new lines.

Our examples have illustrated that the constraints and functions used in the productions can either be linear arithmetic constraints or be defined by the application programmer. The latter are written as C++ procedures, which extend the `Constraint` class. It is the responsibility of the application programmer to make appropriate calls to the constraint solver so as to ensure the diagram semantics is modelled correctly. In some cases, for example `OnCircle`, there may be a mismatch between the desired constraints, in this case quadratic, and the constraints provided by the underlying solver, which are linear. This requires the programmer to approximate the desired constraints using those provided.

4. THE CONSTRAINT SOLVER

The constraint solver employed in the Penguins system is a revised version of the QOCA toolkit [Helm et al. 1995; Borning et al. 1997; Marriott et al. 1998a; Marriott and Chok 2002]. This is a C++ constraint solving toolkit, which supports the *metric space model* for various classes of arithmetic constraints.

In the metric space model, at any time there is a set of constraints over some variables, a metric that gives the distance between two assignments to the variables, and a current solution that is an assignment to the variables, which satisfies the constraints. The variables correspond to graphical attributes of the graphics objects in the diagram and the diagram on the graphics display is, therefore, simply a visual representation of the current solution.

Interaction in the metric space model can occur in three ways. First, a constraint may be added to the current set of constraints in which case the new solution is the assignment that is closest to the old assignment and that satisfies the new constraint set. Second, a constraint may be deleted from the current set of constraints, in which case the current solution remains unchanged. Finally, the current solution may be manipulated by “suggesting” a value for several of the variables. The new solution is the assignment that is closest to the old assignment and to the requested variable values and that satisfies the constraint set. The constraint set remains unchanged.

The metric space model provides a good formal basis for incremental manipulation of diagrams in a visual language specified by a constraint multiset

grammar. The constraints are generated in the parsing process. They capture the semantics or meaning of the diagram, for example in a state transition diagram, they capture that a transition starts and ends at a state. During manipulation of the diagram the constraints ensure that only valid (sub-)diagrams in the language are constructed.

The QOCA toolkit provides three main classes. An instance of the `CFloat` class, behaves like a float except that if its value is set, this is treated as advice to the constraint solver, not as a true assignment to the variable. Only the constraint solver can set the value of a `CFloat`. `LinConstraints` represent the equality or inequality constraints in the problem. They are expressions built on top of `CFloats`. Actually, neither `CFloats` nor `LinConstraints` are directly manipulated by the programmer. Instead, for efficiency and safeness, they are manipulated by means of the reference-counted “handles” `CFloatHandle` and `ConstraintHandle`, respectively, which can be assigned and constructed in the obvious ways.

To the application programmer the `CFloat` appears to have a single floating point value, which they set using the method `SuggestValue` and then read after calling the constraint solver using `GetValue`. Internally, however, the desired and actual values are kept separately. It also has a *stay weight* and an *edit weight* which, respectively, indicate the importance of leaving the variable unchanged if it is not being edited and the importance of changing it to the desired value when the variable is being edited. Both weights are non-negative floats and are set when the `CFloat` is created and cannot be subsequently changed.

QOCA currently provides three different solvers. Here we use the constraint solver `CassSolver` for solving linear equations and inequalities, and `LinEqSolver` specialized for solving only linear equalities. `CassSolver` uses an incremental version of the simplex algorithm for constraint solving and for finding the closest solution to the suggested solution that satisfies the equations [Borning et al. 1997] while `LinEqSolver` uses an incremental version of Gauss Jordan elimination [Marriott et al. 1998a].

Constraints are added to the solver one at a time using `AddConstraint`. With each addition the solver checks that the new constraint is compatible with the current constraints. This method is used to add constraints that enforce the geometric relationships found in parsing. `RemoveConstraint` allows the removal of a constraint that is currently in the solver. This is used when an object is deleted by the user or when invalidation of a negative constraint causes parsing to be undone.

The solver provides four methods for direct manipulation. First the application programmer tells the solver which variables are to be edited using multiple calls to `SetEditVar`. Next `BeginEdit` is called. This initializes internal data structures for fast “resolving” of the constraints. Now during manipulation the application programmer repeatedly sets the desired values of the edit variables and then calls the solver function `Resolve`, which efficiently computes the new solution to the constraints which is as close as possible to the old solution and to the new desired values of the edit variables. Finally the application programmer calls `EndEdit` to signal the end of direct manipulation.

The following (slightly modified) example taken from Marriott et al. [1998a] gives some feel for QOCA's interface. Consider a diagram consisting of a point (x_m, y_m) and a line from (x_l, y_l) to (x_u, y_u) in which the point is constrained to lie at the midpoint of the line. The following program fragment creates the variables and constraints, adds them to the solver and calls `Solve` to compute the initial position. The constructor for `CFloatHandles` takes three arguments: the stay and edit weights together with an initial desired value. Note that both x_m and y_m have a stay weight of zero indicating that they are “dependent variables,” although they of course have a non-zero edit weight (otherwise, editing would never change their value). Next the program chooses x_m and y_m to be the edit variables, and then repeatedly samples the mouse to find the desired values and calls `Resolve` to compute the new value until the user releases the mouse button, which finishes the edit cycle.

```

CFloatHandle xl(1,1000,45), xm(0,1000,0), xu(1,1000,60),
yl(1,1000,45), ym(0,1000,0), yu(1,1000,60);
ConstraintHandle xcon = (1*xl + 1*xu - 2*xm == 0),
ycon = (1*y_l + 1*y_u - 2*y_m == 0);
CassSolver solv;

solv.AddConstraint(xcon);
solv.AddConstraint(ycon);
solv.Solve();
DrawLine(xl.GetValue(), yl.GetValue(), xu.GetValue(), yu.GetValue());

solv.SetEditVar(xm);
solv.SetEditVar(ym);
solv.BeginEdit();

while (mouse.button.down) {
    xm.SuggestValue(mouse.x);
    ym.SuggestValue(mouse.y);
    solv.Resolve();
    DrawLine(xl.GetValue(), yl.GetValue(), xu.GetValue(), yu.GetValue());
}

solv.EndEdit();

```

5. THE PARSER GENERATOR

The traditional role of a parser is to determine whether a particular sentence is in the language defined by a grammar. In the intelligent diagram metaphor another equally important role of the parser is to establish the geometric constraints between diagram components. Parsing of visual languages has been studied for more than three decades. However, support for the intelligent diagram metaphor raises three novel issues.

The first is *incrementality*. As the intelligent diagramming metaphor is intended to support interactive diagramming, it is necessary for the generated

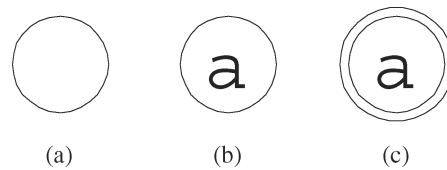


Fig. 3. Constructing a final state.

spatial parser to be incremental. Parsing must be performed incrementally whenever new graphic primitives are added to the diagram, thus providing immediate feedback to the user about the semantics of the current diagram.

The second issue is *error correction*. It is almost impossible for a human to draw the components of a diagram flawlessly so that, for instance, circles are truly concentric and arcs just touch states. This is even more of an issue with hand-drawn input. One solution is to use “snap-to-grid,” but this restricts object placement.

The third issue is how to handle *negative constraints*. This is the main difficulty when parsing interactively with constraint multiset grammars. The problem is that the addition of a new token may invalidate diagram structures previously identified, that is parsing is non-monotonic. For instance, consider the construction of a final state in which the user draws the final state by adding the tokens in the order shown in Figure 3. The intermediate diagram shown in Figure 3(b) will be recognised as a normal state, but, once the second circle is added in Figure 3(c), this initial identification must be undone since the negative constraint associated with the normal state is now invalid and instead a final state must be recognised.

We also note that there is an implicit negative constraint associated with collection productions since if a new token leads to recognition of a symbol of the type being collected by the production then the collection production needs to be reevaluated.

Apart from these new concerns, another important issue is that of *efficiency*. Unfortunately, the theoretical complexity of visual language parsing is considerably higher than that of parsing string grammars. Parsing for even the analogue of regular grammars is NP-hard [Marriott and Meyer 1997]. Concern with efficiency is not new, but the need for real-time incremental parsing makes it even more pressing.

5.1 The Basic Parsing Algorithm

We will now sketch how parsing is performed in the Penguins system, emphasizing how the above issues have been solved. For simplicity, we first ignore negative constraints and collection productions.

Parsing is performed in a bottom-up fashion, by taking a multiset of terminal symbols and repeatedly using satisfiable production rules in the grammar to combine symbols into larger and larger parse trees. Essentially, a *parse tree* is a tree of symbols in which each internal node has an associated production.

The main data structure employed by the parser is the *ParseForest*, which stores the parse trees recognized so far. The *ParseForest* is a collection of nodes of form

$$\langle t, [N_1, \dots, N_n], [N'_1, \dots, N'_m], Rule \rangle$$

indicating that t is the token constructed from reducing the parse tree nodes N_1, \dots, N_n using the production $Rule$ in the presence of the existentially quantified nodes N'_1, \dots, N'_m . Terminal tokens are represented by leaf nodes, that is nodes of the form $\langle t, [], [], NULL \rangle$.

Parsing combines parse trees in the *ParseForest* to create larger parse trees. More exactly, if the *ParseForest* contains nodes N_1, \dots, N_n that are at the root of a parse tree and nodes N'_1, \dots, N'_m that can occur anywhere in the parse tree and there is production, $Rule$,

$$\begin{aligned} T(\vec{A}) ::= & V_1 : T_1, \dots, V_n : T_n \text{ where} \\ & \text{exists } V'_1 : T'_1, \dots, V'_m : T'_m \text{ where}(C) \\ & \vec{A} = \vec{E}. \end{aligned}$$

for which the assignment θ from the *reduction variables* V_1, \dots, V_n and the *existential variables* V'_1, \dots, V'_m to the symbols in N_1, \dots, N_n and N'_1, \dots, N'_m respectively satisfies C , then the new node

$$\langle t, [N_1, \dots, N_n], [N'_1, \dots, N'_m], Rule \rangle$$

where symbol t is of type T with attributes \vec{A} set to $\theta(\vec{E})$ can be added to the *ParseForest* and represents a new parse tree with this node as its root. This is called a *reduction*. Note that after the reduction the nodes N_1, \dots, N_n are no longer root nodes. Also note that parsing is deterministic: once a node in the *ParseForest* has been reduced, it is not considered again for reduction.

Parsing stops when no further reduction can take place. A sentence is in the language of the grammar if when parsing finishes there is a single parse tree in the *ParseForest* and this has a node whose associated symbol has the *start* type as its root.

At the core of parsing is a procedure *EvalProduction_R* created by the parser generator for each production R . This performs reductions with the production R by exhaustively trying assignments of nodes in the *ParseForest* to the variables in R to try and find an assignment that satisfies the constraints in R . We use two main techniques to improve the efficiency of this process.

The first technique is to use a variant of the semi-naive fixpoint algorithm used in deductive databases [Ullman 1988]. This requires us to classify the symbols in the *ParseForest* as being either *new* or *old*. The *new* symbols are those that have been added to the *ParseForest* in this call to the incremental parser while the remaining symbols in the *ParseForest* are classified as *old*. The reason for this classification is that during an incremental parse, at least one symbol used in the reduction (assigned to either a reduction or an existential variable) must be a *new* symbol. This greatly reduces the number of productions and symbols that need to be considered at each step. This is correct since if the reduction only relied on symbols that were old, the reduction would have occurred in a previous call to the incremental parser.

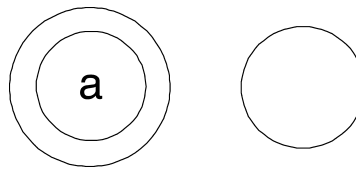


Fig. 4. Sample diagram.

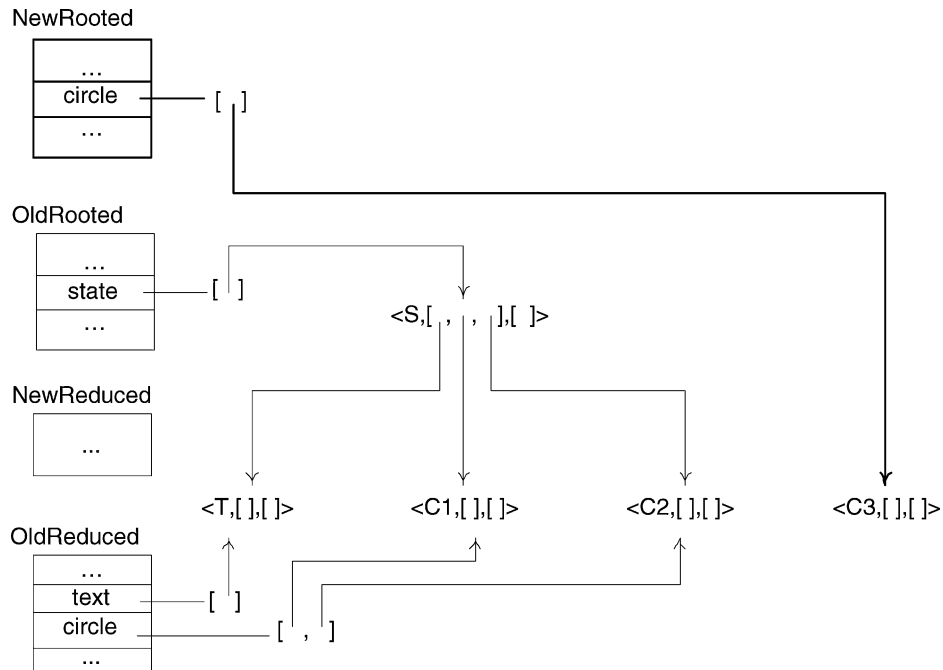


Fig. 5. The *ParseForest* and auxiliary index data structures for the diagram in Figure 4.

It is also useful to be able to readily distinguish between nodes that are at the root of a parse tree and those that are not, that is have previously been reduced. Therefore, the parsing algorithm relies on four auxiliary data structures, *NewRooted*, *OldRooted*, *NewReduced*, and *OldReduced* to index nodes in the *ParseForest* and so reduce retrieval time. Each node is indexed by exactly one of these data structures. Since we need to search for symbols by type, each of these data structures is an array indexed by symbol type, whose entry for that symbol type is a list of those nodes in the *ParseForest* with that type and classification.

An example may make these data structures a little clearer. Consider the diagram shown in Figure 4, in which the rightmost circle *C3* has just been added. Imagine that the parser has previously recognised that the small circle, *C1*, larger circle, *C2* and text, *T*, form a state, *S*, by using the production for final states, *FS*, detailed earlier. The *ParseForest* and its auxiliary index data structures are shown in Figure 5.

```

let  $v_0$  be the initial variable
let  $V$  be the set of remaining variables
let  $C$  be the set of constraints
 $seq := [assign(v_0)]$ 
 $known := \{v_0\}$ 
while  $C \neq \emptyset$  do
  choose  $c \in C$  s.t.  $c$  minimizes  $|vars(c) \setminus known|$ 
  let  $\{v_1, \dots, v_m\} = vars(c) \setminus known$ 
   $seq := seq::[assign(v_1), \dots, assign(v_m), test(c)]$ 
   $known := known \cup \{v_1, \dots, v_m\}$ 
   $V := V \setminus \{v_1, \dots, v_m\}$ 
   $C := C \setminus \{c\}$ 
endwhile
let  $\{v_1, \dots, v_n\} = V$ 
 $seq := seq::[assign(v_1), \dots, assign(v_n)]$ 

```

Fig. 6. Sequencing Algorithm.

The second technique to improve efficiency of reduction is to carefully use the constraints in the production to filter out wrong symbol choices as quickly as possible. The parser generator uses the Sequencing Algorithm given in Figure 6 to determine the order in which variables (including existentially quantified variables) are assigned symbols and constraints are tested. The aim is to detect failure as early as possible. In particular, constraints are tested immediately after all of the variables occurring in them have been assigned a symbol from the parse tree. The algorithm creates a sequence seq of expressions of form $assign(v)$ and $test(c)$ which, respectively, indicate choosing a symbol for variable v and testing constraint c . The algorithm assumes that there is a distinguished variable v_0 , which is to be the first variable assigned a value. The algorithm proceeds by repeatedly choosing the unscheduled constraint c that has the least number of unassigned variables (possibly none) and scheduling the assignment of values to these variables followed by a test of c . The function $vars(c)$ returns the variables in a constraint, for instance $vars(C1.radius \leq C2.radius)$ returns $\{C1, C2\}$.

To better understand the code for $EvalProduction_R$ produced by the parser generator for a production R , consider the production, FS , for recognising final states again. There is a block for each of the reduction and existential variables in the rule reflecting the choice of that variable as the one requiring an assignment from a new symbol. In the case of FS there are three blocks, one for each of the three reduction variables $C1$, $C2$ and T . The Sequencing Algorithm is used to order assignments to the remaining variable and constraint tests in each block. The code in Figure 7 shows the generated code for the block created for $C1$ from the sequence

```

 $assign(C1), assign(C2), test(C1.mid==C2.mid), test(C1.radius \leq C2.radius),$ 
 $assign(T), test(C1.mid==T.mid)$ .

```

The code for the blocks for $C2$ and T is similar, generated from the sequences

```

 $assign(C2), assign(C1), test(C1.mid==C2.mid), test(C1.radius \leq C2.radius),$ 
 $assign(T), test(C1.mid==T.mid)$ 

```

```

for each  $c_1$  in NewRooted[circle] do
   $c_1.lock := true$ 
   $\theta := \{C1 \mapsto c_1\}$ 
   $reduced := false$ 
  for each  $c_2$  in NewRooted[circle] or OldRooted[circle] s.t.  $c_2.lock = false$  do
     $c_2.lock := true$ 
     $\theta := \theta \cup \{C2 \mapsto c_2\}$ 
    if  $test_{C1.mid==C2.mid}(\theta)$  and  $test_{C1.radius<=C2.radius}(\theta)$  then
      for each  $t$  in NewRooted[text] or OldRooted[text] s.t.  $t.lock = false$  do
         $t.lock := true$ 
         $\theta := \theta \cup \{T \mapsto t\}$ 
        if  $test_{C1.mid==T.mid}(\theta)$  then
           $s := create_{state}(c_1.mid, c_2.radius, t.label, "final")$ 
           $AddNode(s, [c_1, c_2, t], [], FS)$ 
          move  $c_1, c_2, t$  from NewRooted to NewReduced and
          from OldRooted to OldReduced as appropriate
           $reduced := true;$ 
        endif
         $t.lock := false$ 
        if  $reduced = true$  then break endif % exit for loop
      endfor
    endif
     $c_2.lock := false$ 
    if  $reduced = true$  then break endif % exit for loop
  endfor
   $c_1.lock := false$ 
endfor

```

Fig. 7. Code generated for the first block in *EvalProduction_{FS}*.

and

$assign(T)$, $assign(C1)$, $test(C1.mid==T.mid)$, $assign(C2)$, $test(C1.mid== C2.mid)$,
 $test(C1.radius<=C2.radius)$.

respectively.

One subtlety is that each variable must be assigned a unique symbol from the *ParseForest*: different variables cannot be assigned the same symbol. This is achieved by setting a *lock* flag on a symbol to *true* when it has been assigned to a variable. Variables can only be assigned symbols whose *lock* flag is *false*.

The code assumes that the compiler generator creates a Boolean function $test_c(\theta)$ for each constraint c that tests whether an assignment θ satisfies the constraint, and a function $create_s$ for each symbol type s that creates a new symbol of that type given the appropriately ordered values for its attributes as function arguments. It also uses the procedure $AddNode(t, Red, Ex, Rule)$, which creates a node,

$$\langle t, Red, Ex, Rule \rangle$$

adds it to the *ParseForest*, and places it in the *NewRooted* index.

Further improvement in parsing efficiency is achieved by taking into account dependencies between the symbol types. The parser generator computes the *dependency graph* for the grammar and uses it to determine an optimal ordering of reductions in the generated parser, with productions used in order of their

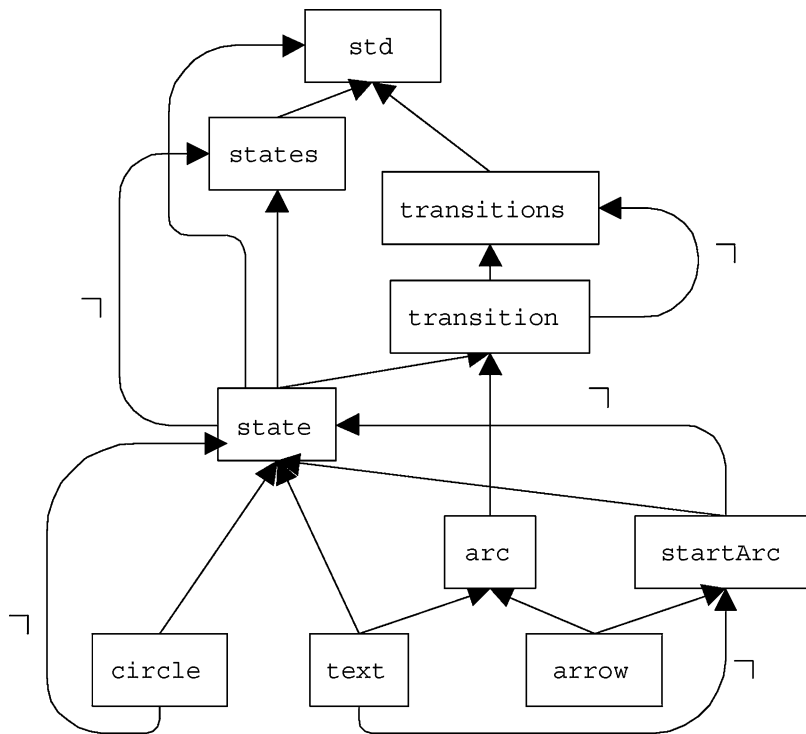


Fig. 8. Dependency graph.

strongly connected components (SCC). More precisely, nodes in the dependency graph are symbol types. If the symbol type A depends on the symbol type B , either through existential quantification or through reduction, then there is a positive dependency between the node for A and the node for B represented by the arc $A \leftarrow B$. A “negative” dependency between two symbol types is also represented by a connection in the dependency graph, but is labelled by “ \leftarrow^- ” instead of “ \leftarrow ”. Figure 8 shows the dependency graph for the state transition diagram grammar presented in the previous section. Note that for collection productions, there is an implicit negative constraint associated with the type it collects, since a collection is invalidated once a new symbol of the type it contains is created. Since the dependency graph works at the level of symbol types, all the defining productions for each symbol type are evaluated as a group. The following shows an optimal sequence of SCC evaluations based on the dependency graph for our running example:

```

SCC1: arc,
SCC2: startArc,
SCC3: [state(normal), state(final), state(start)],
SCC4: transition,
SCC5: states,
SCC6: transitions,
SCC7: std.

```

```

procedure IncParse( $T$ )
  for each  $t$  in  $T$  do
    AddNode( $t$ , [], [], NULL)
  end for
  for each SCC in order do
    repeat
      for each production  $R$  in the SCC do
        call  $EvalProduction_R$ 
      end for
    until no production can be applied
  end for
  for each symbol type  $s$  do
     $OldRooted[s] := NewRooted[s] :: OldRooted[s]$ 
     $OldReduced[s] := NewReduced[s] :: OldReduced[s]$ 
     $NewRooted[s] := nil$ 
     $NewReduced[s] := nil$ 
  end for
end

```

Fig. 9. Incremental Parsing Algorithm.

The parse forest is evaluated using productions in each SCC, starting from the lowest SCC and moving higher. The algorithm for incremental parsing with constraint multiset grammars is given in Figure 9. This takes the set of new tokens. It is called with $IncParse(\{t\})$ whenever a new token t is created by the user. The code for $IncParse$ first adds the tokens in the input set to the *ParseForest* and places them in the *NewRooted* index. It then iterates through the SCCs applying the rules in each SCC until the *ParseForest* is unchanged. Finally, it updates the four index structures.

The final technique used to improve efficiency is to reduce the number of constraints and constraint variables created during parsing. In most cases an attribute of a new nonterminal symbol is set equal to the attribute of some component symbol. In this case rather than introducing a new variable for the nonterminal symbol's attribute and adding a constraint to the constraint solver specifying the equality of the two attributes, the attribute of the nonterminal symbol is actually set to a reference to the variable.

5.2 Handling Negative Constraints

As illustrated earlier, negative constraints and collection productions make incremental parsing more difficult since the addition of a new token can make previous reductions invalid. The parsing algorithm is modified to handle such negative constraints in three main ways.

First, the parser employs an additional data structure, the *negative constraint table*, *NegConstraint*. For each type of symbol this stores a list of those nodes in the *ParseForest* that are the result of a reduction using a production that depends negatively on a symbol of that type and so could be invalidated when a symbol of that type is added.

Second, we add an extra field to each *ParseForest* node, which is the list of parse tree nodes that depend on that node, that is its *parents*. The dependency

```

procedure AddNode(t, Red, Ex, Rule)
  let t have type k
  add node (t, Red, Ex, [ ], Rule) to ParseForest and to NewRooted[k]
  for each n in NegConstraint[k] do
    if n is invalid then RemoveNode(n) endif
  end for
end

procedure RemoveNode(n)
  let n have form (t, Red, Ex, Par, Rule)
  for each n' in Par do
    RemoveNode(n')
  end for
  for each n' in Ex do
    remove link from n' to n
  end for
  for each n' in Red do
    remove link from n' to n
    move n' from OldReduced or NewReduced to NewRooted
  end for
  delete n from ParseForest and from any index referring to it
end

```

Fig. 10. Algorithm for removing invalid nodes.

can be through either existential or reduced variables. Thus the *ParseForest* now has links from parent to child and child to parent.

Third, the routine AddNode is modified so that it removes all nodes that are the result of previous reductions that the new symbol being added makes invalid. This is achieved by looking up the entry in the negative constraint table for the type of the new symbol. This entry is the list of all nodes in the parse forest that have a negative dependency on the type of the new symbol. The relevant negative constraint in the production used to create each of these nodes is rechecked. If one of the node's associated negative constraints is no longer true the node is removed from the parse tree using RemoveNode. Essentially, RemoveNode(*n*) removes the child-parent links to *n* for its children and resets the children assigned to reduction variables so that they become new root nodes. The only subtlety is that RemoveNode first calls itself recursively to remove all nodes that depend on *n*. The algorithms for AddNode and RemoveNode are given in Figure 10.

The code generated to test if a negative constraint holds may be considerably more complex than that for simple attribute tests, since it requires the *ParseForest* to be searched for symbols not satisfying the constraint. As an example, consider the generated code shown in Figure 11 for the negative constraint

```
not exist R:text where (R.mid == A.mid)
```

taken from the production to recognise a startArc.

To clarify the complete parsing algorithm, we now explain how the final state shown in Figure 3 is recognized. We assume that the innermost circle, *C1*, is drawn first, then the label, *T*, and finally the outermost circle, *C2*. Once *C1*

```

function  $test_{\text{not exist } R:\text{text where } (R.\text{mid}==A.\text{mid})(\theta)}$ 
  for each  $r$  in  $NewRooted[\text{text}]$  or  $OldRooted[\text{text}]$  or
     $NewReduced[\text{text}]$  or  $OldReduced[\text{Text}]$  s.t.  $r.\text{lock} = \text{false}$  do
     $r.\text{lock} := \text{true}$ 
     $\theta := \theta \cup \{R \mapsto r\}$ 
    if  $test_{R.\text{mid}==A.\text{mid}}(\theta)$  then return false endif
  endfor
  return true
end

```

Fig. 11. Example code for testing a negative constraint.

is drawn, `IncParse` is called with $\{C1\}$. In turn `AddNode(C1)` is called. Since there are no entries in *NegConstraint*, this simply creates a new node in the *ParseForest* and also places the node in the *NewRooted* index. This is shown in Figure 12(a). `IncParse` tries productions in order of the associated strongly connected components. None are applicable so no reduction takes place, $C1$ is moved to *OldRooted*, and `IncParse` terminates. This is shown in Figure 12(b).

Next T is added and `IncParse({T})` is called. Again there are no entries in *NegConstraint*, so this simply creates a new node in the *ParseForest* and also places the node in the *NewRooted* index. This is shown in Figure 12(c). Productions are tried in order of the strongly connected components. The first production tried is that for `arc`. This cannot be applied since there is no arrow. The next production tried (that for `startArc`) also cannot be applied because it also depends on an arrow token. Next the production, `NSProd`, for recognising normal states is tried. This production is applicable and creates the new state token NS , which is added to the *ParseForest* and to the *NewRooted* index by `AddNode`. The nodes T and $C1$ are moved to *NewReduced* and *OldReduced* respectively. Since the production for normal states depends negatively on circles, NS is added to *NegConstraint[`circle`]*. This is shown in Figure 12(d). The remaining productions are tried but none are applicable, so `IncParse` terminates after appropriately updating *NewReduced*, *NewRooted*, *OldReduced*, and *OldRooted*. This is shown in Figure 12(e).

Now `circle C2` is drawn and `IncParse({C2})` is called. As before `AddNode` is called, creating a new node for $C2$ and adding it to the *ParseForest* and to *NewRooted*. This is shown in Figure 12(f). This time, however, *NegConstraint[`circle`]* contains an entry for the node containing NS . The negative constraints in the production for NS are reevaluated and are found to be invalidated. Thus `RemoveNode` is called to remove this node. This removes the node for NS from *ParseForest* and from *OldRooted* and the nodes containing $C1$ and T are now moved from *OldReduced* to *NewRooted*. This is shown in Figure 12(g). Again, productions are tried in order of the SCCs. This time the production, `FSPProd`, for final state succeeds and the three tokens are recognised as forming the final state FS . No further productions can be applied. The result is shown in Figure 12(h).

5.3 Correctness of the Parsing Algorithm

It should be clear that the parsing algorithm described above is correct in the sense that if the algorithm terminates and there is a single parse tree

in the *ParseForest* and this has a token with *start* type as its root then the sentence is in the language of the grammar. Unfortunately, the parsing algorithm as described above may not terminate, nor is it guaranteed to give the same result even if it does terminate. However, parsing is guaranteed to terminate with the same result if the grammar satisfies the following three requirements.

The first requirement is that the input grammar is *stratified*, that is, there is no recursion through negative constraints. The problem is that a production may directly or indirectly create the symbol it depends on negatively, that is, there may be a cycle in the dependency graph that passes through a negative constraint. For example, the following production has such a cycle:

```
W:negDep() ::= T:text where (
    not exist N:negDep where (
        N.mid == T.mid))
{ ... }
```

If there is a cycle through a negative constraint in the dependency graph, subsequent reductions will produce new symbols that may invalidate an earlier reduction involving the negative constraint. This may lead to an infinite loop in which a symbol is repeatedly created then deleted. To avoid this potential problem the parser generator determines the grammar's dependency graph and checks that the grammar is stratified in the sense that each symbol only depends negatively on those symbol types that are strictly lower in the dependency graph. It is obvious from the dependency graph shown in Figure 8 that our example grammar is stratified. The notion of stratification is loosely based on that introduced for deductive databases [Ullman 1988]. If the grammar is unstratified, the parser generator will generate a warning.

A stratified grammar has an implicit order in which parsing must occur. Basically, we can only reduce with a production if all of the negative symbols upon which it depends have already been recognized. More exactly, the stratification gives a (partial) order (actually an irreflexive and transitive relation) on symbols where $s < t$ if t depends negatively on s . A sequence of reductions for a stratified grammar is said to *respect the stratification* if, whenever $s < t$, all reductions generating a symbol of type s occur before those reductions generating a symbol of type t . Note that our parsing algorithm will always respect the grammar's stratification since it uses the dependency graph to guide the order in which productions are tried.

The second requirement needed to ensure termination is that the reduction sequence is finite in length. A sufficient condition to ensure this is that the grammar is *cycle-free*, in the sense that there are no reduction sequences in which a single symbol can be reduced to give itself without reducing other symbols. For example, the grammar

```
A:Cycle1() ::= B:Cycle2 where (exist C:Dep ) { }
B:Cycle2() ::= A:Cycle1 where (true ) { }
```

has a cycle since a symbol of type `Cycle1` or `Cycle 2` can be reduced to give itself. Of course, cycle-freeness does not dis-allow recursion. For instance,

```
L:List() ::= N: Nil where (...) { ...}
L:List() ::= R:List, I:Element where (...) { ...}
```

is cycle-free.

The parser generator checks that the restricted productions in the grammar are cycle-free before code generation. Though the condition of cycle-freeness can be generalised to unrestricted grammars, currently this has not been implemented and it is the programmer's responsibility to ensure this. Examination of the productions in the state transition diagram grammar reveals that the grammar is cycle-free.

The third requirement for completeness of the parsing algorithm is that the constraint multiset grammar specification be *deterministic*. Essentially this means that during recognition of a valid diagram there should never be the possibility of applying more than one production to reduce the same symbol. This is required for correctness since for efficiency, once a reduction is performed the parsing algorithm does not consider alternatives.

Precisely defining what is meant by deterministic however is a little complex. We must first introduce the *intended language* of the grammar. This is the set of sentences that are in the language of the grammar and that may need to satisfy some additional well-formedness conditions. For instance, in the case of state transition diagrams, these well-formedness conditions might be that no states overlap and no transition labels overlap. In principle the grammar writer could refine their grammar to check these well-formedness conditions, but for simplicity or efficiency this may not have been done. A grammar G is *deterministic* if for any subset S of any sentence in the intended language of G , every maximal sequence of reductions for S that respect the stratification of G give rise to the same multiset of parse trees—there is a unique maximal interpretation for S . The parser generator does not check this condition since it depends upon the intended language of the grammar: it is the responsibility of the grammar writer to confirm it.

5.4 Error Correction

Diagrams constructed with a stylus or with a traditional graphics editor almost always contain geometric errors such as lines not quite touching the objects they should touch, or text not being centered in a circle or precisely placed at the midpoint of an arrow. Therefore, some form of automatic geometric error correction mechanism is crucial, providing immediate feedback to the user about what has been recognized and removing geometric errors.

More precisely, we can define the *geometric error* of a diagram as the distance from the “closest” diagram in the language. In order to do this, some metric, say *dist*, which measures the distance between two geometric attribute values is required. We extend *dist* to non-geometric attributes, by defining *dist* to be 0 if they are the same and ∞ if they are not.

The distance between two tokens T and T' of the same type and with attributes a_1, \dots, a_n is defined to be

$$\text{dist}(T, T') = \sum_{i=1}^n \text{dist}(T.a_i, T'.a_i)$$

while the distance between two tokens of different type is ∞ . The metric is extended to sentences of tokens in the obvious way: Let S be the sequence of tokens T_1, \dots, T_n and S' the sequence T'_1, \dots, T'_m . The *distance* between S and S' is

$$\text{dist}(S, S') = \sum_{i=1}^n \text{dist}(T_i, T'_i)$$

if $m = n$ and ∞ otherwise.

We define the *distance* between two sentences S and S' to be the minimum of the distances between all permutations of S and S' , and the *geometric error* of a sentence S for a constraint multiset grammar G to be

$$\min\{\text{dist}(S, S') \mid S' \text{ is in the language of } G\}.$$

Note that, from the definition of *dist*, if the two sequences have different structures then *dist* is ∞ .

Our notion of geometric error distance is analogous to error distance in string languages in which the difference between the strings S and S' is the length of the minimum sequence of editing commands such as character addition or deletion, required to transform S into S' [Aho et al. 1987]. For both visual language parsing and string parsing, error correction can, therefore, be understood as the problem of finding the sentence in the language that is “closest” to the input sentence.

The definition of geometric error presented so far requires that, in effect, all the sentences in the language G are compared with the sentence S . This definition is theoretically correct but cannot be efficiently implemented. Instead, we use a heuristic that approximates this ideal. In practice we have found that the heuristic is fast and does find the closest diagram. The idea is that the incremental parsing algorithm first determines if the sentence S is “almost” in the language of G before transforming it to the closest sentence S' with the same parse tree as S in the language of G . The incremental parser considers the sentence S to be “almost” in the language of G if the associated geometric error for each constraint is within an acceptable error tolerance level. The heuristic is correct as long as the real closest sentence S'' to the original sentence S has the same parse tree as that found for S (and assuming that the metric used by the solver and the error measurement are in accord).

What this means is that reduction with a production has two stages: In the first stage, constraints in the production are treated as Boolean tests and are considered satisfiable if the associated errors are small enough. For example, points A and B are considered equal if the distance between them is within the acceptable tolerance. If this constraint checking stage is successful, the parser commits to the use of this production.

In the next stage, constraints that capture the geometric relationships between the symbols involved in the reduction are added using `AddConstraint` to the constraint solver. Next the `Solve` method is called. This finds new values for the attributes of symbols that satisfy all of the constraints in the new constraint set, but which are as close as possible to the current values. By doing this, the geometric errors of the diagram are removed. The diagram is re-displayed to reflect the new solution, giving semantic feedback to the user about recognition.

Currently the grammar writer sets a global error tolerance. However, since grammar writers have the ability to define their own constraints, they may customise the error tolerance for particular constraints and productions. The degree of tolerance needs to be chosen with some care since if it is too lenient, the grammar with error tolerance may no longer be deterministic even if the original grammar is.

6. THE DIAGRAM EDITOR

The core graphics editor is combined with the generated incremental parser and the constraint solver to give a customized graphic editor for the specified visual language. The editor allows the user to add or delete symbols and to manipulate components of the diagram.

In the context of the intelligent diagram metaphor, diagram editing operations naturally fall into two categories: those that modify the current diagram semantics, and those that preserve it. Operations like adding or deleting an element change the semantics while manipulations like moving or resizing a subdiagram generally should not change the semantics. In implementation terms, semantics preserving operations do not change the parse forest or constraints in the constraint solver, while those that modify the semantics may change them.

Addition of graphic elements has been discussed in the preceding section. Clearly addition does not preserve the diagram semantics since it modifies the parse forest and constraints in the constraint solver.

Deletion of graphic elements from a diagram also changes the semantics of the diagram since previously identified diagram structures that depend on the deleted tokens become invalid. Thus, the parse forest must be updated. A naive solution to the problem is to perform a complete reparsing of the entire diagram to obtain the corresponding parse forest. This solution is simple, but inefficient. A better approach is to leverage from the code used to recheck negative constraints. We simply call `RemoveNode(t)` for each deleted token *t* to remove the invalid parts of the parse forest, and then call `IncParse({})` to perform an incremental parse. For example, imagine that two concentric circles and some text have been recognized as a final state and that one of the circles is now deleted. The call to `RemoveNode` will remove the circle and the dependent final state from the parse forest. The subsequent call to `IncParse` will cause the circle and text to be recognized as forming a normal state.

During direct manipulation of the diagram, the user usually wants the semantics of the diagram to be preserved in order to create the illusion that the graphics editor “understands” the diagram. This is made possible by the

embedded constraint solving mechanism. As previously mentioned, the constraints representing geometric relationships between symbols in a diagram are added into the constraint set during incremental parsing. When a token is manipulated, its geometric attributes are changed to reflect its new geometric location. For instance, imagine moving a circle that belongs to a normal state in a state transition diagram. When the circle is moved, the associated text and transitions follows in order to preserve the original semantics of the diagram.

However, sometimes the user wants to violate these implicit constraints during direct manipulation. To accommodate this we also allow the user to perform nonsemantics preserving direct manipulation. In this operation the tokens selected for direct manipulation are first removed from the parse tree and only when the user has moved them to their new location is parsing performed.

7. EMPIRICAL EVALUATION

We now provide an empirical evaluation of the Penguins system. All tests were carried out under the Microsoft Windows NT 4.0 operating system on a Dual Pentium-II 450 machine.

Our evaluation is based on six graphic editors we have developed with the system, each employing a different visual language. The visual languages are state transition diagrams, mathematical expressions, structured flow charts, message sequence charts, binary trees and N-ary trees. Example diagrams for each of these visual languages constructed with the generated graphic editors are shown in Figure 13.

Message sequence charts are commonly used to describe interaction between entities, how messages are interchanged between process instances, and the internal actions they perform. The grammar for this visual language uses unrestricted productions. The binary tree example recognizes a binary tree diagram representing a simple mathematical expression, while the N-ary tree example recognizes trees with an arbitrary number of children. Although this visual language appears similar to that of binary trees, the grammatical specification is actually quite different and relies on recursive use of negative constraints. The graphic editors for the first three visual languages also provide visual language specific operations. That for state transition diagrams determines if a string can be recognized by the corresponding finite state automata, that for mathematical expressions recognizes a mathematical expression constructed from the standard arithmetic operators, and generates the corresponding \LaTeX expression, while that for structured flow charts generates the corresponding pseudo-code for the flow chart.

The first aspect of the Penguins system we evaluate is the complexity of specifying a visual language using the constraint multiset grammar formalism and the performance of the parser generator. Table I gives details for each example visual language. Compilation was with Borland C++ Builder 4.0. We see that the specification for all applications is reasonably compact. Comparison of the specification size and the size of the generated C++ demonstrates the expressiveness of the constraint multiset grammar formalism. On average, the generated C++ code is an order of magnitude larger than the original specification. We also see

Table I. Size of Specification Files and Generated Applications

	LS	P	NT	GC	GT	CT	ES
State Transition Diagram	226	11	6	2539	1.1	354	842
Mathematical Expression	239	23	12	6412	1.6	380	1013
Flowchart	225	17	11	4127	1.6	375	880
Message Sequence Chart	196	14	11	2554	1.1	370	833
Binary Tree	111	5	1	2377	1.2	361	891
N-ary Tree	116	6	4	1868	1.2	353	786

Legend:

- LS—total number of lines in the specification,
- P—number of productions,
- NT—number of non-terminal types,
- GC—number of lines in the generated C++ code,
- GT—generation time (seconds),
- CT—compilation time (seconds) and
- ES—executable's size in Kilobytes.

Table II. Time for Parsing with Error Correction

Visual Languages		Size				
		20	40	60	80	100
State Transition Diagram	C	93	186	298	388	502
	V	126	251	390	504	635
	A	0.004	0.009	0.016	0.025	0.035
Binary Tree	C	182	364	544	724	910
	V	208	416	618	816	1014
	A	0.004	0.005	0.008	0.009	0.012
N-ary Tree	C	116	324	487	649	818
	V	202	388	564	750	942
	A	0.007	0.011	0.024	0.040	0.060
Message Sequence Chart	C	84	199	339	501	670
	V	128	256	376	496	616
	A	0.002	0.007	0.010	0.030	0.041
Flowchart	C	145	312	467	627	864
	V	204	418	618	822	1042
	A	0.005	0.009	0.010	0.010	0.018
Mathematical Expression	C	239	528	790	1044	1332
	V	264	568	852	1130	1434
	A	0.035	0.062	0.083	0.139	0.206

Legend:

- Size—number of tokens in diagram,
- C—number of constraints,
- V—number of variables,
- A—average time in seconds taken to add a token while constructing diagram. Time includes error correction.

that the time generating each parser is negligible when compared to the time required to compile the generated C++ code.

The second aspect of the Penguins system we evaluate is the performance of the generated parsers. Each of the parsers for our six examples has been tested with input diagrams of various sizes. Figure 13 shows the largest diagram used in each experiment. We use `LinEqSolver` for the mathematical equations, and `CassSolver` for the other examples. The results are shown in Table II. We see that the incremental parser with error correction is quite fast. In all examples but mathematical expressions, taking less than one tenth of a second.

Table III. Empirical Evaluation of Diagram Manipulation of Various Sized Diagrams

Visual Languages		20	40	60	80	100
State Transition Diagram	B	0.000	0.000	0.000	0.000	0.000
	R	0.000	0.000	0.001	0.001	0.002
Binary Tree	B	0.000	0.001	0.001	0.001	0.001
	R	0.002	0.003	0.003	0.004	0.005
N-ary Tree	B	0.000	0.000	0.000	0.001	0.000
	R	0.000	0.002	0.003	0.004	0.004
Message Sequence Chart	B	0.000	0.000	0.000	0.000	0.001
	R	0.000	0.001	0.001	0.006	0.008
Flowchart	B	0.000	0.001	0.001	0.051	0.055
	R	0.001	0.002	0.002	0.003	0.004
Mathematical Expression	B	0.031	0.219	0.484	0.594	0.750
	R	0.000	0.000	0.000	0.000	0.000

Legend:

- B—time taken for initial call to `BeginEdit`,
- R—maximum time taken for a `Resolve` during direct manipulation.

All times in seconds.

The mathematical expression application is somewhat slow. Examination of the number of constraints indicates why—the grammar for mathematical expressions generates a large number of constraints. However, we note that in practice mathematical expressions are usually quite small and for expressions of up to 60 elements the editor works well.

The final aspect of the Penguins system we evaluate is the speed of constraint solving during diagram manipulation. In Table III, for each example application, we give the time to build the computed solved form in the call to `BeginEdit` and the maximum time taken for a `Resolve` by the constraint solver. We have deliberately moved a diagram component connected to almost all other components in the largest diagram in our test cases, so this is the worst case scenario. Performance of `Resolve` is very reasonable in all examples, typically only a few milliseconds even for large diagrams. Performance for `BeginEdit` is fine for all examples except mathematical expressions. With mathematical expressions, the `BeginEdit` is somewhat slower because of the different constraint solver, `LinEqSolver`, being used and the large number of constraints. Even so `BeginEdit` takes less than one second.

8. CONCLUSION AND FUTURE WORK

Our empirical results clearly demonstrate the practicality of the intelligent diagram metaphor for diagram interaction. They show that for most visual languages, incremental parsing, error correction and semantics preserving direct manipulation can be done in real-time for even large, complex diagrams. Our results also show that automatic generation of diagram editors supporting the intelligent diagram metaphor from a compact grammatical specification of visual language is practical.

Interestingly, the basic parsing algorithm and the important notion of stratification are based on the theory of deductive databases. We feel this is because parsing with multisets is actually more akin to the kind of set-based manipulation required in databases than it is to parsing of string languages.

In hindsight, the one aspect of the Penguin's system we are unhappy with is the choice of semantics for existentially quantified variables. Currently, existential variables can be assigned any symbol in the parse tree as long as they are different. This means that reduction variables and existential variables may be assigned symbols that depend upon each other and so could never appear in the sentence at the same time. For example, the sentence $\{c\}$ will be reduced to $\{a\}$ with the grammar

```
A:a() ::= B:b where (exist C:c) { }
B:b() ::= C:c where (true) { }
```

We now believe a more natural semantics for existentially quantified variables is that they behave like symbols in a context sensitive grammar, namely that symbols assigned to reduction and existential variables in a production should not be allowed to depend upon each other in the sense that one relies on the reduction of the other.

One extension to the Penguins system we have investigated is how to extend the constraint multiset grammar formalism to include specification of layout requirements [Chok et al. 1999] and also to allow the user to add arbitrary constraints to the diagram. As an example, layout requirements for binary trees might specify that nodes on the same level are aligned horizontally and that parents are centered vertically between their children. Such “pretty printing” requirements should not be included as required constraints since this imposes sometimes dramatic restrictions on how the user must draw the diagram. Instead layout requirements need to be specified separately from those for recognition. They can be viewed as additional geometric constraints that are considered by the constraint solver but not the incremental parser.

We would also like to incorporate transformation rules into constraint multiset grammars. Transformation rules allow high-level specification of diagram execution and animation. Unlike production rules, transformation rules are evaluated at the user's command and are similar to those described in Vander Zanden [1989]. As an example, consider a transformation rule `tCircleToSquare` that transforms a labelled circle to a labelled square.

```
[tCircleToSquare] S:square(),T:text() ::=
  C:circle,T:text where (T.mid == C.mid)
  {
    S.mid = C.mid;
    S.width = C.radius * 2;
  } where (S.mid == T.mid)
```

Another area for future research is the extension of the underlying constraint solver to provide nonlinear as well as linear arithmetic constraints. Currently, inherently nonlinear constraints such as `OnCircle` must be approximated by the application programmer using linear constraints.

One final area of future work is to perform usability studies on the Penguins system. One such study involves the development of some practical pen-based applications based on different visual languages and to compare the relative

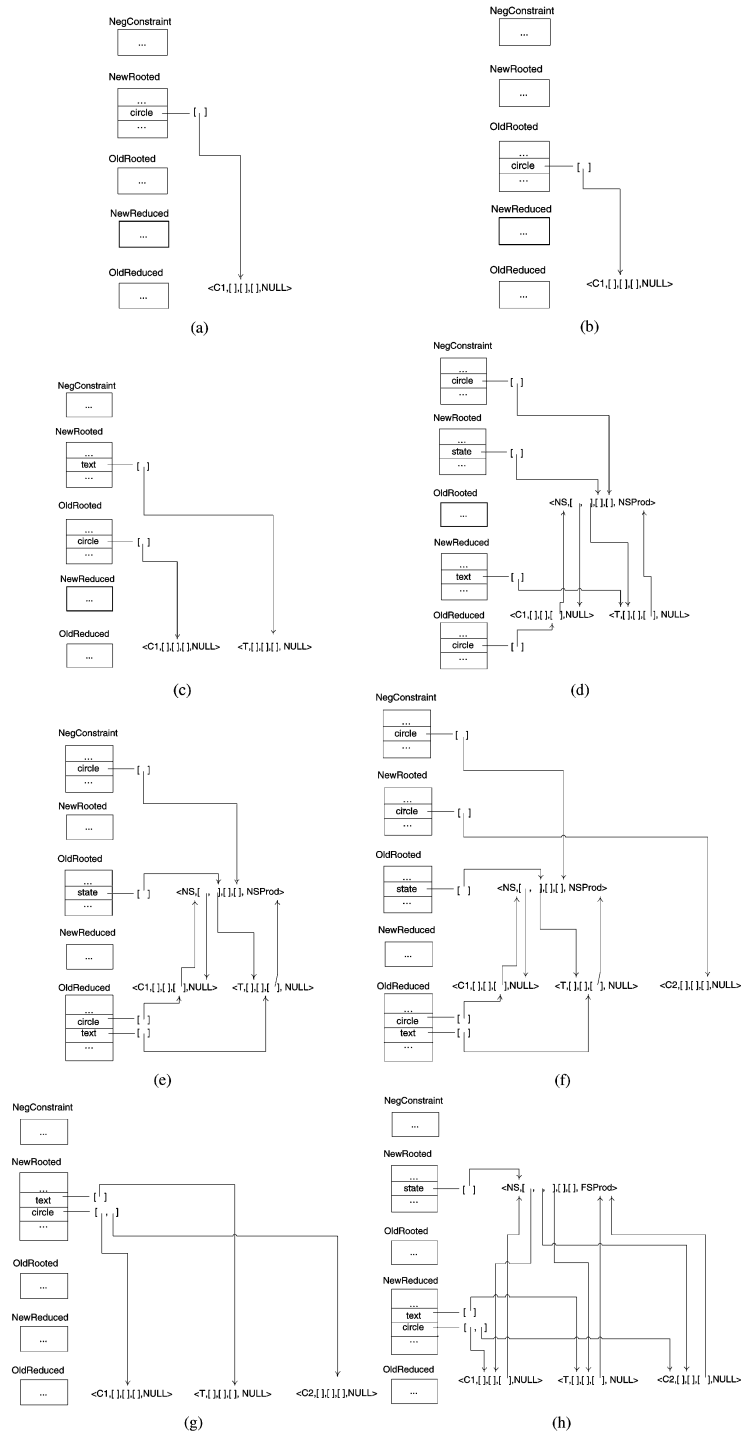
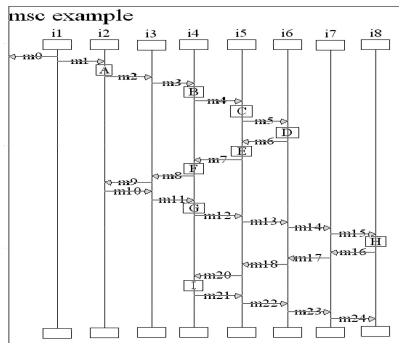
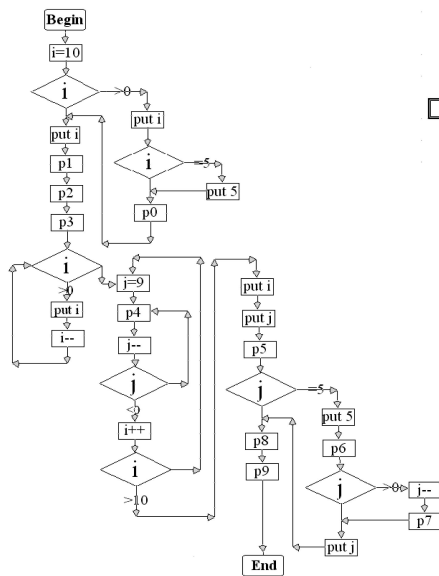


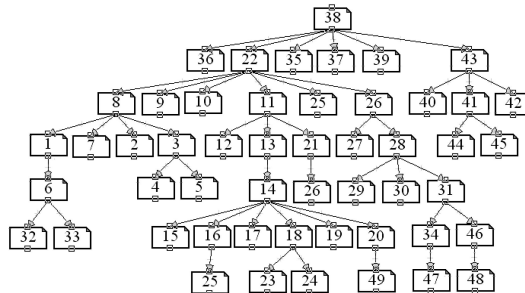
Fig. 12. The *ParseForest* and auxiliary index data structures during recognition of the diagram in Figure 3.



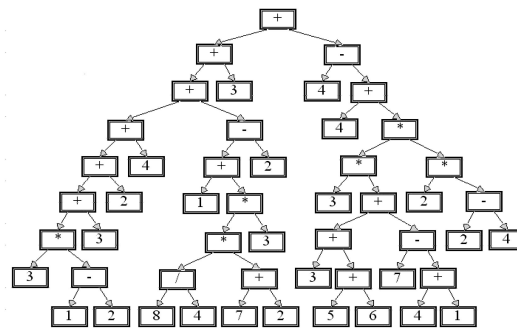
(a) Message sequence chart



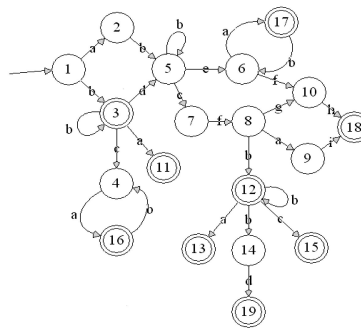
(b) Structured flow chart



(c) N-ary tree



(d) Binary tree



(e) State transition diagram

$$\int_0^{m-1} \int_0^{n-1} \left(\left(\frac{\sin(x^3 - 2\pi x^2 + x)}{\sin(3\pi)^2 \cos(3x)} \right)^{4+xy} \right) dx dy = \sum_{x=0}^{\pi} \frac{\sin(2x^\pi)}{(1-\alpha)^{+y^\pi}}$$

$$\sum_{x=0}^n \int_0^m \frac{2 \tan(3\alpha+1)^\pi}{\sin(x\pi)} dx = \sum_{x=0}^{\alpha} \sum_{y=0}^{\beta} \frac{1}{\alpha} (x^2 + xy - 3)$$

(f) Mathematical expressions

Fig. 13. Example Diagrams.

ease of diagram creation in these applications with traditional graphics editors, both uninterpreted and syntax directed.

ACKNOWLEDGMENTS

Many people have helped in the development of the various components of the Penguins system. Tania Armstrong, Alan Finlay, Richard Helm, Tien Huynh, Andrew Kelly, Toby Sargeant, John Vlissides and Yi Xiao helped in the development of QOCA while Fu Fan Mak and Tom Paton helped with testing and provided feedback for the Penguins system.

REFERENCES

- AHO, A., SETHI, R., AND ULLMAN, J. 1987. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading.
- BORNING, A., MARRIOTT, K., STUCKEY, P., AND XIAO, Y. 1997. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 10th ACM Symposium on User Interface Software and Technology*. 87–96.
- CHOK, S. AND MARRIOTT, K. 1995a. Automatic construction of user interfaces from constraint multiset grammars. In *IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 242–250.
- CHOK, S. AND MARRIOTT, K. 1995b. Parsing visual languages. In *Proceedings of the Eighteenth Australian Computer Science Conference—Australian Computer Science Communication*. vol. 17. 90–98.
- CHOK, S. AND MARRIOTT, K. 1998. Automatic construction of intelligent diagram editors. In *Proceedings of the 11th ACM Symposium on User Interface Software and Technology*. 185–194.
- CHOK, S., MARRIOTT, K., AND PATON, T. 1999. Constraint-based diagram beautification. In *IEEE Symposium on Visual Languages*. IEEE Computer Society Press.
- FERRUCCI, F., PACINI, G., TORTORA, G., TUCCI, M., AND VITIELLO, G. 1991. Efficient parsing of multidimensional structures. In *IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 105–110.
- GOLIN, E. 1991. Parsing visual languages with picture layout grammars. *J. Vis. Lang. Comput.* 2, 371–393.
- GOLIN, E. AND MAGLIERY, T. 1993. A compiler generator for visual languages. In *Proceedings of the 1993 IEEE Workshop on Visual Languages*. IEEE Computer Society Press, 314–321.
- GROSS, M. 1994a. Recognizing and interpreting diagrams in design. In *Advanced Visual Interfaces (AVI)*, T. Catarci, M. Costabile, S. Levialdi, and G. Santucci, Eds. ACM Press.
- GROSS, M. 1994b. Stretch-a-sketch: A dynamic diagrammer. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 232–238.
- GROSS, M. AND DO, E.-L. 1996. Ambiguous intentions: a paper-like interface for creative design. In *ACM 9th Symposium on User Interface Software and Technology (UIST)*. ACM Press, 183–192.
- HELM, R., HUYNH, T., MARRIOTT, K., AND VLISSIDES, J. 1995. An object-oriented architecture for constraint-based graphical editing. In *Object-Oriented Programming for Graphics*, C. Laffra, E. Blake, V. de Mey, and X. Pintado, Eds. Springer-Verlag, 217–238.
- HELM, R. AND MARRIOTT, K. 1986. Declarative graphics. In *Proceedings of the 3rd International Conference on Logic Programming*. LNCS, vol. 225. Springer-Verlag, New York, 513–527.
- HELM, R. AND MARRIOTT, K. 1990. Declarative specification of visual languages. In *IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 98–103.
- HELM, R. AND MARRIOTT, K. 1991. A declarative specification and semantics for visual languages. *J. Vis. Lang. Comput.* 2, 311–331.
- HELM, R., MARRIOTT, K., AND ODERSKY, M. 1991. Building visual language parsers. In *Proceedings of the ACM Conference on Human Factors in Computing (CHI)*. ACM Press, 118–125.
- IGARASHI, T., MATSUOKA, S., KAWACHIYA, S., AND TANAKA, H. 1997. Interactive beautification: A technique for rapid geometric design. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*. Proceedings UIST'97, Banff, Alberta, Canada, 105–114.

- KURLANDER, D. AND FEINER, S. 1992. Interactive constraint-based search and replace. In *ACM Conference on Human Factors in Computing Systems (CHI)*. ACM Press, 609–618.
- KURLANDER, D. AND FEINER, S. 1993. Inferring constraints from multiple snapshots. *ACM Trans. Graphics* 12, 4 (Oct.), 277–304.
- MARRIOTT, K. 1994. Constraint multiset grammars. In *IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 118–125.
- MARRIOTT, K. AND CHOK, S. 2002. Qoca: A constraint solving toolkit for interactive graphical applications. *Constraints* 7, 3/4, 229–254.
- MARRIOTT, K., CHOK, S., AND FINLAY, A. 1998a. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming (CP98)*.
- MARRIOTT, K. AND MEYER, B. 1997. On the classification of visual languages by grammar hierarchies. *J. Vis. Lang. Comput.* 8, 4, 375–402.
- MARRIOTT, K., MEYER, B., AND WITTENBURG, K. 1998b. A survey of visual language specification and recognition. In *Theory of Visual Languages*, K. Marriott and B. Meyer, Eds. Springer-Verlag.
- MINAS, M. AND VIEHSTAEDT, G. 1995. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *IEEE Workshop on Visual Languages*. IEEE Computer Society Press, 203–210.
- PAVLIDIS, T. AND WYK, C. J. V. 1985. An automatic beautifier for drawings and illustrations. In *Computer Graphics*. Vol. 19. Proceedings SIGGRAPH'85, San Francisco, CA, 225–234.
- REKERS, J. AND SCHÜRR, A. 1995. A graph grammar approach to graphical parsing. In *IEEE Symposium on Visual Languages*. IEEE Press, 195–202.
- SERRANO, A. 1995. The use of semantic constraints on diagram editors. In *IEEE International Symposium on Visual Languages*. IEEE Press.
- TUCCI, M., VITIELLO, G., AND COSTAGLIOLA, G. 1994. Parsing nonlinear languages. *IEEE Trans. Soft. Eng.* 20, 720–739.
- ULLMAN, J. 1988. *Principles of Database and Knowledge-Base Systems*. Vol. 1. Computer Science Press.
- ÜSKÜDARLI, S. 1994. Generating visual editors for formally specified languages. In *IEEE Symposium on Visual Languages*. IEEE Computer Society Press, 278–287.
- VANDER ZANDEN, B. T. 1988. Constraint grammars in user interface management systems. In *Proceedings Graphics Interface*. GI'88, Edmonton, Canada, 176–184.
- VANDER ZANDEN, B. T. 1989. Constraint grammars—a new model for specifying graphical applications. In *Human Factors in Computing Systems*. Proceedings SIGCHI'89, Austin, TX, 325–330.
- WEITZMAN, L. AND WITTENBURG, K. 1993. Relational grammars for interactive design. In *IEEE Symposium on Visual Languages*. 4–11.
- WEITZMAN, L. AND WITTENBURG, K. 1994. Automatic presentation of multimedia documents using relational grammars. In *ACM Multimedia Conference*. 443–451.
- WEITZMAN, L. AND WITTENBURG, K. 1996. Grammar-based articulation for multimedia document design. *Multimedia Systems* 4, 99–111.
- WITTENBURG, K. 1992. Earley-style parsing for relational grammars. In *IEEE Workshop on Visual Languages*. IEEE Computer Society Press, 192–199.
- ZHANG, K., ZHANG, D.-Q., AND CAO, J. 2001. Design, construction, and application of a generic visual language generation environment. *IEEE Trans. Soft. Eng.* 27, 4, 289–307.

Received August 1999; revised June 2002; accepted June 2003