Syntax Analysis for Diagram Editors: A Constraint Satisfaction Problem

Mark Minas
Universitat der Bundeswehr Munchen
85577 Neubiberg, Germany
Mark.Minas@unibw.de

ABSTRACT
Most visual languages of today have a meta-model as syntax specification. Before meta-models became popular, grammars of various flavors have been used for specifying a visual language's syntax. Grammars are more complicated to build than meta-models, but grammars allow for parsing of visual sentences which is necessary for building free-hand editors. Parsing has not yet been considered for meta-model-based specifications. Such visual editors support only structured editing so far. This paper shows that the syntax analysis problem ("parsing") for meta-model-based language specifications can be transformed into a constraint satisfaction problem and solved that way. This approach, therefore, allows for easy free-hand editing and, at the same time, easy meta-model-based language specifications. The efficiency of the approach has been demonstrated by realization in the diagram editor generator DiaGen.

1. INTRODUCTION
Visual editors are part of many sophisticated applications that allow to create and modify internal data structures through such editors. For instance, model driven development depends on tool support for visual models that are usually domain-specific. The need for domain-specific visual editors, therefore, is currently increasing. Several aspects influence the design of visual editors in this context. Three of them are considered here: First, the editor has to support a certain visual language. How should the language be specified? And second, the editor should be tightly integrated with the rest of the application. How easy is communication between diagram editor and application? The easiest solution would be an editor that operates directly on the application's data structures. And finally, how does the user interact with the editor? Usually, structured editing is distinguished from free-hand editing. Structured editors offer the user some operations that transform correct diagrams into (other) correct diagrams. Free-hand editors, on the other hand, allow to arrange diagram components on the screen without any restrictions. The editor has to find out whether the drawing is correct and what is its meaning. Therefore, structured editors offer more guidance to the user which may make editing easy. However, free-hand editors leave more freedom to the user when she edits diagrams. Allowing for (temporarily) incorrect diagrams may even make the editing process easier.

Traditionally, some kind of grammar has been used to specify visual languages for editors providing free-hand as well as structured editors. Some examples are extended positional grammars in VLDesk [3] and constraint multiset grammars in PENGUINS [2] for free-hand editing, and hypergraph grammars in DiaGen [9, 10] for free-hand as well as structured editing. Grammars describe a language's syntax by rules that are applied, starting at a certain starting state, to derive valid sentences of the language. Syntactically analyzing diagrams means trying to find a sequence of rule applications that derive the diagram or some representation of it. Communication between diagram editor and application requires building an abstract representation of the diagram by attribute evaluation, i.e., additional specification and evaluation efforts are necessary.

However, most graph-like languages of today, i.e., the majority of visual languages, at least in computer science, have a meta-model as syntax specification. Meta-models are essentially class diagrams of the data structures that are visualized by diagrams. Some examples for meta-model-based approaches are AToM³ [4], Pounamu [11], and MetaEdit+ [8]. There are several reasons for the success of meta-models. One of them is the training of users in specifying data structures with class diagrams. On the contrary, writing grammars appears to be much more complicated. Moreover, the visual modeling languages of the UML are specified by meta-models. Extending such visual languages then requires to use and extend their meta-models instead of writing grammars.

However, all of the meta-model-based approaches offer structured editing only. We are not aware of any approach supporting free-hand editing although that editing mode would offer more freedom to the user. This paper tries to close this gap by proposing a meta-modeling approach that offers free-hand editing, too. Similar to grammar-based free-hand editors, some syntax analysis is required. We show that syntax analysis based on meta-models is actually a constraint satisfaction problem, i.e., there is already a comprehensive variety of algorithms for solving the syntax
This approach, therefore, allows for easy free-hand editing and, at the same time, easy meta-model-based language specifications. The efficiency of the approach has been demonstrated by realization in the diagram editor generator DiaGen.

The next section describes syntax specification with metamodels which are class diagrams of the edited object structure. Syntax analysis and its mapping to a constraint satisfaction problem are discussed in Section 3. Section 4 then briefly introduces the diagram editor generator DiaGen that realizes the presented approach. Section 5 concludes the paper.

2. SYNTAX SPECIFICATION WITH CLASS DIAGRAMS

Class diagrams are primarily used for specification of object structures that are created by instantiation of classes and associations between them. Subclassing is used to create subtypes that inherit all features of their superclasses. Additional constraints on the object structures may be used to restrict the set of all valid object structures even more. The Unified Modeling Language UML allows for expressing such constraints using the Object Constraint Language OCL. In this paper, we are considering diagram editors that allow to visually create and modify such object structures. Hence, class diagrams together with additional constraints are the specification of the corresponding diagram language. This is a syntax specification rather than a semantics specification because object structure behavior is specified with other means, e.g., statecharts. In the following, we will consider class diagrams without additional constraints in order to simplify discussions, but we keep in mind that those additional constraints are in general necessary for a complete specification.

Class diagrams specify object structures, but they do not specify a diagram language’s syntax directly. Syntax specification by class diagrams is limited to those diagram languages that provide a reasonably simple mapping between a diagram and its underlying object structure. This is particularly true for graph-like diagram languages that have an almost one-to-one relation between diagram components and objects. However, most visual languages of today are graph-like, in particular those of the UML. In the following, we require that a diagram is easily and uniquely mapped to a graph that resembles its object structure. When drawing a diagram in free-hand mode, its representing graph is drawn at the same time. We will briefly discuss a technique to realize such a correspondence between a diagram and its representing graph in Section 4.

We will use simple trees as an example throughout the paper. Fig. 1 shows a class diagram which is similar to the class diagram for the composite design pattern. The class diagram contains class Node as the abstract base class of a tree’s nodes. Each node has a member attribute Name whose type is omitted here. Concrete classes are Root, InnerNode, Leaf, and SingleNode. The latter class represents the only node of trees consisting of just one node. The other classes are used to represent nodes in trees that contain edges also. Root represents root nodes that have at least one child. Leaf nodes with at least one parent are represented by class Leaf. The remaining nodes, i.e., nodes that have a parent as well as at least one child, are represented by InnerNode. Abstract superclasses Parent and Child represent Nodes that act as parents resp. children. Please note that InnerNode is subclass of both classes. Parent-child-relations are represented by the association between Parent and Child with the roles child resp. parent. Please note the cardinalities; they specify that a parent has to have at least one child, and a child has to have exactly one parent.

This class diagram may appear overly complicated for simple trees, but it has been chosen in order to demonstrate certain aspects of syntax analysis that could not be demonstrated for a simpler class diagram. Besides, distinguishing roots, leaves and inner nodes may be required by the application. Methods are omitted here as we do not consider behavior, but distinguishing different types of nodes may be a requirement that leads to a class diagram like in Fig. 1.

Please note that the class diagram does not completely specifies trees. On the one hand, it does not prevent object structures from being circular, on the other hand, data structures need not be connected. The first problem could be solved in an unsystematic way by turning the association into a composite association which, by definition, prohibits circles. The second problem, however, can be described by additional constraints only. We omit them and, therefore, specify sets of trees instead of trees.

Fig. 2 shows a valid tree and the UML object diagram of its object structure. The node names are also used as object identifiers. Checking the object structure and, therefore, the represented tree for syntactic correctness means checking whether the object structure can be created by instantiation of the class diagram. That is obviously true here, i.e., the object structure and the represented diagram are syntactically correct. However, as shown in Fig. 2, root node, inner node and leaves of a tree are not distinguished visu-
ally. Yet, we can deduce from context information that a is a root node, b an inner node, and c as well as d are leaves.

We have already discussed that distinguishing different kinds of nodes by different classes may be necessary, even if all nodes are visualized in the same way. A user may turn a leaf node into an inner node by adding a new child to the former leaf, i.e., the component that was internally represented by a Leaf instance has to be represented by an InnerNode instance now. Free-hand editors, therefore, cannot unchangeably bind an internal representation to the visual component. The editor rather has to reconsider this binding after each diagram modification. This problem is easily solved by binding the common type of all possible internal representations, i.e., class Node in the example, to the visual component. The editor then has to check correctness by deducing the concrete types and checking whether the resulting object structure fits to the class diagram. This problem, deducing the concrete classes and checking against the class diagram is the syntax analysis problem that is considered in this paper.

This kind of syntax analysis requires the diagram to be represented by a structure that is similar to the object structure. The classes of some of the objects are not yet determined. An abstract superclass is used instead. Determining the actual concrete subclass is one of the syntax analysis tasks. The other task of syntax analysis consists of checking whether links of the object structure are correctly set. Syntax analysis has also to make sure that no cardinality constraint of any association is violated and that members, i.e., object attributes, are correctly declared in the class diagram. Directed graphs are a well suited representation of such object structures that allow for deducing concrete classes and checking those properties. Fig. 3 shows the graph for the tree of Fig. 2. Such a graph is called instance graph in order to distinguish it from a diagram’s object structure or object diagram as shown in Fig. 2. Please note that two-way links like the parent-child-link of the object diagram are represented by two opposite edges. This makes the distinction between different roles of a link easier. Moreover, it also allows for easy representation of one-way links that can be navigated in one direction only. However, the tree example does not make use of one-way associations resp. one-way links.

Instance graphs are defined formally in the following in order to allow for presenting the problem of syntax analysis and its solution in the next section. Before defining instance graphs, we define directed graphs. As all structures make use of names (e.g., class names, role names), let \( N \) be a fixed set of symbols that are used as names.

**Definition 1.** A graph \( G \) over \( N \) is \( G = (\vec{G}, G, s_G, t_G, n_G) \) where \( G \) is a finite set of nodes, \( \vec{G} \) a finite set of edges, \( s_G, t_G : G \rightarrow \vec{G} \) assign the source resp. target node to each edge, and \( n_G : G \cup \vec{G} \rightarrow N \) assigns a name to each node and edge.

The graph \( G \) in Fig. 3 consists of the node set \( \vec{G} = \{a, b, c, d\} \), the edge set \( G = \{r, s, x, y, p, q\} \), a node’s name is its type, i.e., \( \forall n \in G : n_G(n) = \text{Name} \), and edges have names, e.g., \( n_G(r) = \text{child} \). Source and target function represent how edges connect nodes. E.g., edge \( r \) starts at node \( a \) and ends at \( b \), i.e., \( s_G(r) = a \) and \( t_G(r) = b \).

Instance graphs are an extension of graphs where nodes may have member attributes, and there are pairs of opposite edges representing two-way links:

**Definition 2.** An instance graph \( I \) over \( N \) is \( I = (\vec{I}, I, s_I, t_I, n_I, m_I, o_I) \) where \((\vec{I}, I, s_I, t_I, n_I)\) is a graph over \( N \), \( m_I : \vec{I} \rightarrow 2^N \) assigns\(^1\) a set of names to each node, and \( o_I : I \rightarrow I \) is a partial function that assigns an opposite edge to some of the edges such that \( \forall a, b \in I : a = o_I(b) \Rightarrow b = o_I(a) \land s_I(a) = t_I(b) \land s_I(b) = t_I(a) \).

The graph aspect \( G \) of the instance graph \( I \) in Fig. 3 has been described above. Member attributes are represented by a function \( m_I \), e.g., \( m_I(a) = \{\text{name}\} \). Pairs of opposite edges are represented by a partial function \( o_I \), e.g., \( o_I(r) = s \) and \( o_I(s) = r \). The conditions on \( o_I \) make sure that edges \( e \) and \( o_I(e) \) are really opposite edges. \( o_I \) is actually a total function here as there are no one-way links. For an edge \( e \) that represents a one-way link, \( o_I(e) \) would be undefined, i.e., \( e \notin \text{dom}(e) \).

Finally, we have to define class diagrams formally, too. Class diagram are also graphs. Actually, a class diagram is like an instance graph with the extension of inheritance, the distinction of abstract and concrete classes, and cardinality constraints being attached to associations. In the following, class diagrams are represented by instance graph schemas:

**Definition 3.** An instance graph schema \( S \) over \( N \) is \( S = (\vec{S}, S, s_S, t_S, n_S, m_S, o_S, A_S, c_S, \subseteq) \) where \((\vec{S}, S, s_S, t_S, n_S, m_S, o_S)\) is an instance graph, \( A_S \subseteq \vec{S} \) is the set of abstract nodes, \( c_S : \vec{S} \rightarrow 2^{N_0} \) assigns a set of nonnegative integer numbers to each edge, and \( \subseteq \subseteq \vec{S} \times \vec{S} \) is a partial ordering on nodes describing subclassing.

Furthermore, node types have to be unique, i.e., \( \forall a, b \in \vec{S} : n_S(a) = n_S(b) \Rightarrow a = b \) and for each node, the roles of leaving associations, i.e., the names of leaving edges are unique, i.e., \( \forall a, b \in \vec{S} : s_S(a) = s_S(b) \land n_S(a) = n_S(b) \Rightarrow a = b \). Finally, attribute names have to be unique with respect to the type hierarchy, i.e., \( \forall a, b \in \vec{S} : a \leq b \land m_S(a) \cap m_S(b) \neq \emptyset \Rightarrow a = b \).

\(^1\)\(2^N\) is the set of all subsets of \( N \).
The instance graph schema $S$ for the class diagram in Fig. 1 consists of the node set

$$S = \{ \text{Node, Parent, Child, SingleNode, Root, InnerNode, Leaf} \}$$

and edge set $\hat{S} = \{ \text{child, parent} \}$. As role names are unique in this diagram, they are used as edge identifiers also.

The representation of the cardinality constraint $1..*$ is $c_S(\text{child}) = \{1, 2, 3, \ldots \}$ and the other $c_S(\text{parent}) = \{1 \}$. The abstract nodes are $A_2 = \{ \text{Node, Parent, Child} \}$. Subclassing is indicated by $\leq$ which is the reflexive transitive closure of Parent $\leq$ Node, InnerNode $\leq$ Parent, InnerNode $\leq$ Child etc.

### 3. Syntax Analysis

Syntax analysis has been informally introduced in the previous section. Syntax analysis consists of deducing concrete subclasses for classes in the instance graph, and checking whether the instance graph (after “adjusting” the classes) fits to the instance graph schema. Fitting means that it is possible to map the instance graph to the instance graph schema such that names and structure are preserved. This is similar to regular graph morphisms. However, as instance graph schemas contain subclassing, names and structure must be preserved “up to the subclassing ordering” which is called $\leq$-morphism in the following:

**Definition 4.** Let $A, B$ be two graphs over $\mathcal{N}$ and $\leq \subseteq B \times B$ a partial ordering on the nodes of $B$. A $\leq$-morphism $h : A \rightarrow B$ from $A$ to $B$ is a pair $(\hat{h}, \check{h})$ of two functions $\hat{h} : \hat{A} \rightarrow \hat{B}$ and $\check{h} : \check{A} \rightarrow \check{B}$ that map $A$-nodes to $B$-nodes resp. $A$-edges to $B$-edges with the following properties:

- Names of nodes are preserved
  $$\forall a \in A : \exists b \in B : h(a) \leq b \land n_A(a) = n_B(b)$$

- Names of edges are preserved
  $$\forall a \in A : n_B(\check{h}(a)) = n_A(a)$$

- Structure is preserved
  $$\forall a \in A : h(s_A(a)) \leq s_B(\check{h}(a))$$
  $$\forall a \in A : h(t_A(a)) \leq t_B(\check{h}(a))$$

Preservation of node names means that each $A$-node is mapped to a $B$-node (i.e., a class) that is a subclass of a $B$-node with the same class name as the mapped $A$-node. This property makes sure that classes of instance graph nodes are “adjusted” to subclasses only. Preservation of edge names means that edge images have the same names. And structure preservation means that $A$-edges are connected to nodes of classes which are also connected by corresponding edges in $B$ when considering the ordering $\leq$.

$\leq$-morphisms are a generalization of graph morphisms. However, instance graphs have to be mapped to instance graph schemas, i.e., graphs with additional properties that have to be accounted for. Member attributes have to fit, pairs of opposite edges of the instance graph have to correspond to pairs of opposite edges in the instance graph schema, and, finally, cardinality constraints have to be obeyed:

**Definition 5.** Let $I$ be an instance graph over $\mathcal{N}$ and $S$ an instance graph schema over $\mathcal{N}$ with subclass ordering $\leq$. $I$ corresponds to $S$, written $S \models I$, iff there is a $\leq$-morphism $h : I \rightarrow S$ with the following additional properties:

- Each instance has its declared attributes
  $$\forall n \in I : m_I(n) = \bigcup \{ ms_S(t) \mid t \in S \land h(n) \leq t \}$$

- Each link has its declared opposite link
  $$\forall a \in I : (a \in \text{dom} (o_A) \iff \check{h}(a) \in \text{dom} (o_S))$$

- Cardinality constraints are not violated
  $$\forall n \in I : \forall e \in S : h(n) \leq s_S(e) \Rightarrow [(I \in I \mid s_I(l) = n \land \check{h}(l) = e)] \in c_S(e)$$

- Each instance is mapped to a non-abstract class
  $$\forall n \in I : \hat{h}(n) \notin A_2$$

Property 5 means that the member attributes of each $I$-node is the same as the set of all member nodes of the corresponding $S$-node, i.e., the attributes that are declared for this very node or for any of its superclasses. Property 6 makes sure that only those $I$-edges have opposite edges that have opposite edges in the instance graph schema also, i.e., links are two-way links iff the corresponding associations can be navigated in both directions. And the mapping preserves the opposite relation between two edges because of property 7. Property 8 actually counts the number of links that are mapped to the same association and checks whether the resulting numbers are permitted by the cardinality constraints. Finally, property 9 makes sure that instance graph classes are adjusted to concrete classes only.

These definitions now allow a precise characterization of the syntax analysis problem. The task consists of two consecutive steps: first, concrete subclasses have to be deduced for each node of an instance graph $I$. The second step consists of checking whether the result fits to an instance graph schema $S$ that specifies the diagram language’s syntax. This task is solved by searching for an appropriate $\leq$-morphism $h : I \rightarrow S$ which assigns the right type to each node in the instance graph.

We show in the following that this problem can be mapped to a standard problem, namely to a constraint satisfaction problem (CSP). This mapping allows for efficient, standard constraint solving algorithms.

**Definition 6.** A constraint satisfaction problem (CSP) consists of a finite set of variables $V$ and a finite set of constraints $C$. Each variable $v \in V$ has a fixed domain $D(v)$. Each constraint $C \in C$ restricts the values
of some variables \(v_1, \ldots, v_n \in V\) by defining a relation \(C \subseteq D(v_1) \times \cdots \times D(v_n)\). A solution of a CSP is an assignment \(\omega : V \rightarrow \bigcup_{v \in V} D(v)\) with \(\forall v \in V : \omega(v) \in D(v)\) such that each constraint is satisfied, i.e., \(C(\omega(v_1), \ldots, \omega(v_n))\) for the appropriate variables for each constraint.

The network of variables of a CSP and the constraints on them can be visualized as a constraint graph (cf. Fig. 4).\(^2\) For simplicity, we say that a constraint restricts its variables instead of the values of the variables.

Constraint Satisfaction Problems have been a subject of research in Artificial Intelligence for many years. The problem of finding solutions is in general NP-hard. The simplest, but most expensive solution is using a depth-first search by setting variable values and testing all constraints. However, a variety of much more efficient algorithms has been proposed in the literature. A survey is presented in [6]. We will not discuss different algorithms in this paper, but use a rather simple one (constraint propagation) as it has proven well suited to the syntax analysis of CSP that is devised in the following. Application of constraint propagation is described afterwards.

Syntax analysis consists of searching for appropriate images of instance graph nodes and instance graph edges such that the properties in def. 4 and 5 are satisfied. This suggests the following transformation into a CSP.

Let \(I\) and \(\mathcal{S}\) be a fixed instance graph resp. a fixed instance graph schema over \(\mathcal{N}\). The variable set consists of all nodes \(\hat{I}\) of the instance graph and all of its edges (links) \(\hat{I}\), i.e., \(V = \hat{I} \cup \hat{I}\). The domains of the variables are the obvious candidates from the instance graph schema \(\mathcal{S}\). However, nodes have to be mapped to non-abstract classes because of property 9. The domain \(D(v)\) of a variable \(v \in V\) is

\[
D(v) = \begin{cases} \hat{S} \setminus \hat{A}_S & \text{if } v \in \hat{I} \\ \hat{S} & \text{if } v \in \hat{I} \end{cases}
\]

The syntax analysis problem consists of determining a value assignment \(\omega : V \rightarrow \bigcup_{v \in V} D(v)\) with \(\forall v \in V : \omega(v) \in D(v)\) such that \(\omega\) defines a \(\leq\)-morphism with the additional properties of def. 5. The properties of definitions 4 and 5 are easily translated in the following constraints:

- For each variable \(v \in \hat{I}\), an assigned value \(\alpha = \omega(v)\) satisfies constraint \(C_I^v\):
  \[C_I^v(\alpha) \iff \exists t \in \hat{S} : \alpha \leq t \land n_I(v) = n_S(t)\]

- For each variable \(v \in \hat{I}\), an assigned value \(\alpha = \omega(v)\) satisfies constraint \(C_I^v\):
  \[C_I^v(\alpha) \iff n_S(\alpha) = n_I(v)\]

- For each variable \(v_1 = \hat{I}\) and variable \(v_2 = s_I(v_1) \in \hat{I}\), the assigned values \(\alpha_1 = \omega(v_1)\) and \(\alpha_2 = \omega(v_2)\) satisfy constraint \(C_3\):
  \[C_3(\alpha_1, \alpha_2) \iff \alpha_2 \leq s_S(\alpha_1)\]

\(^2\)Constraint graphs are usually constraint hypergraphs as constraint graphs actually require binary constraints, i.e., constraints restricting the values of two variables each.

Please note that \(C_3\) does actually not depend on the specific variables \(v_1\) and \(v_2\).

- For each variable \(v_1 = \hat{I}\) and variable \(v_2 = t_I(v_1) \in \hat{I}\), the assigned values \(\alpha_1 = \omega(v_1)\) and \(\alpha_2 = \omega(v_2)\) satisfy constraint \(C_4\):
  \[C_4(\alpha_1, \alpha_2) \iff \alpha_2 \leq t_S(\alpha_1)\]

- For each variable \(v \in \hat{I}\), an assigned value \(\alpha = \omega(v)\) satisfies constraint \(C_I^v\):
  \[C_I^v(\alpha) \iff m_I(v) = \bigcup \{m_S(t) : t \in \hat{S} \land \alpha \leq t\}\]

- For each variable \(v \in \hat{I}\), the assigned value \(\alpha = \omega(v)\) satisfies constraint \(C_I^v\):
  \[C_I^v(\alpha) \iff \alpha \in \text{dom}(\alpha)\]

- For each variable \(v_1 \in \text{dom}(\alpha)\) and the variable \(v_2 = s_I(v_1) \in \hat{I}\), the assigned values \(\alpha_1 = \omega(v_1)\) and \(\alpha_2 = \omega(v_2)\) satisfy constraint \(C_3\):
  \[C_3(\alpha_1, \alpha_2) \iff \alpha_2 = s_S(\alpha_1)\]

- For each variable \(v \in \hat{I}\) and each variable \(v_i \in \hat{I}\) such that \(s_I(v_i) = v\), the assigned values \(\alpha = \omega(v)\) and \(\alpha_i = \omega(v_i)\) satisfy constraint \(C_S\):
  \[C_S(\alpha, \alpha_1, \ldots, \alpha_k) \iff \forall \epsilon \in \hat{S} : \alpha \leq s_S(\epsilon) \Rightarrow \{i \mid \alpha_i = \epsilon\} \in c_S(\epsilon)\]

Fig. 4 shows the constraint network for the instance graph in Fig. 3 with respect to the class diagram in Fig. 1. Circles indicate variables, i.e., instance graph nodes and
edges, rectangles are constraints as defined above. Lines between constraints and variables indicate which variables are restricted by which constraint. Integers indicate the ordering of restricted variables for constraints that restrict more than one variable.

Table 1 shows the predicates, i.e., relations of each of the constraints in Fig. 4 by enumerating all tuples satisfying the predicate. However, we omit those tuples that refer to abstract classes as abstract classes are not contained in the node variables' domains. The ordering of tuple components obeys the ordering of variables that are restricted by the corresponding constraint.

Depth-first search is the simplest algorithm for solving a CSP. Backtracking is necessary when reaching dead-ends or when several solutions exist. The latter issue will be considered later, but the former issue can be reduced by detecting dead-ends early and avoiding to try the same dead-end several times. This can be achieved by constraint propagation as preprocessing prior to backtracking search. Constraint propagation considers only a limited set of constraints, and reduces the domains of all restricted variables. This leads to the reduction of the domains of other variables, and so on, i.e., constraints propagate through the network. The simplest constraint propagation techniques is using node consistency and arc consistency [7]. Node consistency means removing all values from the domain of a variable that do not satisfy the unary constraints that restrict just this variable. Arc consistency considers every binary constraint and removes those values from the domains of the restricted variables that are not supported by the constraint, i.e., there is no corresponding value of the other variable such that the constraint is satisfied. This process, of course, has to be iterated such that domain reductions propagate through the network. Several algorithms have been proposed that achieve node and arc consistency [6]. The worst case behavior is quadratic in the size of the variables' domains and linear in the number of constraints.

Domains of variables in syntax analysis CSPs consist of sets of edges and nodes of the instance graph schema, i.e., essentially the class diagram. For unary and binary constraints, node and arc consistency, therefore, is quadratic in the size of the class diagram and linear in the number of constraints, i.e., the size of the instance graph. However, syntax analysis CSPs also have constraints that restrict more than two variables \( (C_3^v) \). Arc consistency can be generalized to hyper-arc consistency. Hyper-arc consistency is ensured for a constraint by removing those values from the domain of any restricted variable that is not supported by the constraint. This process is in general more expensive as combinations of values of all restricted variables have to be considered. However, when checking and propagating those constraints at the end, domains have been considerably reduced by the steps before. This is demonstrated by our example of trees, also.

Table 2 shows the initial domains of all variables and the reduced domains after ensuring node consistency according to the unary \( C_2^v \)-constraints and then arc consistency of the binary \( C_3^v \)-constraints. Finally, hyper-arc consistency of the \( C_4^v \)-constraints is ensured after domains of the edge variable have been reduced to an unique value, i.e., ensuring hyper-arc consistency is easy to ensure. Moreover, each domain has been reduced to a singleton. As each constraint is consistent, this is a solution to the syntax analysis CSP. Searching by backtracking is not necessary. Experiences with an implementation of the approach (cf. Section 4) reveal that search by backtracking for practically relevant languages has never been necessary. Only syntax analysis problems for particularly constructed languages required backtracking search.

CSPs generally do not have unique solutions. Syntax analysis may have several solutions also, i.e., the classes of certain instance nodes can be "adjusted" in more than one way. This is analogous to an ambiguous grammar which is usually considered an error of grammar design. Ending up with non-unique solutions of a syntax analysis CSP, therefore, is considered a specification error, too. Besides, having more than one possible solution also means that backtracking search is necessary. As mentioned above, we have not yet experienced a practically relevant language that makes backtracking necessary, i.e., non-unique solutions could not be experienced either.

4. DIAGEN

The presented syntax specification and analysis approach has been integrated into the DIAGEN tool. DIAGEN provides an environment for rapidly developing diagram editors. The previous version of DIAGEN supported grammar-based specifications and parsing of diagram languages only [9, 10]. The new version, which is discussed here, uses the presented approach as an alternative specification and analysis method.

DIAGEN is implemented in Java and consists of an editor framework and a program generator. DIAGEN makes use of the Eclipse Modeling Framework EMF [1] for specification and implementation of object structures and class diagrams. Diagram editors which have been developed using DIAGEN (such editors are called "DIAGEN editors" in the following) are structured as shown in Fig. 5. DIAGEN editors provide the following features:

DIAGEN editors always support free-hand editing, i.e., the editor user can arbitrarily create, delete, and modify diagram components (nodes and edges for trees) like with an off-the-shelf drawing tool. The editor uses the provided class diagram as described in the previous section for analyzing the instance graph after each editing operation performed by the user. Feedback is provided to the user where the "drawing" is not a correct diagram. The instance graph is obtained by mapping the diagram to a graph. This mapping process is implemented by the modeler and the reducer and is considered later.

The result of syntax analysis (object model) is an object structure as shown in Fig. 2, i.e., the result of instantiating classes and associations of the specifying class diagram after

---

3Abstract classes have not been prohibited in the definition of the constraints in order to provide easier predicates.

4Blank fields mean that the domain has not been modified.

5DiAGEN is free software and it is available from www.unibw.de/inf2/DiaGen.
Table 1: Constraint predicates for the constraint network in Fig. 4.

<table>
<thead>
<tr>
<th>Constraints</th>
<th>permitted tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1^7$, $C_3^7$, $C_5^7$, $C_6^7$</td>
<td>Root, InnerNode, Leaf, SingleNode</td>
</tr>
<tr>
<td>$C_2^8$, $C_3^8$, $C_4^8$, $C_6^8$</td>
<td>child</td>
</tr>
<tr>
<td>$C_2^8$, $C_3^8$, $C_5^8$, $C_6^8$</td>
<td>parent</td>
</tr>
<tr>
<td>$C_3^9$</td>
<td>(parent, InnerNode), (parent, Leaf), (child, Root), (child, Inner Node)</td>
</tr>
<tr>
<td>$C_4^9$</td>
<td>(parent, Root), (parent, InnerNode), (child, InnerNode), (child, Leaf)</td>
</tr>
<tr>
<td>$C_5^9$, $C_7^9$, $C_8^9$</td>
<td>Root, InnerNode, Leaf, SingleNode</td>
</tr>
<tr>
<td>$C_6^9$, $C_8^9$, $C_9^9$, $C_{10}^9$, $C_{11}^9$</td>
<td>child, parent</td>
</tr>
<tr>
<td>$C_7^9$</td>
<td>(child, parent), (parent, child)</td>
</tr>
<tr>
<td>$C_8^9$, $C_9^9$, $C_{10}^9$</td>
<td>(Root, child), (Leaf, parent)</td>
</tr>
<tr>
<td>$C_{10}^9$</td>
<td>(Root, child, child, child), (InnerNode, child, child, child), (InnerNode, child, parent, child), (InnerNode, child, child, parent), (InnerNode, parent, child, parent), (InnerNode, parent, child, parent), (InnerNode, parent, parent, child), (InnerNode, parent, parent, parent), (InnerNode, parent, parent, parent)</td>
</tr>
</tbody>
</table>

Table 2: Constraint propagation for the CSP in Fig. 4 and Table 1.

<table>
<thead>
<tr>
<th>Variable</th>
<th>domain after $C_3^2$</th>
<th>domain after $C_3^4$</th>
<th>domain after $C_8^9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Root, InnerNode, Leaf, SingleNode</td>
<td>Root, InnerNode</td>
<td>Root</td>
</tr>
<tr>
<td>b</td>
<td>Root, InnerNode, Leaf, SingleNode</td>
<td>InnerNode</td>
<td>InnerNode</td>
</tr>
<tr>
<td>c</td>
<td>Root, InnerNode, Leaf, SingleNode</td>
<td>InnerNode, Leaf</td>
<td>Leaf</td>
</tr>
<tr>
<td>d</td>
<td>Root, InnerNode, Leaf, SingleNode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>child, parent</td>
<td>child</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>child, parent</td>
<td>parent</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>child, parent</td>
<td>child</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>child, parent</td>
<td>parent</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>child, parent</td>
<td>child</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>child, parent</td>
<td>parent</td>
<td></td>
</tr>
</tbody>
</table>

The following paragraphs briefly survey the main steps in processing a diagram after creating or modifying in free-hand mode, i.e., the process chain consisting of drawing tool, modeler, reducer, and syntax analysis in Fig. 5. Structured editing and layout [9] as optional extensions are omitted here.

DIAGen editors are based on hypergraphs as internal diagram models. Hypergraphs are generalizations of directed graphs: they have a set of labeled hyperedges instead of edges. Each hyperedge has a fixed number of labeled tentacles which is determined by the hyperedge’s label. Tentacles connect the hyperedge with nodes visited by the hyperedge. A regular directed graph is a hypergraph where each hyperedge has two tentacles with labels source and target.

Diagram components (e.g., nodes and edges for trees) have attachment areas, i.e., the parts of the components that are allowed to connect to other components (e.g., start and end of an edge). The most general and yet simple formal description of such a component is a hyperedge which connects to the nodes which represent the attachment areas of the diagram components. These nodes and hyperedges first make up an unconnected hypergraph. The modeler connects nodes by additional edges if the corresponding attachment areas are related in a specified way, which is described in the specification. The result of this step is the hypergraph model of the diagram.

Hypergraph models tend to be quite large even for small diagrams. In order to allow for efficient syntax analysis, a reducer creates the actual, smaller data structure that is then syntactically analyzed. The previous DIAGen version created another hypergraph model that has then been parsed according to the diagram language’s hypergraph grammar [9]. The new version of DIAGen creates the instance graph from the hypergraph model. The reducer is specified by some transformations that identify those sub-hypergraphs of the hypergraph model which carry the information of the diagram and build the instance graph ac-
Figure 5: Architecture of a DiaGen editor using syntax definition with class diagrams.

cordingly. This step is similar to the lexical analysis step of traditional compilers. The reducer specification is omitted here, as it is quite similar to already published work [9].

The final step, syntax analysis, applies the results that have been described in the previous sections. Additionally to the described concepts, DiaGen uses the Kent OCL TOOLKIT [5] for specifying and evaluating constraints on object structures as described in Section 2. Moreover, syntax analysis creates an object model that can be used by other applications immediately. This is a major benefit with respect to the previous DiaGen version that employed attributed grammars. Creating an object model from the parser’s derivation structure requires the specification of attribute evaluation rules. This is not necessary for class diagram-based specifications. On the contrary, class diagrams as syntax specification are the specification of the object model, too.

5. CONCLUSIONS

This paper has shown that free-hand editors can make use of a meta-model as syntax definition. An efficient solution has been identified by transforming the syntax analysis problem into a constraint satisfaction problem. This solution has many advantages over the traditional grammar-based approach: First, users are accustomed to build class diagrams. Building a syntax definition by a meta-model, therefore, is easier for them than building a grammar. Experience shows that syntax analysis by constraint satisfaction solution is much more efficient than parsing with respect to a grammar. And finally, integration of meta-model-based editors into applications is easier than the integration of grammar-based editors. Whereas the latter requires translation of derivation structures into some abstract representation, this is not necessary for the former ones. Instead, edited data structures can be used by the application immediately.

Favoring free-hand editing over structured editing may be a matter of debate. Many users are satisfied by current editors, even if usability may be improved by free-hand editing or a combination of free-hand and structured editing. However, we are currently working on sketching-support for generated editors. Sketching, i.e., drawing diagrams with a stylus on a tablet or directly on the screen of a Tablet PC, does not allow for structured editing but requires free-hand editing and syntax analysis. Our intention is to combine meta-modeling with sketching. This combination has been made possible by the presented approach.

6. REFERENCES