GEM: A Generic Visualization and Editing Facility for Heterogeneous Metadata

Jürgen Göres, Thomas Jörg, Boris Stumm, Stefan Dessloch
Heterogeneous Information Systems Group, University of Kaiserslautern
{goeres|joerg|stumm|dessloch}@informatik.uni-kl.de

Abstract: Many model management tasks, e.g., schema matching or merging, require the manual handling of metadata. Given the diversity of metadata, its many different representations and modes of manipulation, meta-model- and task-specific editors usually have to be created from scratch with a considerable investment in time and effort. To ease the creation of custom-tailored editing facilities, we present GEM, a generic editor capable of visualizing and editing arbitrary metadata in an integrated manner. GEM provides a stylesheet language based on graph transformations to customize both, the mode of visualization and the available manipulation operations.

1 The importance of metadata

The vision of *generic model management* spurred by the works of Bernstein et al. [BHP00, MRB03] aims at reducing the effort to create metadata-intensive applications by defining generic operators that work on entire models and providing a model management system that implements these operators. Metadata-intensive applications can then be built on these systems like data-intensive applications are built on database management systems today. Examples of such applications include the broad area of information integration or the development of complex software systems.

In our research group, we work on novel approaches to create and maintain information integration systems. Creating an integration system subsumes numerous tasks, which all require the handling of metadata artifacts: Integrated schemas are designed from scratch or created by merging the source schemas. Semantic correspondences between the schemas have to be identified and be made explicit by schema matching. Based on these correspondences or "matches", mappings that perform the required data transformations have to be developed, e.g., by configuring wrappers of a federated DBMS and specifying view definitions over the wrapped sources, or by creating ETL scripts for replication-based integration. Existing integration systems require intensive maintenance operations: Changes to system components require the modification of matches and mappings.

30+ years of research have resulted in numerous approaches to automate some of these tasks, like automatic schema matching and merging techniques. However, for the foreseeable future, these approaches can at best be used in a semi-automatic fashion, therefore requiring human expertise to review, correct, and amend their results. Other tasks, like the design of schemas and software artifacts, are intrinsically manual. Human integration experts and software engineers therefore have to be provided with suitable interfaces to manipulate the many different kinds of metadata required for these tasks: Database schemas

are often designed using conceptual metamodels like one of the many E/R variants, and are only later mapped to physical schemas, represented by a data definition language of the respective data model like SQL DDL or XSD. For the design of the structure and the dynamic aspects of software components, diagrams of the UML family are used. Conceptualizations of application domains as ontologies are often represented by RDF or OWL documents.

Problem statement Metadata can truly be said to be omnipresent in many disciplines of computer science and manual metadata manipulation is often a necessity. Due to the complexity of metadata artifacts and the diversity of metadata representations this is by no means a trivial task. Many types of metadata have a native textual representation, but graphical representations are generally considered easier to handle by humans. Already a 1:1 mapping from textual to a graphical representation can often improve the understanding and manageability of models. But given the enormous volume and complexity of real-world metadata, different degrees of abstraction are often an absolute necessity to make handling of large models by human developers feasible.

The development of editors for such graphical representations means a significant investment in time and effort, and incurs the well-known risks of any large software project. This is acceptable when creating a commercial tool for an established and well-defined metamodel, e.g., a UML CASE tool. These tools are, however, limited to their native metamodel and only support the editing functionality anticipated by their designers. Extending the capabilities requires changing the code of the editor – if available as open source – or is simply not possible for closed-source tools.

This is especially a problem in the context of information integration and metadata or model management in general, where both the metadata and the operations on this metadata are often too diverse to be handled by a single tool. As a consequence, different tools have to be used in combination. Consequently, not only do developers have to to familiarize themselves with each of these tools, but are also impeded by the lack of interoperability, caused by the many proprietary formats to represent the metadata artifacts that are created and manipulated. In addition, often metadata from different metamodels has to be handled in an integrated fashion, e.g., for schema matching. Simply using different tools in parallel cannot help here. As an alternative, integrated tool suites avoid the metadata exchange problem (at least for those aspects covered by the suite), but will in general not be able to provide the optimal solution for each individual task. In addition, tools and tool suites are often tightly coupled with other products of their respective vendor, limiting the potential reuse of the artifacts created by such a tool.

While a manual combination of existing tools and tool suites may sometimes offer a cumbersome yet working way to perform the desired metadata management tasks in production environments, researchers often have very specific editing requirements for which no tools exist: They have to handle proprietary, complex models, where both the metamodels and the kinds of operations on the models are ever evolving as research progresses. Unable to commit the resources to create their own editor from scratch and adapt it to the changes, very often they have to go without a suitable editing tool.

Contribution In this paper, we present *GEM*, a generic visualization and editing facility for arbitrary graph-based data. GEM allows to rapidly develop metamodel- and task-specific editors without the need to write a single line of program code. For this purpose, GEM provides a declarative *graph stylesheet language* to easily adapt its visualization and editing functionality to any such metamodel and editing task.

Graph stylesheets are based on the concept of *graph transformations* and define a set of *visualization rules* that translate the application data provided in a graph representation to a *visualization graph*. The nodes and edges of the visualization graph correspond to an extensible set of user interface elements (so-called *widgets*). The visualization graph is directly interpreted and displayed by the editor.

For editing, a graph stylesheet contains a set of *edit operations*, which define how manipulations of the graphical elements are to be propagated to the application model. GEM stores application and visualization graphs in a relational database system; graph transformations are translated to SQL DML statements. This implementation approach has shown to have adequate performance even for large models, and outperforms any of the existing graph transformation systems we evaluated.

The remainder of the paper is structured as follows: Section 2 gives an overview of the general concepts of the editor. Section 3 introduces the graph-based representation of arbitrary (meta-)data. Section 4 gives an introduction on graph transformations in general and the specific transformation formalism underlying the GEM stylesheets, which are presented in detail in Section 5. Section 6 demonstrates GEM's practical usability in a realistic scenario. Section 7 highlights interesting aspects of the prototype's implementation and gives performance measurements. Section 8 gives an overview of related work, and Section 9 closes with a summary and an outlook on future work and usage scenarios.

2 Overview

In this section, we give an introduction into GEM. First, we describe the editor from a user perspective, to show its applicability in real world scenarios. Then, we give an overview from a stylesheet developer perspective, illustrating the steps needed to adapt the editor to a certain metamodel. Finally, we give a high-level architectural introduction into the editor.

The primary goal in the development of GEM was to provide a customizable editor for arbitrary metadata. However, the editor is not limited to the role of metadata editing: it works on graphs representing any kind of data. To emphasize our focus on metadata editing and to use a terminology that is in line with our application scenarios, we will refer to the data being displayed and edited as an *application model*, or *application graph*. The prototype of GEM will be made publicly available on the GEM project site[1]. Currently supported DBMSs are Apache Derby, H2, IBM DB2, and PostgreSQL.

[1] http://wwwlgis.informatik.uni-kl.de/cms/index.php?id=GEM

2.1 GEM from a user perspective

The basic process of editing an application model consists of several steps. First, the user imports an application model file into the editor database, comparable with the "open" action in conventional editors. Several different models can be imported and edited simultaneously. Before the model can be edited, the user has to apply a graph stylesheet to it. GEM will then visualize the application model according to the rules defined in the stylesheet. The default stylesheet gives an 1:1 view of the graph. Custom stylesheets can provide an abstracted view of the model, depending on the actual needs. For example, an SQL schema could be represented in an UML-like notation, as an E/R diagram, or even as a set of plain, pretty-printed DDL statements. The design of the GEM prototype makes it easy to enable support for multiple views of the same application model. For example, an abstract view can be combined with a detail view (maybe only of the selected part of the graph), or different spanning trees for hierarchic displays of the same model may be produced. This can be achieved by using different stylesheets on the same model.

Editing functionality can be separated into two categories: GEM directly provides a basic layouting functionality pertaining solely to the presentation of the model, like manually or automatically layouting graph elements. These operations will not change the application model. All such layout data is preserved between edit sessions. To actually edit the application model, a stylesheet provides *edit operations*. An edit operation is an arbitrarily complex, model-specific operation. A stylesheet for SQL schemas could provide operations like "create new table", "add column to table", or "rename table". For model management, there also might be more complex operations like "copy table" or "denormalize tables". GEM automatically provides a menu with all possible edit operations defined in the stylesheet. To apply an operation, the user might have to select a part of the graph (e.g., the table to which he wants to add a column) and provide input parameters (like column name and type). Then the operation is executed, directly changing the application model. After that, the visualization is updated to reflect the changes in the application model, preserving as much of the manual layouting as possible. Manipulations done through one editor window will directly update the views in the other windows. At any point, the user can export the application model for further processing by other tools.

2.2 Stylesheets in GEM

One of the distinguishing characteristics of GEM is the use of graph stylesheets to customize and adapt the editor to a specific data model. This allows us to visualize and edit not only SQL schemas or XSD files, but *arbitrary* metadata, possibly in a specialized format. Thus, the customization process must be powerful enough to cope with a great diversity of requirements, but also simple enough to allow for a fast adaptation of GEM to a specific data model even by non-experts. We believe that we have achieved this goal and in this section we will illustrate this by looking at GEM from the perspective of a stylesheet developer.

The first step in developing a custom stylesheet is creating the *visualization part*. The developer needs to define rules to specify how elements in the application model should be displayed in the edit pane. This is similar to the development of CSS or XSLT stylesheets.

To represent stylesheets, the current GEM prototype uses an XML format, but future versions will support a more concise textual and later a more vivid graphical representation. With a stylesheet, a developer can choose to present an SQL model as plain text DDL, or choose to hide details and only show the table names and how they are related through foreign key relations. The details of defining these rules are explained in Section 4.

The second part of a stylesheet are the *edit operations*. Instead of having users resort to tedious, fine-grained operations on the atomic elements of the graph structure like with other graph editors, the stylesheet developer defines high-level edit operations for all tasks that should later be performed by users of the stylesheet. This not only greatly improves the usability of the resulting editing functionality, as very complex modifications can now be performed with a single editing step. Properly designed edit operations can also guarantee the consistency of the application model, as users can only perform semantically valid edits. Moreover, edit rules allow us to resolve the view-update problem which arises when editing a model through an abstracted visualization.

For most requirements, using the built-in widgets and functions is sufficient for defining visualization rules and edit operations to adapt GEM to custom needs. Sometimes, however, the built-in functionality might not be enough, or might require very complex and non-intuitive stylesheets. Therefore, GEM allows customization not only by defining declarative stylesheets, but also by providing an extension mechanism for user-defined functions and widgets.

User-defined functions in GEM are written as static Java methods, allowing to implement virtually any domain-specific functionality. In a graph stylesheet, the developer can use these functions in visualization rules and edit operations. GEM provides several basic widgets to visualize application models. Besides simple boxes, ellipses, or arrows, there are grouping widgets which allow sophisticated layouts. If this is not sufficient, the stylesheet developer can define his own widgets by implementing a GEM widget interface.

2.3 GEM architecture

Figure 1 gives a high-level overview of the important concepts of our approach and their interactions. In the center, the complete model graph consisting of the application graph and the visualization graph is depicted. The two subgraphs are connected via RepresentedBy edges, each connecting an element in the application graph with an element in the visualization graph that represents this element. Application and visualization elements can be in n:m relationships, as an application graph element can be represented by more than one visualization node (and thus appear more than once in the edit pane), and a single visualization node can represent more than one metadata element.

On the right hand side of the figure, the editor pane and the widgets are depicted. They are responsible for the display of the visualization graph and observe it for changes. Visualization rules are used to initially create and later update the visualization graph and the RepresentedBy edges. They can refer to the application graph as well as to the visualization graph, but they can modify only the visualization graph. Edit operations can refer to both subgraphs, but modify only the application graph and never the visualization graph. To keep the visualization synchronized with the application model, each application of a

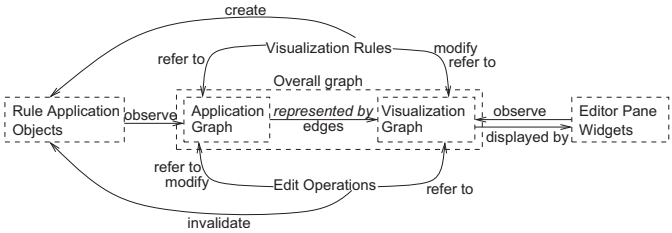


Figure 1: Overview of the stylesheet-based graph visualization and editing approach.

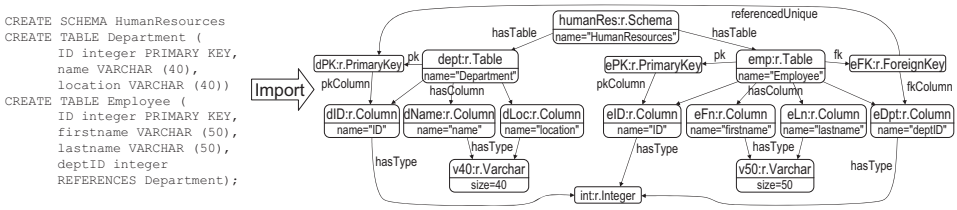
visualization rule is recorded in a rule application object (RAO). Execution of edit operations can invalidate RAOs, if after the edit operation the premise for the rule application does no longer hold. Invalidated RAOs are then undone and visualization rules are selectively re-applied, so that after a change of the application model this change is directly reflected in the visualization model. We will discuss this in greater depth in Section 7.2.

3 Metadata representation

In this section, we will first introduce graphs as the common representation for all metadata to be edited with GEM. Graphs are probably the most general data structure, allowing a lossless representation of virtually any metadata artifact. In general, a graph consists of a set of nodes N that are connected by a set of edges E. Many variations of graphs exist that differ in aspects like their support for labels and attributes on nodes and edges, whether edges are directed or can connect more than two nodes (hypergraphs) etc. Likewise, the formal definitions differ, e.g., whether edges are seen as independent objects or are only given as a relation $E \subseteq N \times N$. We have chosen to use directed, attributed, labeled multigraphs, since they strike a median point in between the verbose representations that result from using a very basic graph formalism (e.g., labeled graphs) and the complexity of some of the more elaborate formalisms. Our Definition 1 is based on [KR90]:

Definition 1 (Directed, attributed multigraph) *A directed, attributed multigraph G is a tuple $(N, E, src, tgt, \Gamma, \Sigma, att_N, att_E)$. N and E represent the set of nodes and edges, resp. Γ is the set of attribute identifiers. Σ is the set of attribute values. The mappings $src\colon E \to N$* and *$tgt\colon E \to N$ associate a start and end node with each edge. The mappings $att_N\colon N \times (\Gamma \cup \tau) \to \Sigma \cup \bot$* and *$att_E\colon E \times (\Gamma \cup \tau) \to \Sigma \cup \bot$ return the (possibly empty) value of the given attribute identifier for the given node or edge, respectively. τ is the identifier of a special attribute representing the* type *of nodes and edges.*

The advantages of this basic graph metamodel is its generality and minimality. Nodes can represent arbitrary application elements, while edges can be used to represent the relationships between them. Node attributes represent the properties of objects, while edge attributes can be used to model more refined kinds of relationships, e.g., a *position* attribute can specify an ordered relationship. Metadata to be edited with GEM first has to be transformed from its native representation (e.g., SQL DDL statements) to a multigraph according to Definition 1 (see Figure 2). The editor does not place any further restrictions on



a) SQL metadata in native representation

b) SQL metadata in graph representation

Figure 2: Representing metadata as graphs.

this representation. The graph representation can therefore be customized to the applications needs, e.g., to ease the definition of stylesheets or to provide round-trip capabilities. GEM's primary vehicle for graph import is the Graph Exchange Language [WKR01], an XML format for the exchange of graphs and type graphs. The research projects PALADIN [Gö05, GD07] and Caro [Stu06, SD07] provide converters from their internal representation of models to GXL, enabling the import of, e.g., SQL DDL and XSD. Also, direct extraction of metadata from the information schema of an RDBMS is supported. The effort for converting custom models to and from GXL is manageable, since mapping of any kind of object-oriented representation to and from our graph model is straightforward.

4 Graph transformations

In this section we will describe the basics of our graph transformation formalism that is the foundation for our graph stylesheet language. While graph transformations have been a research topic for approximately 40 years, they are still not widely employed outside the graph community. Only recently has the concept attracted rising interest in the context of model-driven architecture (MDA), where it is used for model-to-model transformations (e.g. [CH03]). Two main reasons for this lack of acceptance can be named: First and foremost, very few graph transformation systems (GTSs, also *graph rewrite systems*) have reached a degree of maturity beyond that of research prototypes. A second problem for the limited acceptance of graph transformation systems as a generic tool for software and information systems engineering is the diversity of existing approaches, both regarding the formal semantics of transformation rules and systems, as well as in the different languages used to describe them. We have no intention to discuss the benefits and drawbacks of each of the approaches to motivate our formalism, but instead hope that we can let its clarity and naturalness speak for itself. A common criticism on graph representations and operations is the issue of computational complexity: many graph problems, in particular the subgraph isomorphism problem, which is of high relevance for graph transformations, are known to be NP-complete. However, since most practical scenarios use labeled and regular graphs, the average case complexity is well within reasonable bounds, as we will see in our evaluation in Section 7.3.

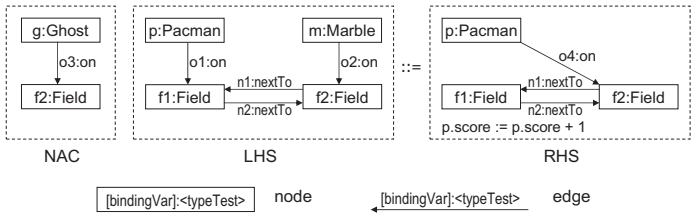


Figure 3: A simple production rule.

4.1 Production rules

A graph transformation system takes as inputs a *host graph* representing the input data to be transformed, and a set of graph transformations rules. It then outputs a new graph or changes the input graph. To specify an individual graph transformation, we use a syntax based on production rules (see [Hec06] for an overview). They have a *left-* and a *right-hand-side* (LHS, RHS). As an example, Figure 3 shows a simple production rule that models a part of a game of Pacman, where the host graph represents the current game state (labyrinth fields and their connections, the position of Pacman, ghosts, and marbles). The LHS specifies a *graph pattern* consisting of pattern nodes and edges (*pattern elements*), which has to be found in the host graph for the rule to be applicable. Pattern nodes and edges can specify a type test, given as the part of an element's label behind the colon. In our example rule, the LHS specifies a situation where Pacman *p* is on a Field *f1* that is next to a Field *f2* containing a Marble *m*. A subgraph of the host graph that matches the LHS is called an *occurrence*. Occurrences can be seen as a mapping from the elements of the graph pattern to those of the host graph. For the scope of this paper, we assume occurrences to be injective, i.e., an occurrence will never map two pattern elements to the same host graph element.

The RHS describes how an occurrence is modified by the rule. To indicate corresponding elements on LHS and RHS, *binding variables* are used. They are essentially an identifier for pattern elements and form the first part of a rule element's label in front of the colon. Elements on the LHS that also occur on the RHS indicate that the element is preserved, for example the Field nodes *f1* and *f2*, or the Pacman node *p*. An element on the LHS that does not occur on the RHS indicates that the element will be deleted (e.g., the Marble node *m* or the on edges *o1* and *o2*), while an element occurring only on the RHS signals the creation of a new element (e.g., the on edge *o4*). The part of a RHS element's label behind the colon is a type assignment, to indicate the type of new nodes and edges, or to change the type of preserved elements. In addition to changing node types, a RHS can modify attributes of nodes and edges by referring to their binding variables (e.g., incrementing Pacman's score attribute). So our example rule will move Pacman from his current field to a neighbouring field containing a marble. In the process, the marble will be consumed and Pacman scores a point.

Additional *application conditions (ACs)* can be specified to limit the applicability of the rule. These can refer to attributes of LHS elements (pure *attribute conditions*) or specify graph patterns that must or must not occur for the rule to be applicable (structural ap-

plication conditions): Positive structural application conditions (PACs) specify additional structures that have to be fulfilled, while negative structural application conditions (NACs) specify structures that must not occur. Pattern elements in ACs can also be connected to the LHS by binding variables. In our Pacman example, the rule is only applicable when there is no Ghost *g* on the Field *f2* containing the Marble *m*.

For a more compact presentation of rules, we will use an *integrated representation* in the remainder of this paper. Figure 4 shows the rule of Figure 3 in integrated representation: deleted nodes and edges are indicated by dashed lines, while created elements are shown with bold lines. Elements of application conditions are grouped and marked with PAC or NAC, respectively. While the expressiveness of this representation has some limitations, it is sufficient for the compact representation of the rules used in this paper.

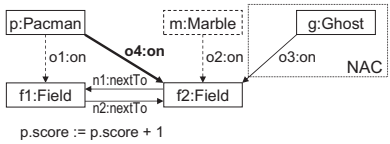


Figure 4: The production rule of Figure 3 in integrated representation.

4.2 Operational semantics

The operational semantics of graph transformation systems varies for the different approaches proposed in the literature. The original mode of application of a set of production rules is similar to that known from formal languages: As long as production rules are applicable, the system randomly chooses both a rule and an occurrence, and applies the rule. This non-deterministic rule application often makes it difficult for the author of a graph grammar to ensure that the rules do exactly what is intended. Very often, rules have to be applied in a certain sequence or to specific regions of a host graph. A number of approaches have been introduced to control the application of rules more precisely (see [BFG95] for an overview). It should be noted that none of them increases the expressive power of a GTS as a whole, but can considerably reduce the number and complexity of individual rules.

Rule layering is a concept that was first introduced by AGG [Tae99], a graph transformation system implemented in Java. Layers define a partial order among rules: A layer contains a number of rules which are applied non-deterministically as described above, until no more rules of the layer are applicable. Rule application then continues with the rules from the next layer. Within a layer, the formal properties of non-deterministic graph grammars hold, while the ordering gives a basic degree of control over the application of rules. In GEM, the visualization of application graphs is performed using a restricted layering approach, where each layer contains exactly one rule (see Section 5.1).

While layering is sufficient for simple sequential application of rules, complex operations cannot be expressed as easily. We wanted to be able to specify complex operations in a natural way, so we chose the three basic control structures introduced by Göttler [Gö88] due to their limited complexity and minimality for the definition of edit operations: *Se-*

quential application (SAPP) applies the contained rules (or nested control structures) once in the given sequence. *Case application (CAPP)* will apply the first applicable rule of a given ordered list of rules. *While applicable (WAPP)* will apply a contained rule (or nested control structure) repeatedly, until it is no longer applicable .

5 Graph stylesheets

With the basics on graph transformations introduced in the previous section, we now present GEM's graph stylesheet language. As described earlier, a graph stylesheet consists of *visualization rules* and *edit operations*: Visualization rules describe what kind of visualization model elements should be produced for which structures (subgraphs) of the application model. All visualization rules together create the complete visualization graph for the application model. The semantics of stylesheet application is non-destructive, i.e., application model subgraphs are not replaced by their visualization model counterparts, but instead application and visualization model coexist in one common graph. All visualization model elements are connected by special RepresentedBy edges to all those (possibly many) application model elements they represent. In conjunction with the currently selected widgets, an edit rule can use the RepresentedBy edges to determine the affected application model elements and apply the appropriate changes.

5.1 Visualization rules

As previously mentioned, visualization rules follow a restricted layering approach, where each rule is applied until no further occurrences exist, after which processing continues with the next rule. Since the visualization of large graphs requires a large number of rule applications, some restrictions have been defined to allow an optimized processing, and also help to make the visualization rules "safe": First, visualization rules must not modify, delete or create any elements outside the visualization model. Further, visualization rules may only *create* new elements in the visualization model and may not overwrite attribute values. These restrictions are needed to allow for the selective updating described in Section 7.2. Visualization rules may only be applied once per occurrence. This is enforced by an implicitly created (negative) application condition that corresponds to the newly created elements on the RHS. Moreover, negative application conditions must not refer to elements that could have been created by a previous application of the same rule. That way, we assure that the sequence in which a visualization rule is applied to its occurrences does not change the outcome of the stylesheet. It also allows optimizing the search for occurrences, as we do not have to recompute them after each rule application.

To illustrate the basic use of visualization rules, we defined a *default stylesheet* shown in Figure 5, which visualizes any application graph in a 1:1 fashion. The XML representation of the stylesheet was omitted due to space constraints. The stylesheet consists of two rules: displayNodes visualizes the application model nodes: Each one is represented by a Box widget, carrying the node's type attribute as label. To connect application and visualization model, we add a RepresentedBy edge between the application model node and the

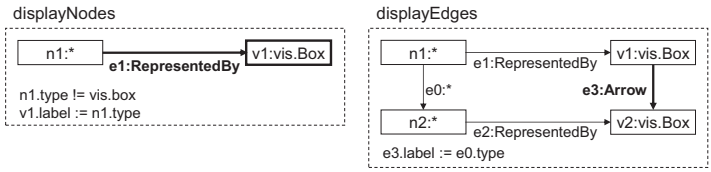


Figure 5: Visualization rules for a generic labeled digraph, integrated graphical syntax.

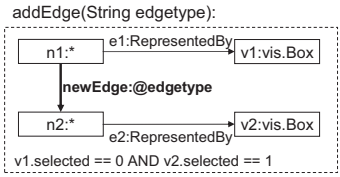


Figure 6: A simple edit operation (integrated representation)

visualization model Box node. Due to the layered application of visualization rules, the displayNodes rule will be applied on all graph nodes. To make sure that we do not create boxes for other boxes, we add an attribute condition indicating that the node *n1* must not be of type vis.Box. The second visualization rule displayEdges looks for two nodes which are connected by an arbitrary edge. For each such pair of nodes, a new Arrow edge is created between their Box nodes, taking the type of the connecting edge as label. Note that we can omit tests to guarantee that *n1* and *n2* are actually application model elements, since the first rule already makes sure that only application model nodes will have RepresentedBy edges to box nodes. This simple stylesheet will already reproduce the graph's structure and typing. In Section 6 we will demonstrate how a more complex set of visualization rules can be used for a customized representation of proprietary metadata.

5.2 Edit operations

For metadata manipulation, a stylesheet defines a set of edit operations. An edit operation consists of a number of production rules (edit rules) and a control flow specification using the control flow constructs (*WAPP*, *CAPP*, *SAPP*) from Section 4.2. A user modifying the graph through an edit operation in general wants the operation to be applied on a specific set of elements (e.g., a specific table or column) at a time. To indicate the place of rule application, the user selects the appropriate widgets in the editor pane. Their selection state is propagated to the underlying visualization graph, where it can be referred to by edit rules. Besides selecting the widgets of affected elements, often user input is needed for the rule, e.g., the name of a new table or element. Edit operations can therefore specify *user parameters*. When the edit operation is activated, the user can enter the appropriate information.

Figure 6 shows a very basic edit operation with only one edit rule that adds an edge between two nodes selected by the user. The rule specifies a user parameter edgetype, which is used in the created part to set the type attribute of the new edge. Note the reference

to the selected state of the visualization model nodes: A selected value < 0 indicates that the element is not selected, while values ≥ 0 indicate the sequence of selection. So the rule adds the edge pointing from the node selected first to the node selected second. Also note that the edit rule does *not* add an Arrow edge between the two Box nodes representing the application model nodes, as updating the visualization model is left to the visualization rules. The method for updating will be explained in Section 7.2.

6 Example scenario

The development of GEM was initiated by the practical needs in the projects PALADIN [Gö05] and Caro [Stu06] of our group. To demonstrate GEM's usefulness as a generic metadata editor, we will present a stylesheet that displays arbitrary models represented in the internal representation of the PALADIN metamodel architecture and allows the editing of semantic correspondences between these elements. We will use a simplified version of the stylesheet to illustrate GEM's customizability beyond the basic examples shown so far. A first step to integrate GEM with PALADIN was to provide a serialization of models given in PALADIN's internal object-oriented representation to GXL graphs. Objects and values of atomic fields map naturally to GXL nodes and their attributes, while values of complex fields themselves map to nodes. An object's class is stored as the type label. References between objects and complex fields are mapped to edges. Since many multi-valued references are ordered, we use a position attribute on edges to preserve the ordering. This very basic mapping is already able to provide full round-trip capability, i.e., PALADIN models can be losslessly reconstructed from the GXL representation.

PALADIN uses a *Core metamodel* to capture common aspects of different actual metamodels. Elements of concrete metamodels inherit from the Core elements. This allowed us to start with a single set of rules that uniformly displays arbitrary schemas (e.g., SQL or XML Schema models), since it is sufficient to refer to the common base classes. For example, SQL tables and columns as well as XML elements and attributes are all subclasses of the Core class Feature. In addition, concrete relationships like hasColumn (between table and column) or subelement (between an element and its subelements and attributes) are derived from the Core relationship content. So only two visualization rules were sufficient to display any PALADIN model. A more elaborate version of the stylesheet can easily provide metamodel-specific representations of schemas. In the current GEM prototype, node type inheritance is implemented via a user-defined function *subclassOf* which takes the full-qualified names of two node types A and B as input, and returns true if A is the same class or a subclass of B, or false otherwise. We use this function in the rule's application conditions.

Figure 7 shows the displayFeatures rule to create properly labeled widgets for PALADIN Features. Unlike in the default stylesheets' displayNodes rule, a Group node instead of a plain Box is created for each Feature node. A Group can contain other widgets and display them in a hierarchical fashion. By using two application conditions we make sure that the node is a subclass of Feature and no subclass of Match (matches are displayed by a separate rule). The Group's label is created by concatenating the Feature's actual type (e.g., Column, Table, etc.) and the name attribute common to all PALADIN classes.

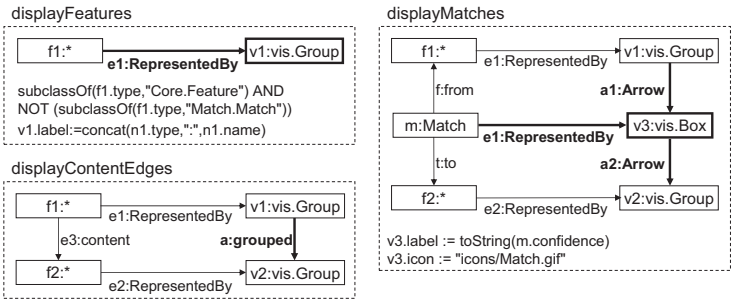


Figure 7: The visualization rules for PALADIN models and semantic correspondences.

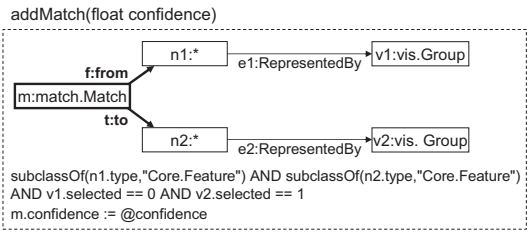


Figure 8: An edit rule to add a match between two PALADIN model elements

The displayContentEdges rule is based on the displayEdges rule and searches for pairs of nodes that are already represented by a Group widget and connected by a content edge. We can directly refer to the content edge type in the stylesheet, since we decided for our model serialization to only create edges for base references, i.e., there are no distinct edges for references that are derived from content, like hasColumn. For each content edge, a new grouped edge is created that indicates membership of its target node in the source group.

The third rule displayMatches creates the representation for Match nodes and their to and from edges that indicate which two features the match connects. It searches for pairs of nodes that are already represented by a Group each and are connected by a Match node. The Match node itself is represented by a newly created Box, which is distinguished from other boxes representing schema elements by adding an icon. The Match node's confidence attribute is set as a label. The to and from edges are represented by unlabeled Arrows. Note how both Arrows now point into the direction of the to edge, so that the match's direction can be determined visually without edge labels.

To create new matches, an addMatch (Figure 8) edit rule has been defined. Like the createEdge rule, it refers to the selection state of the visualization elements and adds a Match node whose from edge points to the node selected first, and whose to edge points to the node selected second. The match's confidence is queried from the user via a user parameter. Figure 9 shows a schema matching session with GEM using only this simply stylesheet. The result of a somewhat more elaborate stylesheet for editing entity-relationship models is shown in Figure 10.

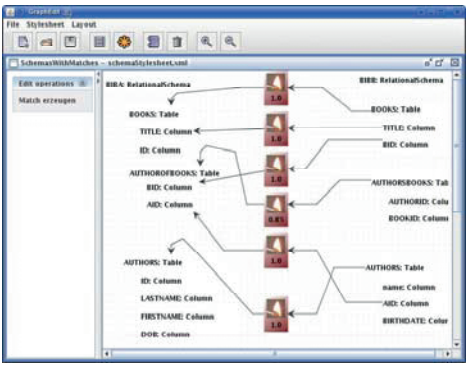


Figure 9: A schema matching session.

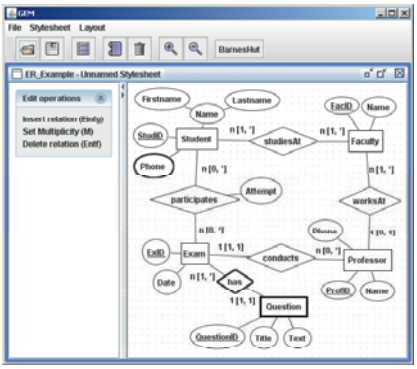


Figure 10: An ER-modeling session.

7 Implementation and performance

In this section, we discuss the most interesting aspects of our GEM prototype. Also, we give results on GEM performance for different sized application models. The GEM prototype was developed as part of a diploma theses [Jö07] in our group. For the display of graphs we use the JGraph[2] library. It also provides the basic user interaction functionality, like automatic layout, moving and resizing nodes, rerouting edges etc.

7.1 Using a relational database for graph transformations

As previously mentioned, the graph transformation engine used by GEM has been built in-house taking advantage of the features of relational database systems. The main rationale behind a custom implementation was the insufficient performance or maturity of the few existing publicly available graph transformation engines.

For each graph loaded into GEM, a new schema with four relations is created, to store nodes, edges, node attributes and edge attributes. The schema definition is straightforward, with the exception of the `Edge` table, which is denormalized. Besides the edge information, it also contains the complete information for nodes (their id, type and current selection status) to improve query performance by reducing the number of joins needed. A separate `Node` table is still needed to represent unconnected nodes.

Query generation For each production rule of a stylesheet, an SQL query based on the rule's LHS (which corresponds to the union of the elements in the preserved and deleted sets when shown in our integrated representation) and application conditions is generated, as well as a number of `DELETE`, `INSERT`, and `UPDATE` statements for the RHS. To illustrate query generation, the query generated for the displayEdges rule of the default stylesheet from Figure 5 is shown in Figure 11.

For each edge in the LHS, a reference to the `Edge` relation is added to the `FROM` clause

[2] http://www.jgraph.com

```
1 SELECT e0.id AS e0, e1.id AS e1, e2.id AS e2, e1.dest AS v1, e0.src AS n1, e2.src AS n2, e2.dest AS v2
2 FROM EDGE e0, EDGE e1, EDGE e2
3 WHERE e1.edgetype = 'vis:RepresentedBy' AND e1.desttype = 'vis:Box' AND
4     e2.edgetype = 'vis:RepresentedBy' AND e2.desttype = 'vis:Box' AND
5     e0.src = e1.src AND e2.src = e0.dest AND
6     NOT e1.dest IN (e0.src, e2.src, e2.dest) AND NOT e0.src IN (e2.src, e2.dest) AND
7       NOT e2.src IN (e2.dest) AND NOT e0.id IN (e1.id, e2.id) AND NOT e1.id IN (e2.id) AND
8     NOT EXISTS (SELECT * FROM EDGE e3
9       WHERE e3.edgetype = 'vis:Arrow' AND e3.src = e1.dest AND e3.dest = e2.dest AND NOT e3.id IN (e0.id))
```

Figure 11: The query generated for the LHS of the *displayEdges* rule of Figure 5

(line 2), and the edge's ID is added to the `SELECT` clause (line 1). If the edge specifies a type test, an appropriate predicate is added to the `WHERE` clause (lines 3, 4). For each node in the LHS, the query generator determines its in- and outbound edges. If at least one such edge exists, the node's ID is already available in this edge's reference to the edge relation. Then simply a reference to the corresponding `src` or `dest` column is added to the `SELECT` clause. If the pattern node has several connecting edges in the LHS (i.e., its ID occurs in several edge relation references), we have to make sure that all edges refer to the same host graph node. So the IDs of source and target nodes of edges connecting to the same node are tested for equality in the `WHERE` clause (line 5). Only if the pattern node is isolated (i.e., not connected via edges to other elements of the LHS), a reference to the `Node` relation is added to the `FROM` clause, as well as a type test predicate to the `WHERE` and the node ID to the `SELECT` clause (not in this example). As discussed in Section 4.1, we are by default looking for subgraphs of the host graph that are *isomorphic* to the rule's pattern graph, i.e., we have to make sure that no single host graph element matches several rule elements. So for each node and edge, we conjunctively add an *isomorphic* predicate to the `WHERE` clause that ascertains that its ID is not the same as the IDs of one of the other pattern elements (lines 6–7 and 9). For structural application conditions, conditional subqueries are generated in the `WHERE` clause. For example, lines 8–9 show the implicit negative application condition generated based on the rule's *created* part. The algorithm to create the subquery is the same as the one for the main query, just with a simplified `SELECT` clause, and with outer references for those elements that have been identified with LHS elements (line 9). For each node and edge attribute occurring in an application condition, a join with the `Node` or `Edge` relation reference and the `NodeAttribute` or `EdgeAttribute` relation is added to the `FROM` clause (not in this example). An join condition and the application condition itself are conjunctively added to the `WHERE` clause.

Processing pattern query results Each row in the result set of a pattern query represents a single occurrence in the host graph. One transformation step selects one of these occurrences and modifies the host graph appropriately. If the rule is to be applied repeatedly, as is the case for visualization rules, in general it would not be possible to continue with the results obtained from the first issuing of the query, since the modification could have invalidated previously valid occurrences or created new possible occurrences: (1) nodes or edges of the occurrence or an application condition could have been deleted, or (2) new elements could be created so that a negative application condition would now fail. Also, (3) attribute values could have been modified causing an attribute condition to fail.

However, with the restrictions defined for visualization rules (Section 5.1) none of these cases can occur: Visualization rules must not delete any elements, eliminating case (1). Their negative application conditions must not refer to elements possibly created by a previous application of the same rule, so case (2) is avoided. In addition since attributes once initialized must not be modified, case (3) cannot occur as well. So for each visualization rule, its respective query has to be issued only once, instead of reissuing it after applying the RHS on a single occurrence. Since the actual number of occurrences for a rule depends on the rule and on the structure of the application graph, we can give no general estimate on the improvement due to this optimization. However, for most real-world stylesheets and application graphs, the number of occurrences scales with the size of the graph, which is several orders of magnitude larger than the number of rules or queries. For each result tuple, the `INSERT`, `UPDATE`, and `DELETE` statements prepared for the rule's RHS are executed, parameterized with the node and edge IDs taken from the result tuple. To set the type and selected state of the source and target nodes of a new edge, subqueries are used. Once all tuples of a visualization rule are processed, the query for the next visualization rule is issued. When all rules are processed, the editor can interpret the visualization graph and display it to the user.

The statement generation is the same for edit rules. The control flow statements SAPP, CAPP, and WAPP are interpreted by the editor, who will, depending on the outcome of rule applications, choose the query for the next rule and move forward in the control flow. Since the restrictions imposed on visualization rules do not apply for edit rules, the optimization described for visualization rule processing can not be applied to a single edit rule surrounded by a WAPP statement. Instead, the query has to be issued again after one occurrence is processed. Since edit rules are usually only applied selectively to small parts of the graph, this does not imply a noticable performance penalty.

7.2 Selective updating of the visualization

As described in Section 2.3, edit rules are not supposed to modify the visualization graph, but only the application graph. Therefore, once an edit operation applied its changes on the application model, the visualization rules have to be applied again, so that the representation reflects the changes. A naive implementation would simply discard the existing visualization graph and re-apply the stylesheet on the entire application graph. This is undesirable for two reasons: First, the application of a stylesheet is a costly operation which can take a noticeable amount of time. Second, simply discarding the visualization graph would also loose all those kinds of information that were not derived from the application model but result from *non-application relevant edit operations* (Section 2.1), like moving or resizing of widgets or adding bends to edges etc. Given the effort needed for a manual layout, this information has to be maintained as completely as possible even in the presence of edits. We therefore employ a *selective updating* of the visualization model, which only includes the elements that were affected by the edit operation.

In a first step, the effects of visualization rule applications that were possible in the original application graph, but are no longer applicable in the edited graph, have to be undone. Afterwards, the new application graph has to be tested to find out if new visu-

alization rule applications are possible. To be able to trace rule applications, we use *rule application objects* (RAOs). They contain a reference to the rule and the occurrence used, as well as information on the nodes, edges and attributes that have been created or set by the rule application. Following the observer pattern, a RAO registers itself as observer to certain events on those elements of the application model that are part of its occurrence: (1) If a *deletion* event occurs on the nodes and edges of a RAO's occurrence, the rule is no longer applicable and is undone using the information in the RAO on created objects and set attribute values. (2) If an attribute used in an *attribute* condition *changes*, the rule has to be retested for applicability. If it is not applicable, the rule is undone. (3) If the value for an attribute assignment on the rule's RHS is *derived* from one or several attributes in the LHS, the value has to be recalculated if one of the base attributes changes.

Besides observing existing objects, negative application conditions that were fulfilled when the rule was first applied might no longer hold after an edit operation. Since these objects were non-existant, the RAO cannot register itself to any concrete object, but only to abstract insertion or deletion events of nodes and edges that have the same type as one of the elements in one of the rule's NACs. Since NACs can also contain attribute conditions, the RAO also has to register itself to any modifications of attributes that belong to a node or edge which has the same type as the node or edge in the NAC and have the same attribute name. For positive application conditions (PACs), similar considerations hold, but here we are interested in *deletion* modification events on nodes, edges, and their attributes.

7.3 Performance

In the conceptualization phase for GEM, we evaluated existing graph transformation systems. The only system that seemed both mature enough and did not depend on legacy system environments was the AGG system [Tae99]. We evaluated its performance by measuring the time required for applying a basic stylesheet consisting of two visualization rules. Our measurements discouraged the use of AGG as GEM's underlying engine. We therefore chose to implement our own GTS, building on a relational DBMS to do subgraph search and transformation. We first implemented a basic version of the DBMS-based GTS and tested it against the same input schemas using three different underlying database systems, two open-source systems (Derby and PostgreSQL), and IBM's DB2. All three systems scaled nearly linearly with application graph size and number of occurrences, with the commercial system outperforming its open-source contenders by an approximate factor of two. For small- to medium-sized graphs, both showed acceptable matching times, DB2 outperforming AGG by a factor of up to 120. Encouraged by these promising results, we decided to go for a DBMS-based solution.

While the initial measurements helped us estimate which degree of performance and scalability to expect, they were no final indication of the actual performance of the editor. We therefore ran a series of benchmarks of the latest version of GEM against four different DBMS (now also including the embedded H2 DBMS), application graphs of different sizes, and stylesheets of different complexity. The *default* stylesheet is the same as presented in Section 5. The *PALADIN* stylesheet is a slight variation of the stylesheet from Figure 6, modified to provide a representation that better suits SQL metadata. For a realis-

Stylesheet	Graph size	DB2	Derby	PostgreSQL	H2
default	157N/239E	1.4	2.3	2.0	0.7
default	237N/405E	1.7	3.0	1.6	1.0
default	599N/1026E	3.0	4.7	3.5	2.6
default	1068N/2088E	4.7	9.5	7.5	4.6
default	1785N/3723E	8.1	14.2	11.5	7.7
PALADIN	6T/67C	1.3	2.2	1.4	0.4
PALADIN	12T/111C	1.6	2.6	2.0	0.5
PALADIN	25T/294C	3.1	5.4	4.1	1.2
PALADIN	50T/602C	6.6	8.4	8.4	2.2
PALADIN	100T/1071C	9.6	14.6	12.0	3.4

Table 1: Stylesheet application time (in seconds) on different application graph sizes and DBMSs.

tic example scenario, we chose the schema of the open-source OpenExchange groupware system[3], scaled to five different sizes. We ran the benchmarks on a typical desktop environment, a 2GHz Core Duo machine with 2GB RAM and locally installed DBMSs. We repeated each of the individual tests three times and averaged the results.

The first part of Table 1 shows the results of applying the default stylesheet to the five different graphs using the four DBMSs. We see that all perform adequate for small and medium sized graphs, with DB2 maintaining its lead over Derby and PostgreSQL. A pleasant surprise was the H2 database, which – unlike Derby – appears to take considerable advantage of the reduced communications overhead of an embedded DBMS. This is particular obvious both in the significantly higher base response time of the other three systems and in H2's better scalability with increasing graph size and occurrence count. The second part of Table 1 shows the results for the PALADIN stylesheet: H2 extends its lead, as it apparently benefits from the reduced insert and update load of this stylesheet, which aggregates metadata and thus creates less nodes and edges for the visualization model. While H2's snappy response times make it the ideal choice for our example stylesheets, additional experiments (not presented due to space constraints) have shown its performance to drop considerably once rules (and thus queries) get more complex. This behavior can be explained with known limitations of H2's current cost-based optimizer. So for larger graphs and complex rules, DB2 is usually the better choice, as here its principal disadvantages and its significant schema setup overhead during graph import are offset.

8 Related Work

The basic principle we follow with our approach is the separation of model and view in a generic way. This is a pervasive paradigm in computer science. A similar approach to ours, although limited to displaying and editing XML files, is the XMLmind XML Editor (XXE)[4]. It uses Cascading Stylesheets with some proprietary extensions to display XML files. For editing, the user can insert and delete elements or change attribute values. XXE makes sure that the resulting document is always valid and corresponds to its DTD or XML

[3] http://www.open-xchange.com/

[4] http://www.xmlmind.com/xmleditor

schema definition. For more complex operations it is possible to define element templates, the equivalent to the edit operations in GraphEdit. Another system that inspired the development of GEM are the Graph Stylesheets used in IsaViz[5]. They are used to style RDF graphs, but are not able to abstract from the structure of the underlying RDF graph, as they just provide means to change color, shape and fonts of the nodes. The Eclipse Modeling Framework (EMF)[6] supports the generation of basic editors for object-oriented models. However, representation is limited to a tree-like structure following the containment relationships among objects. Further, the default editors can only perform atomic editing operations (creating objects and references between them, setting values of fields etc.). Any kind of higher-level editing functionality requires complex customizations through Java code. The Graphical Modeling Framework (GMF)[7] builds on EMF and allows the creation of truly graphical editors. However, the graphical definition models used by GMF are less expressive than GEM stylesheets. Consequently, the shape of the visualization graph is largely predetermined by the structure of the underlying EMF application model and therefore does not allow the degree of abstraction that is possible with GEM. Any customizations of a generated GMF editor require Java coding.

The idea to implement graph transformations with a relational database system is in itself not new. Previous implementation attempts have been criticized for severe performance deficits: In [VSV05], the authors compare a number of native GTSs (among them the aforementioned AGG) with an RDBMS implementation. According to their initial measurements, performance of the DBMS solution was overall far inferior to the native GTSs. These findings not only conflicted with our observation that even in a prototypical state, GEM's graph transformation engine significantly outperforms the AGG system by several orders of magnitude, but have also since been revised by the authors in [VFV06].

9 Conclusions and Outlook

In this paper, we presented GEM, a visualization tool and editor for arbitrary metadata. To internally represent the metadata to be visualized and edited, we use a model based on attributed, typed multigraphs, which is general enough to losslessly represent any kind of metadata. Visualization of the metadata is customized via user-provided stylesheets, which allow for a wide variety of different visual representation, like arranging metadata elements in – possibly nested – tables or in trees. In order to hide details or to focus on a special aspect of the model, stylesheets can perform aggregations that abstract from the concrete metadata representation. Also, one can visualize and edit heterogeneous metadata in an integrated way. Both visualization rules and edit operation are based on the concept of graph transformations.

Our metadata editor is not only useful for editing schemas and matches, but supports any data which can be reasonably represented as a graph. In the semantic web community, visualizing and editing ontologies and RDF data in an adequate and user-friendly manner is still an open problem. Here, GEM can possibly be of great value, because it allows the

[5] http://www.w3.org/2001/11/IsaViz/
[6] http://www.eclipse.org/modeling/emf/
[7] http://www.eclipse.org/modeling/gmf/

fast creation of domain-specific visualization and editing tools for arbitrary ontologies, and allows to "'mix"' different ontologies within one document. Other possible applications include networking problems, mind maps, or any other area that uses graphs to represent its data. The editor's underlying graph transformation system can also be used independently to do reasoning on the application graph, or to perform model-to-model transformations. Some open issues remain: Currently, GEM supports node and edge type inheritance only by user-defined functions. This is a flexible mechanism and allows the adaptation of the type model to the respective needs. Introducing type graphs into GEM would make type inheritance a first class concept, allowing for better consistency checking in application models. These aspects, and a much more powerful GT language with a more concise textual syntax are supported by an advanced GT engine developed in the context of the PALADIN project. We intend to use this engine in future versions of GEM.

References

[BFG95] D. Blostein, H. Fahmy, and A. Grbavec. Practical use of graph rewriting. Technical Report 95-373, CDN, 1995.

[BHP00] Phillip A. Bernstein, Alon Y. Halevy, and Rachel A. Pottinger. A vision for management of complex models. *SIGMOD Rec.*, 29(4):55–63, 2000.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. volume OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[Gö88] Herbert Göttler. *Graphgrammatiken in der Softwaretechnik: Theorie und Anwendungen*, volume 178 of *Informatik-Fachberichte*. Springer-Verlag, Berlin, 1988. Venia Legendi Thesis (Habilitation).

[Gö05] Jürgen Göres. Towards Dynamic Information Integration. In *Data Management in Grids*, number 3836 in LNCS, pages 16–29, 2005.

[GD07] Jürgen Göres and Stefan Dessloch. Towards an Integrated Model for Data, Metadata, and Operations. In *BTW 2007*, 2007.

[Hec06] Reiko Heckel. Graph Transformation in a Nutshell. *Electr. Notes Theor. Comput. Sci.*, 148(1):187–198, 2006.

[Jö07] Thomas Jörg. Entwicklung eines regelbasierten Editors für Graphmodelle (Development of a Rule-based Editor for Graph Models). Master's thesis, University of Kaiserslautern, February 2007. in German.

[KR90] Kreowski and Rozenberg. On structured graph grammars. 1990.

[MRB03] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *SIGMOD 2003*, 2003.

[SD07] Boris Stumm and Stefan Dessloch. Change Management in Large Information Infrastructures – Representing and Analyzing Arbitrary Metadata. In *BTW 2007*, 2007.

[Stu06] Boris Stumm. Change Management in Large-Scale Enterprise Information Systems. In *EDBT Workshops*, pages 86–96, 2006.

[Tae99] Gabriele Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In *AGTIVE*, pages 481–488, 1999.

[VFV06] Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a Graph Transformation Engine in Relational Databases. *Software and System Modeling*, 5(3):313–341, 2006.

[VSV05] Gergely Varro, Andy Schurr, and Daniel Varro. Benchmarking for Graph Transformation. In *VLHCC 2005*, pages 79–88, Washington, DC, USA, 2005.

[WKR01] Andreas Winter, Bernt Kullbach, and Volker Riediger. An Overview of the GXL Graph Exchange Language. In *Software Visualization*, pages 324–336, 2001.