Automatic Construction of User Interfaces from Constraint Multiset Grammars

Sitt Sen Chok and Kim Marriott
Monash University, Victoria, Australia

{css,marriott}@cs.monash.edu.au

***Abstract*— We describe tools which automatically generate a sophisticated user interface from a constraint multiset grammar specification of a visual language. The user interface allows the user to construct diagrams in the visual language from primitive tokens such as text, lines, rectangles, or circles. These tokens are incrementally parsed into sub-diagrams. During parsing, automatic error correction removes geometric errors, providing feedback about what has been recognized. The user may add or delete tokens and may manipulate components of the diagram. During manipulation, the user interface takes account of the semantics of the diagram, preserving the geometric relationships dictated by the visual language. Our research has particular application to user interfaces for pen-based computers.**

Keywords: Parsing, Constraint, Graphic Editor, Visual Languages

I. Introduction

A promising new direction in human-computer interaction are user interfaces based on application specific visual language composed of text and diagrams. Such an interface allows users to express themselves naturally and directly in the visual *lingua franca* of their application domain such as state transition diagrams, musical notation, structural-chemical formulae or mathematical equations. However, constructing a user interface for each visual language is complex and time-consuming.

Here we describe two tools – a parser generator and a graphics editor – which automatically generate a sophisticated user interface from a grammatical specification of a visual language. To construct a user interface for a particular visual language, a grammar specifying the language is feed into the parser generator. The parser generator produces a parser which is combined with a constraint-based graphics editor to give the user interface. The resulting user interface allows the user to construct diagrams in the visual language from text, lines, rectangles, or circles. These are incrementally parsed into sub-diagrams. During parsing, automatic error correction removes geometric errors in the original diagram, providing feedback about what has been recognized. The user may add or delete tokens and may manipulate components of the diagram. During manipulation, the user interface takes account of the semantics of the diagram, preserving the geometric relationships dictated by the visual language.

Our approach is based on *constraint multiset grammars (CMGs)* [12]. These give a general, high-level framework for the definition of visual languages. Productions in a CMG have the form

$$P ::= P_1, ..., P_n \ where \ C$$

indicating that the non-terminal token P can be recognized from the multiset of token P_1, ..., P_n whenever the attributes of all tokens satisfy the constraints C. The constraints enable information about spatial layout and relationships to be naturally encoded in the grammar. In particular, constraints may also refer to other sub-diagrams in the picture by requiring the existence, or non-existence, of tokens.

The main tool to construct the user interface is the parser generator. This takes a CMG and produces a parser for the visual language described by the CMG. The parser has two novel features which suit it to interactive applications:

- **Incremental parsing**. Whenever a token is input, the parser updates the parse tree. This is quite difficult because negative constraints mean that parsing is non-monotonic – adding a new token may make an earlier reduction invalid.
- **Automatic geometric error correction**. Geometric error correction occurs during the incremental parsing so as to provide immediate feedback to the user about what has been recognised. This is important because 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.

The second tool is a constraint-based graphic editor. This is combined with a parser to give a complete user interface based on the visual language. The editor allows the user to add or delete tokens and to manipulate components of the diagram. During manipulation, the editor takes account of the semantics of the diagram. For instance, imagine that a circle and text form a state in a state transition diagram. When the circle is moved then the text also moves. The editor does this by maintaining the constraints in the parse tree as these capture the semantic relationships between the tokens in the diagram. When a token is manipulated, re-solving the constraints and redrawing the other tokens ensures that the manipulation preserves the semantics of the diagram.

All components of the user interface are written in C++. Empirical results given here for three example visual languages – state transition diagrams, mathematical equations and trees – show that the generated user interface is very efficient. New tokens are incrementally parsed in milliseconds and whole diagrams parsed in much less than one second. Manipulation of diagrams is also very quick.

The most closely related work is that of Gross [8] who describes a pen-based drawing program which recognizes simple relationships between tokens and preserves these during manipulation of the diagram. He allows the user to customize the relationships which are recognized. Our work extends his by automating construction of the recognizer from a grammar describing the underlying visual language, automating error correction and by allowing more powerful relationships such as negative constraints in the recognizer.

Research into two-dimensional languages and static parsing has a long history. Constraint multiset grammars [12], are related to logic grammars used in natural language parsing [1], definite set grammars used in visual language generation [10], [11] and visual mappings [13]. CMGs are also related to attributed multi-set grammars [7], [5], [6] and relation grammars [4]. Negative constraints are a key difference between CMGs and these other formalisms for specifying visual languages. Without negation, it is difficult to specify many naturally occurring visual languages, such as state transition diagrams or binary trees, with a deterministic grammar. Once a non-deterministic grammar is used, efficiency is compromised because the parsing problem becomes intractable. However, constraint multiset grammars are expressive enough to allow deterministic parsing of these languages. A parser generator which produces a static parser for a deterministic CMG is described in [3] .

Incremental parsing of visual languages has not been widely studied. The papers [17] , [18] describe incremental parsers for very restrictive classes of visual languages. [14] gives an incremental parsing algorithm for CMGs without negative constraints. Here we extend this algorithm to handle negative constraints. This is non-trivial because parsing is now non-monotonic and requires us to keep track of dependencies in the parse tree.

To our knowledge this is the first work to consider geometric error correction in visual language parsing in a systematic way.

Since SketchPad [15] there has been a lot of research into constraint based graphic editors. The most significant difference of our work, is that constraints are generated automatically during parsing. The user does not need to provide the constraints as the user interface is intelligent enough to infer them from knowledge about the particular visual language. Related work also includes syntax based graphic editors [16]. However our graphic editor is not syntax based – it is the parser which is concerned with syntax. This means that diagrams can be drawn in any order.

II. CONSTRAINT MULTISET GRAMMARS

In this section we provide an informal introduction to constraint multiset grammars (CMGs). We do this by means of an example grammar for state transition diagrams. Given the diagram in Fig. 3(a) our example grammar can be used to recognize that this represents a finite state automata with start state $S1$, final states $S2$, $S6$ and $S8$, normal states $S3$, $S4$, $S5$ and $S7$ and transitions $\langle S1, a, S3\rangle$, $\langle S1, b, S2\rangle$ etc. Productions in the grammar specify how states and transitions are composed from the terminal symbols which are circles, texts, lines and pointer.

Every symbol in a CMG has a *(token) type* with associated *attributes*. In the example grammar the types are circle, line, text, arrow, start_arc, arc, state, transition, and fsa. The attributes of a circle are its radius and midpoint and the attributes of text is its midpoint and the string of text. Types are either *terminal* or *non-terminal*. There is one distinguished non-terminal type called the *start* type. Attributes are divided into two classes: *geometric* attributes such as *radius* or *midpoint*, and *non-geometric* attributes such as the *kind* attribute of a state and the *string* attribute of text. We use $S : T$ to specify that variable S has type T, and $S.A$ to indicate the A attribute of S.

A *constraint multiset grammar (CMG)* consists of production rules which define how to recognize non-terminal types. Production rules in a CMG have the form:

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

where T is a non-terminal type with attributes $\vec{A}$; $V_1, ..., V_n, V'_1, ..., V'_m$ are variables and $T_1, ..., T_n$, T'_1, ..., T'_m are types; C is a constraint and $\vec{E}$ is a sequence of expressions. The existentially quantified variables are optional. We require that the production rules be well-typed in the obvious way – attributes of variables must match the attributes of the variable's type.

The first production in the grammar for state transition diagrams, specifies that an arc is composed of an arrow and a text label. The mid point of the arrow should be close to the midpoint of the text label. The function `close` is defined by an auxiliary C++ Boolean function. It specifies that two points are close if the Euclidean distance between them is less than some threshold. If the constraint holds for arrow A and text T then A and T are reduced to an arc whose attributes are given by assignment statements at the end of the production.

```
arc(point start, point end, point mid,
      string text) ::=
   A:arrow, T:text where (
      close(A.mid, T.mid)
   ) {
      start = A.start;
      end   = A.end;
      mid   = A.mid;
      text  = T.text;
   }
```

A transition is an arc between two states.

```
transition(string start, string label,
      string end) ::=
   A:arc where (
      exist S1:state, S2:state where (
         on_circle(A.start, S1.mid, S1.radius) &&
         on_circle(A.end, S2.mid, S2.radius)
      )
   ) {
      start = S1.name;
      label = A.text;
      end   = S2.name;
   }
```

The function `on_circle(P,C,R)` holds if `P` is on the circle with center `C` and radius `R`. The above production uses *existential quantification*. This allows a production to test for the existence of sub-diagrams without having to reduce them. In this example, it is used to test that the arc is between two states.

Recognizing states is more complex. The reason is that we would like deterministic productions which distinguish between the three different kinds of state: final, start and normal. A final state is defined as two concentric circles with an enclosed text label.

```
state(point mid, integer radius, string name,
      string kind) ::=
   C1:circle, C2:circle, T:text where (
      C1.mid == C2.mid &&
      close(T.mid, C1.mid)
   ) {
      mid    = C1.mid;
      radius = max(C1.radius, C2.radius);
      name   = T.text;
      kind   = "final";
   }
```

Similarly, a start state is composed of a "start arc" and a circle containing a text label. The start arc points to the circle. There should be no other circle around the text label. The start arc should be unlabeled. These conditions are expressed using *negative constraints* which check that sub-diagrams do not exist.

```
start_arc(point end) ::=
   A:arrow where (
      not exist R:text where (close(R.mid, A.mid))
   ) {
      end = A.end;
   }

state(point mid, integer radius, string name,
      string kind) ::=
   A:start_arc, C:circle, T:text where (
      not exist M:circle where (
         M.mid == C.mid) &&
      on_circle(A.end, C.mid, C.radius) &&
      close(T.mid, C.mid)
   ) {
      mid    = C.mid;
      radius = C.radius;
      name   = T.text;
      kind   = "start";
   }
```

Similarly we can give a production to recognize a normal state. A collection production is used to collect all of the sub-diagrams of the same type which satisfy the constraints into a multiset. The exact order in which the sub-diagrams are "collected" is not defined. The final rule in the grammar is the finite state automaton production. A finite state automata is composed of states and transitions which are collected together.

```
fsa(string states, string transitions) ::=
   all S:state, all T:transition where (true) {
      states      = { S.name };
      transitions = { T.start, T.label, T.end };
   }
```

A key feature of CMGs is their use of constraints over the type attributes to define the relationships between a diagram and its components. In subsequent sections we shall see that it is because of constraints that we can perform automatic error correction and semantics preserving manipulation of diagrams. We shall need to distinguish between *geometric constraints* over geometric attributes and *non-geometric constraints*. Geometric constraints must, in essence, be a conjunction of arithmetic equality and inequality constraints. The grammar writer must provide C++ functions for the primitive constraints and functions and C++ classes for the attributes. In addition for geometric constraints, the grammar writer must detail the conjunction of arithmetic equality and inequality constraints the constraint corresponds to. As we have seen, more complex "constraints" are built from the primitive constraints using conjunction and negation.

III. INCREMENTAL PARSING

In the last section we introduced CMGs. In this section we develop algorithms for incremental parsing for CMGs. We first consider how to parse incrementally as tokens are added for *positive* CMGs, that is CMGs without negative constraints, then CMGs with negative constraints. Then we consider how to handle incremental deletion of tokens. Finally we give sufficient conditions for correctness of the algorithms.

A. Positive CMGs

CMGs are used to recognize diagrams where a diagram is represented as a multiset of terminal tokens. A *token* of type T assigns a value to each attribute. In C++ terms, a token of type T is an instance of the class T. A multiset of tokens is called a *sentence*.

Recognition is performed by taking a multiset of terminal tokens and repeatedly using production rules in the grammar to combine tokens into larger and larger parse trees until a single parse tree with a token of start type as its root is obtained. Essentially, a *parse tree* is a tree of tokens and in which each internal node has an associated production. The *yield* of a parse tree or collection of parse trees is the multiset of leaf tokens.

```
procedure Parse(t)
    AddToken($t$)
    for each SCC in order do
        repeat
            for each rule $R$ in the SCC do
                EvalRule(R)
            end for
        until $ParseForest$ is unchanged
    end for
end
```

Fig. 1. Incremental Parsing Algorithm

The main global data structure in the incremental parser for CMGs is a *ParseForest* which represents a multiset of parse trees. The $ParseForest$ is a list of nodes of form

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

where t is the token constructed from reducing the parse tree nodes N_1,...,N_n using the production rule *Rule* in the presence of the existentially quantified nodes N'_1,...,N'_m. The flag *Root* is true if this node is the root of the parse tree. Terminal tokens are represented by leaf nodes, that is nodes of the form $\langle t, [], [], NULL, Root \rangle$.

Parsing is the process of combining parse trees. More exactly, if the $parseForest$ contains root nodes N_1,...,N_n and possibly non-root nodes N'_1,...,N'_m and there is production rule, *Rule*,

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

for which the assignment θ from the variables $V_1, ..., V_n$ V'_1 , ..., V'_m to the tokens in N_1,...,N_n and N'_1,...,N'_m respectively satisfies C, then the new node

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

where token t is of type T with attributes $\vec{A}$ set to $\theta(E)$ can be added to the $ParseForest$. This is called a *reduction.* Note that after the reduction the nodes N_1,...,N_n are no longer root nodes.

For efficiency in parsing, the parser does not consider all production rules at a time. Instead the parser generator first computes the "call-graph" of dependencies between productions in CMG. A rule $R1$ is *call-dependent* on another rule $R2$ if a token in the body of $R1$ has the same type as the token in the head of $R2$. The *strongly connected components (SCCs)* of the call graph are then computed and ordered. Rules in higher SCCs depend on rules in lower SCCs. The parse forest is evaluated using rules in each SCC, starting from the lowest SCC and moving higher.

The incremental algorithm for parsing CMGs is given in Algorithm 1. When a new token t is drawn by the user, the parser is called with $Parse(t)$. The call $Parse(t)$ first adds the terminal token t to the $parseForest$ and then iterates through the SCCs applying the rules in each SCC until the $ParseForest$ is unchanged.

The call $EvalRule(R)$ performs reductions with the production rule R. It repeatedly tries assignments θ of nodes in the $ParseForest$ to the variables in R. For efficiency, at least one variable must be assigned a node which was only created in the last iteration of the repeat-until loop in $Parse$. This is done by a nested loop which tries each combination to see if it satisfies the constraints. If the constraints are satisfied, a new token will be inserted into the token table using $AddToken$.

B. CMGs with Negative Constraints

Negative constraints make incremental parsing rather more difficult. The problem is that addition of a new token can make previous reductions invalid.

For example, imagine that we have an arrow, text and circle which have been recognized as a start state using the productions in the example CMG. This involves two reductions, the first recognizing the the arrow is a start_arc, the second recognizing the start state from the text circle and start_arc. Now the start_arc production has a negative constraint that refers to a text token. Imagine that the user now adds a text token close to the midpoint of the arrow. This new token makes the previous reductions invalid as the arrow is no longer a start_arc but rather an arc and the state is not a start state but a normal state.

The incremental parser given in the last section is modified to handle negative constraints in three ways.

First, when the parser is created by the parser generator, a *negative constraint table* is constructed. For each token type this keeps a list of productions which depend on that token type through a negative constraint. For example, the start_arc production, depends negatively on the text token type.

Second, parse tree nodes have a field added which is the list of other parse tree nodes which depend on them. These dependencies can be through both existentially and unquantified variables. Thus the parse forest now has links from parent to child and child to parent in each parse tree. For instance, in the above example there will be a link from the start_arc node to the start state node.

Third, the routine $AddToken$ is now modified so as to remove previous reductions which the new token to be added makes invalid. Adding a new token triggers a search in the negative constraint table for an entry that may indicates negative dependency on the new token. If such an entry is found, the relevant constraints of all the tokens constructed with that production are re-checked. If the constraint within a production constructing a token become false, then the token is removed from the parse tree using $RemoveToken$. Essentially, $RemoveToken(t)$ removes the child-parent links to t for its children and resets the children assigned to unquantified variables so that they become root nodes. $RemoveToken$ is then called to recursively remove all token which depend on t.

For instance, when the text token is added in our example, $AddToken$ looks at the negative constraint table and determines that the production for start_arc depends negatively on text tokens. This causes the production giving the start_arc token in the parse tree to be reevaluated. It fails, indicating that the reduction is no longer valid. The parent dependency links are traversed and the start_arc and start state tokens are removed from the parse forest and their children have the links to these nodes removed and are reset to be root nodes.

C. Incremental Deletion

Given the *RemoveToken* routine described in the last section it is easy to now handle deletion of a token. Whenever a token t is deleted, all that needs to be done is to call $RemoveToken(t)$ and then to start the main parsing loop.

For example, imagine that two concentric circles and some text have been recognized as a final state. If one of the circles is removed, then *RemoveToken* will remove the circle and the dependent final state token from the parse forest. Restarting the parsing will cause the circle and text to be recognized as now being a normal state.

D. Correctness

The incremental parsing algorithms we have given are not guaranteed to terminate for various pathological types of CMGs. We now identify the class of CMGs for which they are correct and will terminate. There are three restrictions.

The first restriction is that the CMG is *cycle-free*, in the sense that there are no infinite reductions in which a single token is repeatedly reduced. This means that there cannot be an infinite sequence of reductions between the parse trees, and consequently ensures termination of the parser.

The second restriction is to do with negative constraints. The problem is that in a production may directly or indirectly create the token it depends on negatively. That is, there may be a cycle in the call graph of types which passes through a negative constraint. For example, the following production has such a cycle.

```
p() ::= r : s where
    (not exist t:p where (true))
```

The problem is that subsequent reductions will produce new tokens which may invalidate an earlier reduction which involved a negative constraint. In our implementation this will lead to an infinite loop in which a token is repeatedly created then deleted. We feel that it is problematic to give such CMGs a semantics and instead restrict attention to CMGs that are *stratified* in the sense that types only depend negatively on types which are strictly lower in the call graph. The parser generator checks that the SCCs are stratified. For more details see [3].

The third restriction is that the CMG is *deterministic*. A stratifiable CMG is deterministic if each maximal sequence of reductions which respect the stratification, give the same multiset of parse trees. This means that different choices of which parse trees to reduce give the same outcome.

All practical examples of CMGs that we have seen are cycle-free, stratified and deterministic. For such CMGs the algorithms we have given are correct.

IV. ERROR CORRECTION

In this section we describe how we can modify the intcremental parser described in the last section so that it automatically corrects errors involving the geometry between tokens in the diagram.

We define the *geometric error* of a diagram to be the distance from the "closest" diagram in the language. More exactly, we assume a metric *dist* which measures the distance between two geometric attribute values. For semantic attributes *dist* is 0 if they are the same and ∞ if they are not. The metric is extended to sentences in the obvious way. We use ∞ to indicate that the sentences have different structure, and so are not comparable. Let S be the sequence of tokens $T_1(\vec{\theta}_1), ..., T_n(\vec{\theta}_n)$ and S' the sequence $T'_1(\vec{\theta}'_1), ..., T'_m(\vec{\theta}'_m)$. The *distance* between S and S', $dist(S, S')$, is

$$\sum_{i=1}^{n} dist(\theta_i, \theta'_i)$$

if $m = n$ and for all i, $T_i = T'_i$. Otherwise it is ∞. The *distance* between sentences S and S' is the minimum of the distances between all permutations of S and S'. The *geometric error* of a sentence S for CMG, G, is

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

Our notion of geometric error distance is analogous to error distance in string languages. In string languages, the difference between 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' [2]. For both visual language and string parsing, error correction can, therefore, be understood as the problem of finding the sentence in the language which is closest to the diagram.

In the case of CMGs it is theoretically possible to perform perfect geometric error correction in parsing. We now describe an algorithm to do this.

A *token skeleton* is like a token except that instead of attributes it contains distinct *attribute variables* as arguments to all geometric attributes. A *parse tree skeleton* and *sentence skeleton* are like parse trees and sentences except that they contain token skeletons instead of tokens.

A parse tree skeleton or sentence skeleton represents a class of parse trees or sentences. Members of the class are obtained by applying an *attribute variable assignment*, θ, to the parse tree or sentence. The assignment maps each geometric attribute variable to a value. Of course, in the case of a parse tree, the assignment must satisfy the constraints in the productions associated with the parse tree. Every parse tree P or sentence S has a skeleton, denoted by $ske(P)$ or $ske(S)$ respectively, which it is an instance of. This skeleton is unique up to renaming of the attribute variables.

To find the closest sentence in the language of a CMG to an input sentence S, we iterate through all of the skeleton parse trees whose yield is $ske(S)$ and find the skeleton parse tree of the sentence which is closest to S. Because

```
procedure ErrorCorrect($S$)
    $S1 := ske(S)$
    $error := \infty$
    $closest := NULL$
    for each skeleton parse tree $P$ with yield $S1$ do
        compute $\theta$ which minimizes $dist(S, \theta(S1))$
            with respect to $con(P)$
        if $dist(S, \theta(S1)) < error$
            $error := dist(S, \theta(S1))$
            $closest := \theta(S1)$
        end if
    end for
    return $closest$
```

Fig. 2. Algorithm for Perfect Error Correction

the CMG is cycle-free there are only a finite number of skeleton parse trees with yield $ske(S)$. For each skeleton parse tree, we find the instance which yields the sentence closest to S. Given a skeleton parse tree P for $ske(S)$, we extract the geometric constraints in P, denoted by $con(P)$ and find the assignment θ to the attribute variables in P which satisfies the constraints in $con(P)$ and whose instance $\theta(S1)$ is closest to S. Note that geometric attribute assignments give rise to equality constraints in $con(P)$. The algorithm is given in Fig. 2.

Unfortunately the algorithm we have given for perfect error correction is not very practical. There are two reasons. First is the need to consider all skeleton parse trees of a skeleton sentence. Second the constraint minimization problem can be extremely expensive to solve for arbitrary arithmetic constraints.

The algorithm we use for error correction uses several heuristics so as to achieve reasonable efficiency. First, it only considers a single skeleton parse tree. An instance of this parse tree, $P1$, is constructed by the parsing algorithm in the last section. The only difference is that equality constraints in the CMG are treated "sloppily". That is, when parsing, an equality $s = t$ is regarded as satisfied as long as $dist(s,t)$ is less than some threshold error distance. The algorithm then finds the assignment θ which is the closest instance of skeleton of $P1$ to S. It does not consider all constraints in $ske(P1)$, however. The optimization problem is only solved with respect to the linear arithmetic equalities in the positive constraints of $ske(P1)$. The non-linear equalities are approximated by linear equalities and inequalities and arithmetic constraints in negative constraints in $P1$ are ignored. Optimization problems over linear equalities with a guadratic optimization problem can be solved very efficiently. We use a constraint solver which is described in more detail in the next section. After θ is computed, we check that it solves the inequalities and negative constraints. If it does, then it has been safe to ignore them in the optimization problem, and θ is the closest instance of skeleton of $P1$ to S. If the inequalities and negative constraints are not satisfied, an error is returned and the senetence is not recognized. In practice, if S is close enough to a sentence in the grammar, the inequalities and negative constraints will remain satisfied.

To illustrate our algorithm consider that we have input three tokens – a circle $C1$ with radius 2 and center $(10.9, 11.1)$, a circle $C2$ with radius 2.1 and center $(11.0, 10.9)$, and text T with center $(11.1, 11.0)$. Using a tolerance of 0.25 when testing equality constraints, incremental parsing will recognize that this is a final state FS. The geometric constraints in the parse tree are:

$$\begin{aligned}
&C1x = C2x \wedge C1y = C2y \wedge \\
&C1x = Tx \wedge C1y = Ty \wedge \\
&FSx = C1x \wedge FSy = C1y \wedge FSr = C2r
\end{aligned}$$

where Ox, Oy and Or are attribute variables for the x and y position of the object O's center and its radius respectively. To find the error corrected position, the parser uses the constraint solver to find the assignment to attribute variables which minimizes

$$\begin{aligned}
&(C1r - 2.0)^2 + (C1x - 10.9)^2 + (C1y - 11.1)^2 + \\
&(C2r - 2.1)^2 + (C2x - 11.0)^2 + (C2y - 10.9)^2 + \\
&(Tx - 11.0)^2 + (Ty - 11.0)^2
\end{aligned}$$

with respect to the above constraints. This is just $C1r = 2.0$, $FSr = C2r = 2.1$, $FSx = C1x = C2x = Tx = 11.0$, $Fsy = C1y = C2y = Ty = 11.0$. The diagram will be redrawn with the tokens at their new position indicating to the user that a final state has been recognized.

V. Graphic Editor

In any interactive application the user should be able to add or delete tokens and also move tokens in the diagram. A user interface which provides these capabilities is obtained by combining a generated parser, of the type described in the last section, with a constraint-based graphics editor. The editor provides the high level user interface but calls the parser and also examines constraints in the current parse tree.

Addition and deletion of tokens is essentially done by calling the parsing functions *Parse* and *RemoveToken*. Diagram manipulation is more difficult as manipulation should take into account the semantics of the diagram. For instance, imagine that a circle and text form a state in a state transition diagram. When the circle is moved then the text should also moves.

We use the *metric space model* as a simple metaphor for manipulation of diagrams. This is possible because the underlying grammatical formalism, CMGs, are couched in terms of constraints which explicitly capture the geometric relationships in a diagram. In the metric space model of constraint interaction, at any time there is a set of constraints over some variables, a metric which gives the distance between two assignments to the variables, and a current solution which is an assignment to the variable that satisfies the constraints. The variables correspond to graphical attributes of the graphical objects in the diagram and the diagram on the graphics display is, therefore, just 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

TABLE I
EMPIRICAL EVALUATION OF PARSER GENERATOR

	State	Math	Tree
No. of productions	11	21	7
Generation time (sec)	6	6	5
Compile time (min)	8.04	8.37	8.25

of constraints in which case the new solution is the assignment which is closest to the old assignment and which satisfies the new constraint set. This occurs when tokens are added and parsing generates new constraints, and error correction finds the closest solution. Second, a constraint may be deleted from the current set of constraints, in which case the current solution remains unchanged. This occurs when tokens are deleted. Finally, the current solution may be manipulated by "suggesting" a value for several of the variables. The new solution is the assignment which is closest to the old assignment and to the requested variable values and which satisfies the constraint set. The constraint set remains unchanged. This is what happens when tokens are manipulated.

In our implementation of the graphic editor and error correction in the parser we have used a revised version of QOCA [9]. This is a C++ constraint solving toolkit designed to support the metric space model paradigm over various classes of arithemetic constraints. The current implementation of QOCA supports linear arithmetic equalities and uses the Euclidean space metric to measure the distance between solutions.

VI. EMPIRICAL EVALUATION

In this section we give some preliminary empirical data about the efficiency of the generated user interfaces. The current versions of the parser generator, called VisualGen, and the graphics editor and QOCA-II run under the Microsoft Window 3.11 environment on an IBM 80486DX2-66 machine. Timers are deliberately embedded into the program to record the time taken for some of the given data in this section.

We have tested the system with thre example CMGs. The first example is the grammar for state transition diagrams which was described earlier. Fig. 3 (a) is an example input used in the test. The second example is a grammar for simple mathematical expressions. Fig. 3 (b) shows the sample input for this grammar. The third, and final example, is a grammar for binary trees. Fig. 3 (c) shows a sample input for this grammar. The first test details how long the parser generator takes to generate and compile the incremental parser from the example CMG. Table I gives the empirical results.

The second test examines how long parsing takes without error correction. Each of the three example parsers is tested with input diagram of different sizes. The size of an input diagram is measured by the number of terminal tokens used to build the diagram. For each visual language, diagrams of approximate size 10, 20, 30, 40, 50 have been tested for the time taken to parse the whole diagram statically (S) and the average time (I) taken for incremental parsing when tokens are added one at a time.

TABLE II
EMPIRICAL EVALUATION OF THE INCREMENTAL PARSER

	State		Math		Tree	
Size	S	I	S	I	S	I
10	0.00	0.10	0.00	0.10	0.00	0.00
20	2.00	0.10	1.00	0.35	0.00	0.05
30	1.00	0.13	2.00	0.33	1.00	0.20
40	1.00	0.15	2.00	0.30	3.00	0.38
50	1.00	0.14	4.00	0.32	3.00	0.35

TABLE III
EMPIRICAL EVALUATION OF THE INCREMENTAL PARSER WITH ERROR CORRECTION

	State	Math	Tree
10	1.0	4.0	2.0
20	5.7	5.7	3.3
30	4.6	5.7	3.5
40	5.6	9.3	3.6
50	4.6	8.6	3.6

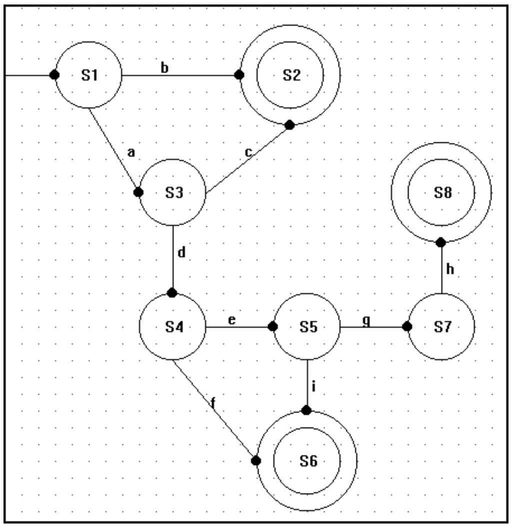
Table II presents the empirical result for each generated visual language parser. All times are measured in milliseconds.

The third test examines how long parsing takes with error correction. Again each of the three example parsers is tested with input diagram of different sizes. We give the average time taken for incremental parsing with error correction when tokens are added one at a time in Table III. All times are measured in milliseconds.

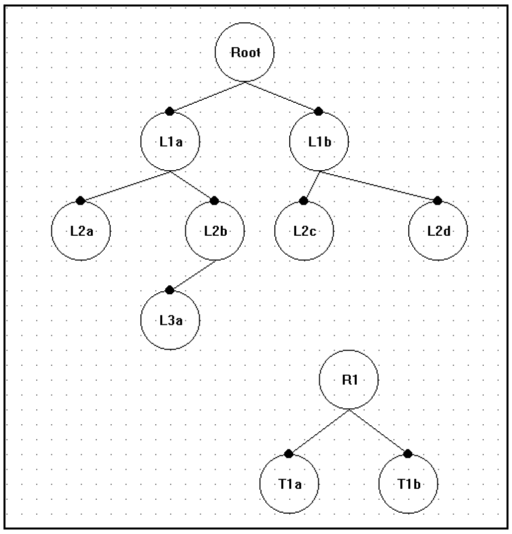
CONCLUSION

We have described two tools – a parser generator and a graphics editor – which allow the automatic generation of a sophisticated user interface from a grammatical specification of a visual language. The user interface allows the user to construct diagrams in the visual language from text, lines, rectangles, or circles. These are incrementally parsed into sub-diagrams. During parsing, automatic error correction removes geometric errors in the original diagram, providing feedback about what has been recognized. The user may add or delete tokens and may manipulate components of the diagram. Empirical tests verify that the user interface is reasonably efficient, with complete diagrams parsed in much less than one second, and incremental parsing even with error correction taking only a few milliseconds when a token is added.

One of the main applications of our interface is to pen-based computers. Recent hardware advances have brought interactive graphic tablets and pen-based notepad computers into the market place. This new technology, however, while offering great potential has not yet been very successful. One of the main reasons is that software for pen-based computers is still immature: gesture recognition is poor and few applications make use of the new capabilities of the pen. Our tools promise to support construction of application programs in which the user communicates with the computer using an application specific visual language composed of handwritten text and diagrams.


S1
b
S2
a
c
S3
S8
d
h
S4
e
S5
g
S7
f
i
S6

(a)


Root
L1a
L1b
L2a
L2b
L2c
L2d
L3a
R1
T1a
T1b

(b)