Electronic Communications of the EASST
Volume 1 (2006)

Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006)

Generating Meta-Model-Based Freehand Editors

Mark Minas

13 pages

Guest Editors: Albert Zündorf, Daniel Varró
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Generating Meta-Model-Based Freehand Editors

Mark Minas[1]

[1]Mark.Minas@unibw.de, http://www.unibw.de/Mark.Minas/
Institut für Softwaretechnologie
Universität der Bundeswehr München, Germany

Abstract: Most visual languages as of today (e.g., UML) are specified using a model in a meta-model-based approach. Editors for such languages have supported structured editing as the only editing mode so far. Free-hand editing that leaves the user more freedom during editing was not supported by any editor or editor framework since parsing has not yet been considered for meta-model-based specifications. This paper describes the diagram editor generator framework DIAMETA that makes use of meta-model-based language specifications and supports free-hand as well as structured editing. For analyzing freely drawn diagrams, DIAMETA parses a graph representation of the diagram by solving a constraint satisfaction problem.

Keywords: Diagram editors, meta-modelling

1 Introduction

Each visual editor, i.e., tool for editing data structures visually, implements a certain visual language. Several approaches and tools have been proposed to specify visual languages and to generate editors from such specifications. These approaches and tools can be distinguished by the way (1) the diagram language is specified, and by the way (2) the user interacts with the editor and creates resp. edits diagrams. These distinguishing features are considered in the following:

1. 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 [CDP05] and *constraint multiset grammars* in PENGUINS [CM95] for free-hand editing, and *hypergraph grammars* in DIAGEN [Min02, Min04] for free-hand as well as structured editing. Grammars describe a language syntax by rules that are applied, starting at a certain starting symbol, 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 current graph-like languages, i.e., the majority of visual languages, at least in computer science, have a model as (abstract) syntax specification. Models are essentially class diagrams of the data structures that are visualized by diagrams. This approach is generally called *meta-model-based* since the syntax of models is specified by models, too. Some examples for meta-model-based approaches are AToM^3 [LVA04], Pounamu [ZGH04], and MetaEdit+ [Met05]. There are several reasons for the success

of this approach. One of them is the training of users in specifying data structure with class diagrams. On the contrary, writing grammars appears to be much more complicated. Moreover, the visual modeling languages of the Unified Modelling Language (UML) are specified by models, too. Extending such visual languages then requires to use and extend their models instead of writing grammars.

2. When considering user interaction and the way how the user can create and edit diagrams, *structured editing* is usually 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 from a language-specific set 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 make the editing process even easier.

There are many examples of grammar-based tools supporting structured editing and free-hand editing. However, all of the meta-model-based approaches offer structured editing only. We are not aware of any tool supporting free-hand editing although that editing mode would offer more freedom to the user. A recent paper has shown that analyzing the correctness of a diagram and determining its meaning based on a meta-model-based approach can be efficiently solved by transforming this task into a constraint satisfaction problem [Min06]. Based on this approach, the tool DIAMETA is described in the following. Diagram languages must be specified by models, and DIAMETA generates visual editors offering structured editing as well as free-hand editing from such specifications. Generated editors, therefore, allow for easy free-hand editing and, at the same time, easy meta-model-based language specifications.

The next section describes the syntax specification with models that are class diagrams of the edited object structure. Moreover, the basic concepts of syntax analysis as described in [Min06] are outlined. Section 3 introduces the common editor architecture of each editor built using DIAMETA and the diagram analysis when editing a diagram in free-hand mode based on the concepts presented in section 2. Section 4 presents details of the DIAMETA environment, in particular on specification and code generation. Section 5 concludes the paper.

2 Syntax Specification and Analysis Based on 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 visually creating and modifying such object structures. Hence, class diagrams together with additional constraints are the specification of the corresponding diagram language. However, we will ignore additional constraints in the following in order to simplify discussions.

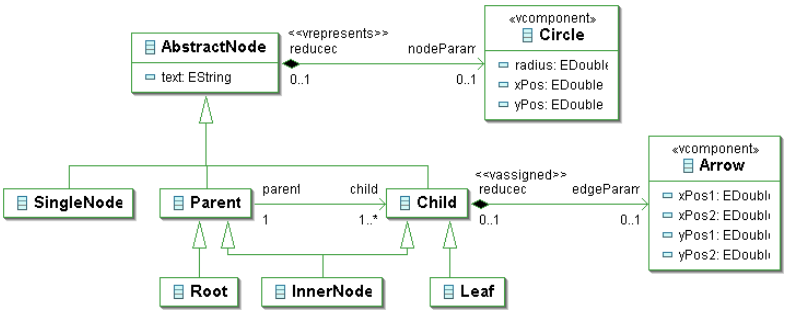

Figure 1: Model (i.e., class diagram) of trees

Class diagrams specify object structures, but they do not specify a diagram language 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. DIAMETA requires that a diagram is easily and uniquely mapped to a graph that resembles its object structure.[1] When drawing a diagram in free-hand mode, its representing graph is created by the editor.

We use simple trees as an example in the paper. The class diagram in Fig. 1 contains class *AbstractNode* as the abstract base class of a tree's nodes. Each node has a member attribute *text*[2]. Concrete classes are *Root*, *InnerNode*, *Leaf*, and *SingleNode*. Abstract superclasses *Parent* and *Child* represent nodes that act as parents resp. children. Please note that *InnerNode* is a 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 must have at least one child, and a child must have exactly one parent. The classes *Circle* and *Arrow* represent aspects of the concrete syntax aspect of the visual components as described in section 3.2.

The class diagram does not completely specify trees. It does not prevent object structures from being circular, and data structures may be disconnected. The first problem could be solved 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 represented tree for syntactic correctness means checking whether the object structure can be created by instantiation as described by the class diagram.[3] That is obviously true here, i.e., the object structure and the represented diagram are

[1] This process of mapping a diagram to an object structure is performed by the *reducer* as described in section 3.4. Since the reducer is rather powerful, DIAMETA is not restricted to graph-like diagrams, only. For instance, hierarchical diagrams like statecharts are supported as well as Nassi-Shneiderman diagrams.

[2] Fig. 1 is a screenshot of the EMF model as used in DIAMETA based on the Eclipse Modeling Framework (EMF) [EMF06]. EMF type `EString` corresponds to the Java type `String`.

[3] The classes *Circle* and *Arrow* are ignored until section 3.2.

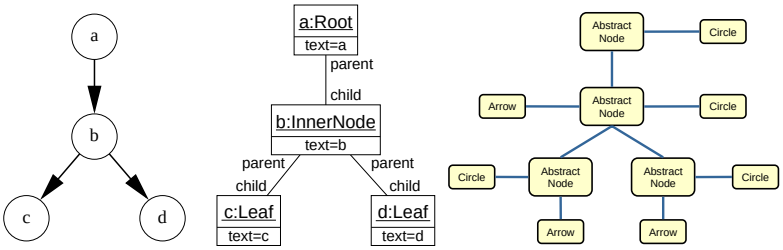

Figure 2: Sample tree, its object diagram with respect to Fig. 1, and its instance graph.

syntactically correct. However, root node, inner node and leaves of a tree cannot be distinguished visually. Actually, each leaf node may be turned into an inner node by adding an outgoing edge to another node. 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. In the example of Fig. 2, an editor can deduce from context information that *a* is a root node, *b* an inner node, and *c* as well as *d* are leaves. The editor has to perform this task by first binding the common type of all possible internal representations, i.e., class *AbstractNode* in the example, to each visual component. The obtained data structure (called *instance graph* in the following) is similar to object diagrams with the exception of the not yet determined concrete component types.

Correctness of the instance graph and, hence, of the diagram is checked by deducing the concrete types and examining whether the resulting object structure fits to the class diagram. In the predecessor paper [Min06], this problem is expressed as the problem of finding a special kind of graph morphism from the instance graph to the graph schema that represents the class diagram. Searching for the concrete types of the instance graph nodes is actually a *constraint satisfaction problem* (CSP). Preprocessing of this CSP requires linear time in the size of the analyzed diagram. Experiments had shown that backtracking was never required after preprocessing the CSP, i.e., syntax analysis is efficient when using meta-model-based syntax specifications.

3 DiaMeta editors

DiaMeta provides an environment for rapidly developing diagram editors based on meta-modelling. Diagram editors developed using DiaMeta (such editors are called "DiaMeta editors" in the following) always support free-hand editing. Each DiaMeta editor is based on the same editor architecture and contains code that is specific for this editor and its diagram language. The editor architecture is described in the following whereas section 4 outlines how DiaMeta's specification and code generation tool, the DiaMeta Designer, is used to generate the specific code of an editor.

3.1 DiaMeta editor architecture

Since DiaMeta is actually an extension of the diagram editor generator DiaGen [Min02, Min04], DiaMeta editors have a structure similar to that of DiaGen editors. Fig. 3 shows

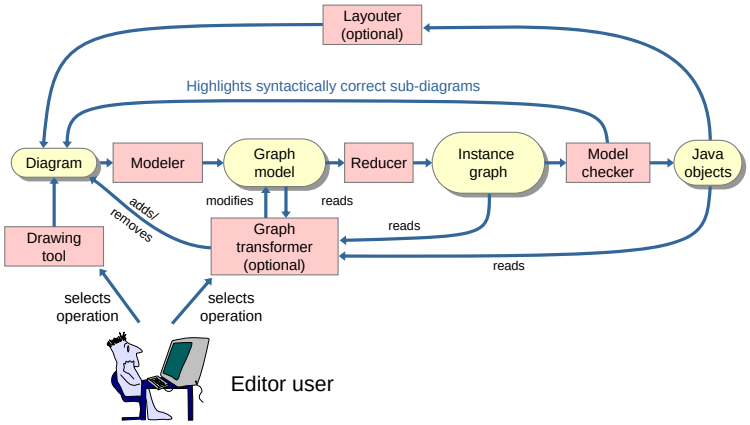

Figure 3: Architecture of a diagram editor based on DIAMETA.

the structure which is common to all DIAMETA editors and which is described in the following paragraphs. Ovals are data structures, and rectangles represent functional components. Flow of information is represented by arrows. If not labeled, information flow means reading resp. creating the corresponding data structures. The structure of DIAMETA editors is very similar to DIAGEN editors; they most prominently differ in the method of abstract syntax specification and, hence, syntax analysis. Moreover, DIAGEN requires a specification of attribute evaluation for creating abstract diagram representations. Since class diagrams as abstract syntax specification are also a specification of abstract diagram representations, DIAMETA does not require such an additional specification.

The editor supports free-hand editing by means of the included drawing tool which is part of the editor framework, but which has been adjusted by the DIAMETA DESIGNER. With this drawing tool, the editor user can create, arrange and modify diagram components which are specific to the diagram language. Editor specific program code which has been generated by the DIAMETA DESIGNER from the language specification is responsible for the visual representation of these language specific components. The drawing tool creates the data structure of the diagram as a set of diagram components together with their attributes (position, size, etc.).

The sequence of processing steps necessary for free-hand editing starts with the *modeler* and ends with *model checker* (cf. Fig. 3): The modeler first transforms the diagram into an internal model, the *graph model*. The *reducer* then creates the diagram's *instance graph* that is analyzed by the *model checker* (cf. section 2). This last processing step identifies the maximal subdiagram which is (syntactically) correct and provides visual feedback to the user by drawing those diagram components with a certain color; errors are indicated by missing colors. However, the model checker not only checks the diagram's abstract syntax, but also creates the object structure of the diagram's syntactically correct subdiagram.

The results of this step are not always uniquely defined, depending on the specification. The model checker cannot always uniquely deduce the concrete object types. However, this is considered a specification error. Moreover, there is not always a unique maximal syntactically correct subdiagram. DIAMETA then selects an arbitrary one. This behavior is sufficient in most cases

since wrong parts of the diagram are emphasized anyway.

The *layouter* modifies attributes of diagram components and thus the diagram layout based on the (syntactically correct subdiagram's) object structure. The layouter is necessary for realizing syntax-directed editing: Structured editing operations modify the graph model by means of the *graph transformer* and add or remove components to resp. from the diagram. The visual representation of the diagram and its layout is then computed by the layouter. However, layouters and structured editing are not considered in this paper.

The processing steps necessary for free-hand editing are described in more detail in the following.

3.2 Diagram components

Each diagram consists of a finite set of diagram components, each of which is determined by its attributes. For the diagram language of trees, there are nodes and arrows. Each node is a circle whose position is defined by its *xPos* and *yPos* coordinates, its size by a *radius* attribute, and its inscribed text by a *text* attribute. Each edge is an arrow whose position is defined by its two end points, i.e., by two coordinate pairs *xPos1* and *yPos1* resp. *xPos2* and *yPos2*. All of these attributes are necessary for completely determining a diagram component, e.g., when storing it to a file or retrieving it again. However, only some of these attributes are essential for a diagram's abstract syntax, too. In the tree example, only the inscribed text of a node is part of the abstract syntax. Position and size attributes are solely member of the concrete syntax. This fact is modelled in the tree model in Fig. 1, too: Attribute *text* is a member of *AbstractNode*; the other attributes do not belong to any class of the abstract syntax. However, they are attributes of the two additional classes *Circle* resp. *Arrow* which describe the diagram components and, therefore, belong to the concrete syntax, indicated by the stereotype $\langle\langle vcomponent \rangle\rangle$.[4] These concrete syntax classes belong to the diagram language's model for two reasons:

1. In order to specify all attributes in a single model, not only attributes from the abstract syntax are modelled in the diagram language model, but also the attributes from the concrete syntax.

2. The object structure as result of model checking is an instance of the diagram language model in terms of its class diagram. In order to contain all the information about the diagram, its concrete syntax attributes have to be represented, too. This is necessary for computing the layout based on the object structure, but also to store an object structure and to be able to retrieve the complete diagram again.

Note that a node's attributes are spread over two classes: *Circle* and *AbstractNode*. The connection between both classes is indicated by an association annotated with stereotype $\langle\langle vrepresents \rangle\rangle$. The code generator of the DIAMETA DESIGNER, when generating code for visual components, uses attributes of classes annotated with $\langle\langle vcomponent \rangle\rangle$ stereotypes and those which are connected by associations annotated with $\langle\langle vrepresents \rangle\rangle$ stereotypes to store a visual component's attributes.

[4] DIAMETA uses the three stereotypes $\langle\langle vcomponent \rangle\rangle$, $\langle\langle vrepresents \rangle\rangle$, and $\langle\langle vassigned \rangle\rangle$ to annotate class diagrams and to control DIAMETA's code generator. The meaning of these stereotypes are outlined in the following.

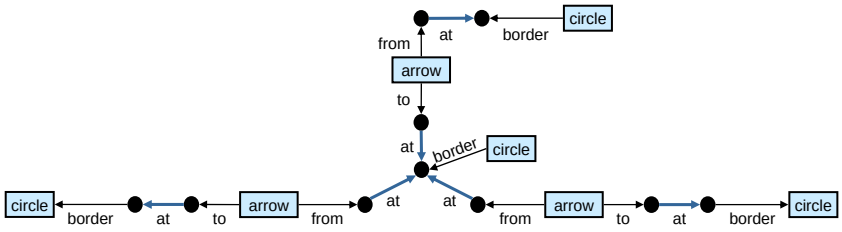

Figure 4: Graph model of the tree in Fig. 2.

Note also that there is an association between classes *Arrow* and *Child* that is not annotated with ⟨⟨*vrepresents*⟩⟩. *Arrow* instances do not have a direct representative in the abstract syntax like *Circle* instances that are represented by instances of *AbstractNode* subclasses. An *Arrow* instance is best assigned to a *Child* object as it has an incoming arrow. Since *Arrow* instances do not have a `text` attribute, stereotype ⟨⟨*vrepresents*⟩⟩ does not make sense for the association between classes *Arrow* and *Child* either. Instead, stereotype ⟨⟨*vassigned*⟩⟩ is used as annotation for this kind of associations.

Diagrams are checked for their correctness in terms of their instance graphs. The main difference between instance graph and object diagram of a diagram are the not yet determined concrete classes of the object diagram. However, instance graphs are the result of a translation process from the diagram's concrete syntax, i.e., mainly the arrangement of is diagram components. This process is described in the following, and it requires an intermediate, uniform representation of the analyzed diagram, its *graph model*.

3.3 Graph model

Arrangements of diagram components can always be described by spatial relationships between them. For that purpose, each diagram component typically has several distinct *attachment areas* at which it can be connected to other diagram components. An arrow representing an edge of a tree, e.g., has its two end points as attachment areas. Connections can be established by spatially related (e.g., overlapping) attachment areas as with trees where an arrow has to end at the border of a node's circle in order to be connected to the node.

DIAMETA uses directed graphs to describe a diagram as a set of diagram components and the relationships between attachment areas of "connected" components. Each diagram component is modeled by a node (called *component node*) whose type is determined by the kind of represented diagram component. Attachment areas are also modeled by nodes (called *attachment nodes*). Edges (called *attachment edges*) connect component nodes with attachment nodes for all attachment areas that belong to a component. Edge labels are used to distinguish different attachment areas. Relationships between attachment areas are modeled by edges (called *relationship edges*) connecting the corresponding attachment nodes. Relationship edges carry the kind of relationship as edge type.

Fig. 4 shows the graph model of the tree in Fig. 2. Attachment nodes are depicted by black dots, component nodes by rectangles. Thin arrows show attachment edges, thick arrows relation-

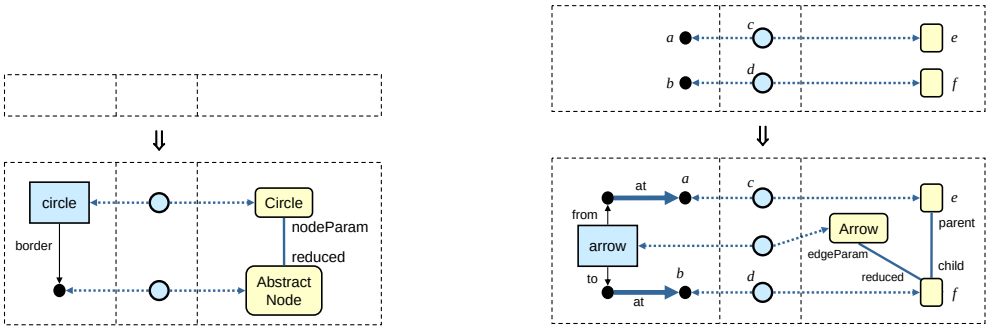

Figure 5: Reducer rules for trees.

ship edges. The only relationship type *at* indicates that the end point of the corresponding arrow is at the border of the connected circle.

3.4 Instance graph

The *reducer* is responsible for translating the graph model to the corresponding instance graph. The translation process has to be specified in the DIAMETA DESIGNER in terms of triple graph grammar rules [Sch94], called *reducer rules* here. The rules specify the simultaneous construction of the graph model as well as the instance graph. Additionally, it builds up a third graph which contains the information on correspondence of nodes of the graph model to nodes of the instance graph. Fig. 5 shows the two rules specifying the reducer for the diagram language of trees. Reducer rules operate on triple graphs: the graph model, the correspondence graph, and the instance graph, from left to right.

The left rule indicates that whenever a circle component node together with its attachment node and their attachment edge is added to the graph model, corresponding nodes are added to the instance graph. The added nodes represent an *AbstractNode* instance for the attachment node of the tree node, and a *Circle* instance for its component node. The latter models the set of those attributes of a tree node that do not belong to the abstract syntax (cf. section 3.2). The added edge represents the link as an instance of the association between classes *AbstractNode* and *Circle* in Fig. 1. The right rule adds an *Arrow* node representing the arrow attributes to the instance graph if an arrow component is added to the graph model. Right-hand side nodes e and f are the *AbstractNode* nodes that have been added by the other reducer rule already. Moreover, two edges are added to the instance graph that correspond to the two associations of the *Child* in Fig. 1.

3.5 Model checking

Finally, the instance graph is checked against the diagram language's model, its class diagram, as described in [Min06] and briefly outlined in section 2. By solving a constraint satisfaction problem, the model checker tries to identify a maximal subgraph of the instance graph that corresponds to the class diagram. This subgraph is used to instantiate the class diagram; the obtained

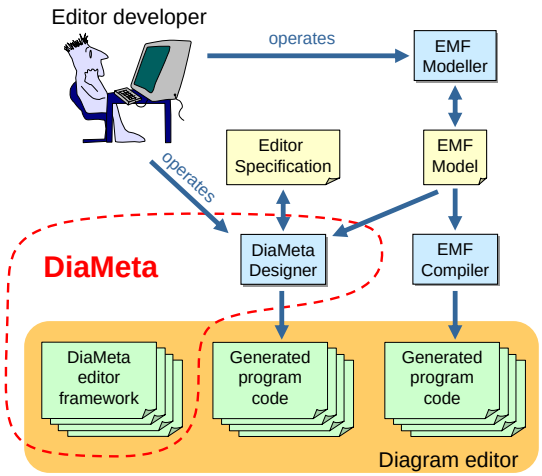

Figure 6: Generating diagram editors with DIAMETA.

structure of Java objects is an abstract representation of the diagram that can be used when integrating the editor in a larger environment. The subgraph, moreover, corresponds to a subgraph of the graph model and, hence, a subset of diagram components that form a syntactically correct subdiagram. Based on this information, feedback to the user is provided as described in section 3.1.

4 DIAMETA Environment

This section completes the description of DIAMETA and outlines its environment supporting specification and code generation of diagram editors that are tailored to specific diagram languages. The DIAMETA environment shown in Fig. 6 consists of an editor framework and the DIAMETA DESIGNER. The framework is an extension of the DIAGEN framework and, as a collection of Java classes, provides the generic editor functionality which is necessary for editing and analyzing diagrams. In order to create an editor for a specific diagram language, the editor developer has to provide two specifications: First, the abstract syntax of the diagram language in terms of its model, and second, the visual appearance of diagram components, the concrete diagram language syntax, the reducer rules and the interaction specification.

DIAMETA uses the *Eclipse Modelling Framework* EMF [EMF06] for specifying language models and generating their implementations. A language's class diagram is specified as an EMF model that the editor developer creates by using the *EMF modeller*. Several tools are available as EMF modeller, e.g., the built-in EMF model editor in the EMF plugin for Eclipse, or EclipseUML by Omondo [Ecl05]. The *EMF compiler*, being part of the EMF plugin for Eclipse, is used to create Java code that implements the model. Fig. 1 shows the tree class diagram as an EMF model. The EMF compiler creates Java classes (resp. interfaces) for the specified classes.

The editor developer uses the DIAMETA DESIGNER for specifying the concrete syntax and the visual appearance of diagram components, e.g., that tree nodes are drawn as circles with

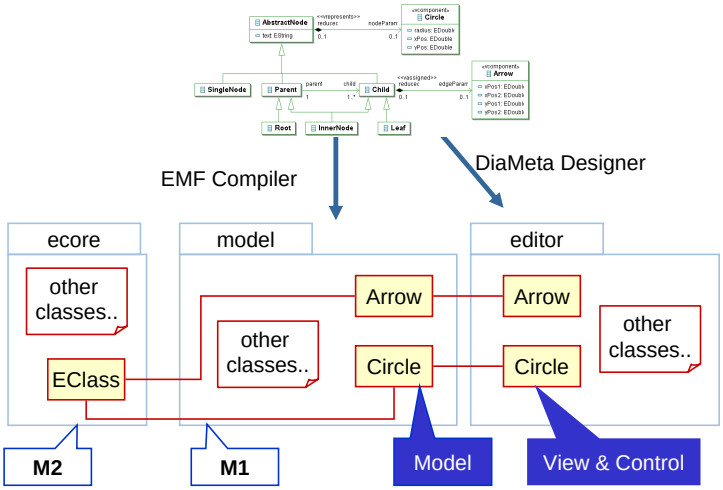

Figure 7: Generated Java classes from the specification.

inscribed name and tree edges as arrows. The DIAMETA DESIGNER generates Java code from this specification. This code, together with Java code created by the EMF compiler and the editor framework, implement an editor for the specified diagram language.

Fig. 7 shows Java classes and packages that are taken from the framework resp. that are generated by the EMF compiler and the DIAMETA DESIGNER for our example of tree diagrams. The EMF compiler creates the package *model*[5] that contains all classes corresponding to the language's EMF model. The DIAMETA DESIGNER generates the package *editor* (or an other chosen name) together with the classes that are responsible for visualizing diagram components, interacting with them in the editor, and language specific classes for diagram analysis.

Note that two classes *Circle* resp. *Arrow* are created – one in package *model* and one in package *editor* each. Together, they establish a *Model-View-Controller* pattern of the diagram components and the diagram: The classes in package *model* created by the EMF compiler represent the model aspect whereas the classes in *editor* created by the DIAMETA DESIGNER represent the view and controller aspects.

The classes in package *model* implement the EMF model and, hence, the abstract syntax of the specified diagram language. However, the EMF compiler does not only generate these classes, but also code that sets up a reflective model representation at start-up time as shown in Fig. 8. The reflective model representation represents the EMF model using *Ecore*, EMF's counterpart of the OMG MOF [Obj06]. This Ecore model allows to inspect the EMF model, i.e., the abstract diagram at editor runtime. In terms of the meta-modelling hierarchy, the abstract syntax of a specific diagram is on the M_0 level. Its EMF model, i.e., the class diagram, is on the M_1 level. Ecore is the model of all EMF models and, hence, on the M_2 level. These Ecore classes of the M_2 level are instantiated at start-up time such that these instances provide a runtime data structure representing the specified EMF model. The model checker makes use of this runtime

[5] The EMF compiler actually creates several packages. To avoid cluttering of the figure, only a single package is shown here.

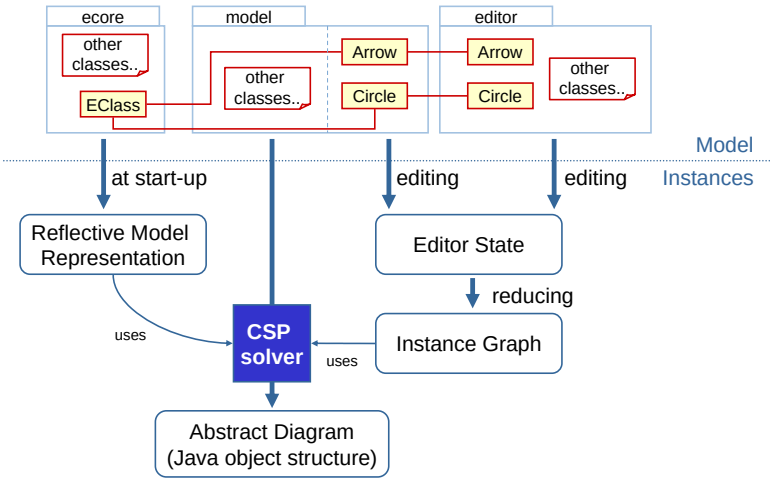

Figure 8: Using the reflective model representation for checking the diagram's syntax by constraint satisfaction.

data structure in order to inspect the EMF model and to find all possible concrete classes for each node of the instance graph and all possible associations for each edge in the instance graph.

5 Conclusions

The paper has described DIAMETA, a tool for generating visual editors that support free-hand and – at the same time – structured editing. However, structured editing has not been considered in this paper. The new contribution of DIAMETA and this paper consists of the combination of free-hand editing and a diagram language specification based on a meta-modelling approach. Earlier to this paper, meta-model-based editors had been restricted to structured editing.

The paper has described the architecture of visual editors generated by DIAMETA, and diagram analysis that checks the correctness of freely drawn diagrams and translates them – if they are correct – into some object structure. However, there are still unsolved questions. The predecessor of DIAMETA, DIAGEN, that used a grammar-based syntax specification, could easily identify maximal subdiagrams that are syntactically correct; DIAMETA with its model-based syntax specification and its syntax analysis based on a constraint satisfaction problem does not yet provide as satisfying results as DIAGEN.

Moreover, DIAMETA does not yet support additional constraints on the object structures. Such constraints are required if the diagram language is not fully specified by a class diagram. It is planned to allow for constraint specification using OMG's OCL and to add an OCL interpreter to DIAMETA.

The current DIAMETA implementation makes use of EMF for modelling diagram languages and for providing an implementation. The language of EMF models is specified by an EMF model, too. An apparent application of DIAMETA, hence, is generating an editor for EMF models; only the specification by the DIAMETA DESIGNER is yet missing. Such an editor would

automatically create Ecore instances when creating EMF models. The EMF compiler could be used without any further efforts for immediately generating code from such models.

Since EMF is a rather restricted meta-modelling framework, current work investigates OMG's MOF 2.0 as an alternative and the MOFLON plugin [Ame04] for the Fujaba tool [NZ00]. Using MOF instead of EMF has the further benefit that the visual languages of the UML are already specified by a MOF model. Hence, generating editors for those languages will become easier. Essentially, only the concrete syntax and the reducer rules have to be specified.

Bibliography

[Ame04] C. Amelunxen. A MOF 2.0 Editor as Plugin for FUJABA. In Giese et al. (eds.), *Proc. 2nd International Fujaba Days*. Volume tr-ri-04-253, pp. 43–48. 2004.

[CDP05] G. Costagliola, V. Deufemia, G. Polese. Towards Syntax-Aware Editors for Visual Languages. *Electronic Notes in Theoretical Computer Science* 127(4):107–125, Apr. 2005. Proc. Workshop on Visual Languages and Formal Methods (VLFM 2004).

[CM95] S. S. Chok, K. Marriott. Automatic Construction of User Interfaces from Constraint Multiset Grammars. In *Proc. 1995 IEEE Symp. on Visual Languages, Darmstadt, Germany*. Pp. 242–249. IEEE Computer Society Press, Sept. 1995.

[Ecl05] EclipseUML on the Omondo web site. http://www.omondo.com/, 2005.

[EMF06] EMF, Eclipse Modeling Framework web page. http://www.eclipse.org/emf/, 2006.

[LVA04] J. de Lara, H. Vangheluwe, M. Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modelling* 3(3):194–209, Aug. 2004.

[Met05] MetaEdit+ documentation on the MetaCase web site. http://www.metacase.com/, 2005.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[Min04] M. Minas. VisualDiaGen – A Tool for Visually Specifying and Generating Visual Editors. In Pfaltz et al. (eds.), *Applications of Graph Transformation with Industrial Relevance, Proc. 2nd Intl. Workshop AGTIVE'03, Charlottesville, USA, 2003, Revised and Invited Papers*. Lecture Notes in Computer Science 3062, pp. 398–412. Springer-Verlag, 2004.

[Min06] M. Minas. Syntax Analysis for Diagram Editors: A Constraint Satisfaction Problem. In Celentano and Mussio (eds.), *Proc. of the Working Conference on Advanced Visual Interfaces (AVI'2006), May 23-26, 2006, Venice, Italy*. Pp. 167–170. ACM Press, 2006.

[NZ00] J. Niere, A. Zündorf. Using Fujaba for the Development of Production Control Systems. In Nagl and Schürr (eds.), *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE*'99), Selected Papers*. Lecture Notes in Computer Science 1779, pp. 181–191. Springer, Mar. 2000.

[Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification. Version 2.0 edition, Jan. 2006. Document - formal/06-01-01.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Tinhofer (ed.), *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*. Lecture Notes in Computer Science 903, pp. 151–163. Springer Verlag, Heidelberg, 1994.

[ZGH04] N. Zhu, J. Grundy, J. Hosking. Pounamu: A Meta-Tool for Multi-View Visual Language Environment Construction. In *Proc. 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VL/HCC'04)*. Pp. 254–256. 2004.