Adaptable Support for Queries and Transformations for the DRAGOS Graph-Database

Erhard Weinell

RWTH Aachen University of Technology, Department of Computer Science 3,
Ahornstrasse 55, D-52074 Aachen, Germany,
Weinell@cs.rwth-aachen.de

Abstract. The DRAGOS database eases the development of graph-based applications by providing a uniform graph-oriented data storage facility. In this paper, we extend the existing database by a basic Query and Transformation Mechanism, which facilitates the construction of graph transformation systems. Users can therefore access the database by applying structured rules instead of using atomic operations provided before. As result, the development of graph transformation tools is eased by providing a mapping of specific graph languages to the Query and Transformation Language, instead of developing interpreters or code generators. In addition, structured rules offer more optimization potential in the underlying graph storage, which is beneficial for existing graph transformation systems. The presented approach is especially designed for extensibility, so its functionality can be adapted corresponding to the demands of the respective application domain.

1 Introduction

During the past decades, graph transformations have evolved to a mature and well-defined formalism to carry out operations on graph-like data structures. Based on different formal backgrounds, many tools and languages emerged in the community using graph transformations in various application areas. A graph transformation tool (GTT) usually comprises a graph storage facility and a code generator or an interpreter to execute the declarative rules.

Despite all previous standardization efforts, unfortunately, these tools rarely rely on a common basis. Instead, they typically use very different data representation, moreover, both the semantics of their graph transformation language and their execution strategy have differences. As result, developers of GTTs have to implement the required functionality anew for each tool. Furthermore, GTTs are hardly able to interact, e.g. by operating on a common host graph using different specification paradigms.

Graph-oriented database management systems (graph-databases for short) may provide a solution for uniform data representation as they allow to store complex data structures directly in the form of graphs. In contrast, relational

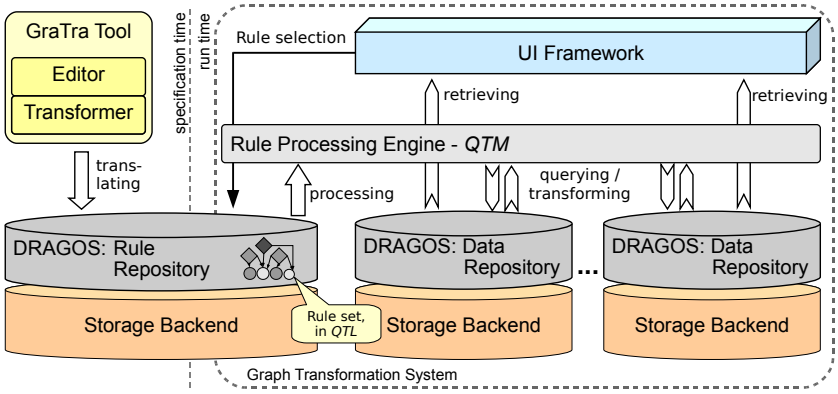


Fig. 1: Applying DRAGOS in graph transformation systems

or object-oriented databases usually require technical helper elements, e.g. additional tables to store many-to-many relations. DRAGOS[1] [1] is the latest representative of this class of databases developed at our department, which supports various back-end databases and model representation formalisms.

However, supporting the interaction between different GTTs is a more difficult challenge, due to semantic mismatches between the languages used by these tools. Exchange formats like GTXL [2] provide a common syntactical representation, but fall short in describing the meaning of a graph transformation rule.

This paper proposes an approach to provide a universal platform which eases the development of GTTs. A coarse-grained overview on the architecture is given in Figure 1. By providing a universal Rule Processing Engine, GTTs can rely on the offered functionality, instead of providing an own code generator or interpreter. The engine is fed by a Rule Repository, which stores the modeled graph transformation system (GTS) in a specialized, low-level graph language. The rule repository actually is a regular DRAGOS instance which stores the GTS in a graph-based form. Graph Transformation Tools *transform* a GTS modeled in their own respective language into the language provided by the processing engine. This transformation step actually maps the semantics of the GTT's language onto the language of the processing engine.

At runtime, the Rule Processing Engine accesses (possibly multiple) DRAGOS instances acting as Data Repository to query and transform the stored graphs. Although, conceptually, this engine acts like a rule interpreter, it might apply code generation techniques internally, depending on its implementation. A UI Framework, which provides a comprehensible representation to the user, selects processing rules from the rule repository, and updates its view structure from the data repositories. Each graph-database uses its own storage backend, which might vary between fast in-memory solutions and relational databases, including

[1] Database Repository for Applications using Graph-Oriented Storage, previously *Gras/GXL*

transaction support. Providing a graph-based interface to existing applications by implementing a corresponding storage wrapper would be feasible as well. Note that the multiple database instances depicted in the figure only illustrate the openness of the framework. They might collapse into a single database in practice. However, the ability to access multiple data storages by a single rule processor is especially useful in the area of data-oriented tool integration, where existing tools are coupled by data translation.

Neither DRAGOS nor its predecessors offer such a processing engine, but only provide atomic retrieval and update operations through an API. Therefore, we currently develop a Query & Transformation Mechanism (QTM) which is able to represent a GTS using a basic Query & Transformation Language (QTL). In contrast to common GTTs, we do not provide a closed system, but strive for an open and extensible architecture to support a wide range of existing graph models and corresponding languages. The QTM yields several advantages from which both existing and newly developed graph transformation tools may benefit:

- Development effort for deriving executable GTS from a declarative specification is reduced, as the "semantic gap" between the specification and the corresponding execution framework is decreased: Instead of generating code based on atomic graph operations, the target domain becomes the DRAGOS QTL, which is considerably closer to a declarative modelling language. Furthermore, the required translation can be conducted based on the abstract syntax graphs of the application-specific graph language and the QTL, essentially making this a matter of model transformation.
- The effort is also reduced for adding additional language constructs to an application-specific graph language, as the QTL is prepared for extensibility itself. Furthermore, such constructs can be transferred to other graph languages by developing an extension of the DRAGOS QTL.
- In case of DRAGOS, evaluating complex queries is significantly more efficient compared to the processing of atomic operations, as conducted by generated code. We will outline this issue towards the end of this paper.
- The set of graph transformation rules can be extended and changed even at runtime of the GTS, thus supporting ad-hoc queries as well.

This paper presents the QTL currently being developed, and discusses its interaction with existing graph languages. In the following Section 2, relevant aspects of the DRAGOS system are introduced. Afterwards, Section 3 introduces the QTL by means of an example. Section 4 then presents the embedding of the QTM into the DRAGOS architecture. Relations to existing approaches are discussed in Section 5 followed by a conclusion in Section 6.

2 DRAGOS architecture and graph model

Unlike its predecessors, DRAGOS does not provide an own graph storage facility. Instead, only a common interface, the so-called core graph model, is defined.

Several implementations of this interface exist, which use existing database management systems as storage facility. Implementations are available for various databases accessible through JDBC and for the Java Data Objects framework. For testing purposes, an in-memory storage is provided. Database-specific implementations initialize connections to the database and perform queries and updates corresponding to the operations invoked on the core model.

Figure 2 shows a coarse-grained overview of the DRAGOS architecture. In the middle, the DRAGOS Kernel encapsulates the core graph model and a set of basic services. The responsibility of services include opening and closing of databases as well as transaction and event management.

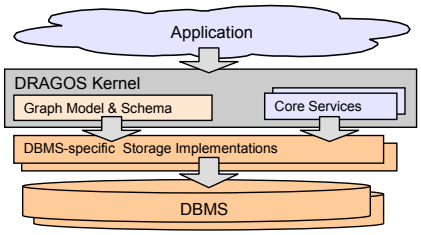
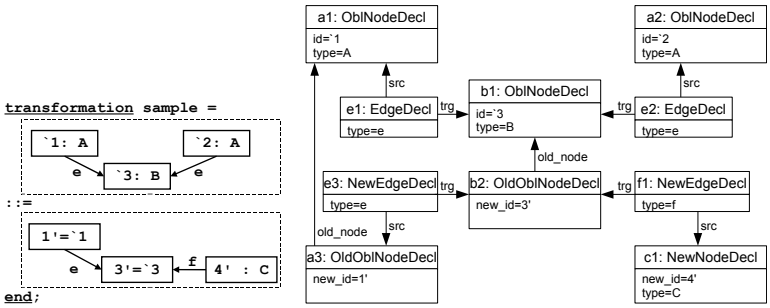


Fig. 2: DRAGOS architecture

Graph model. DRAGOS offers a rich graph model originally inspired by the Graph eXchange Language (GXL) [3]. The model supports hierarchical graphs including graph-crossing connections and n-ary edges (i.e. relations). Nodes, graphs, edges and relations are treated as first-class citizens, and thus can be identified and attributed. This enables flexible connections between entities, e.g. edges connecting edges and the attribution of all entities. All entities need to be typed by some graph entity class. Type hierarchies are supported, including multiple inheritance. As discussed in Section 3, the graph model's overall flexibility however complicates the development of a QTL, as all expressible constructs need to be covered appropriately.

3 Query & Transformation Language

As announced in Section 1, the intended use of the DRAGOS QTL is to provide tool support for existing and novel graph languages. Therefore, the QTL has to be able to cover different graph language approaches appropriately, and to enable an easy translation of the corresponding rules. Furthermore, the QTL should be suited to use the entire DRAGOS graph model, e.g. querying nested graph structures and hyperedges should be supported. This section first examines a transformation rule from an application-specific language, and introduces the QTL by means of this example.



(a) Concrete syntax (b) Abstract syntax

Fig. 3: Example transformation rule in application-specific language

3.1 Application-specific graph language: An example

Figure 3a shows a transformation rule modeled using the PROGRES graph transformation language. A transformation rule describes a graph structure to be found in the runtime graph on its left-hand side (LHS, upper part). In this case, two nodes of type `A` and one node of type `B` connected by edges of type `e` are queried. If this pattern is found in the graph storage, it is transformed corresponding to the rule's right-hand side (RHS, lower part). Here, nodes assigned to `‘1` and `‘3` are preserved, as corresponding variables (`1’` resp. `3’`) are present on the RHS. The node assigned to `‘2` is deleted in the course of the transformation, whereas a new node of type `C` is created and assigned to `4’`. As edges are neither identified nor attributed in the PROGRES graph model, they do not need to be preserved explicitly: Removing all edges of the LHS and inserting edges corresponding to the RHS has the same effect as preserving edges if possible.

Figure 3b shows the same transformation rule represented in the abstract syntax model of PROGRES. Objects of type `OblNodeDecl` and `EdgeDecl` represent entities to be retrieved from the database, with the required type stored in the `type` attribute. The class `OldOblNodeDecl` represents objects on the RHS which retain a node during the transformation. Consequently, they have a `OblNodeDecl` assigned via an `old_node` edge. Nodes assigned to an `OblNodeDecl` without such an edge are deleted during the transformation process. `NewEdgeDecl` represents an edge on the RHS. As explained above, there is no explicit correspondence to edges of the LHS. To process this query by the QTM, it needs to be translated into the QTL.

3.2 Language structure

The Query & Transformation Language should be able to concisely represent rules modeled in an arbitrary application-specific language, and therefore requires a universal basis. We identified the principle of *constraint satisfaction* as such a basis, as it allows a clear distinction between the queried entities, and

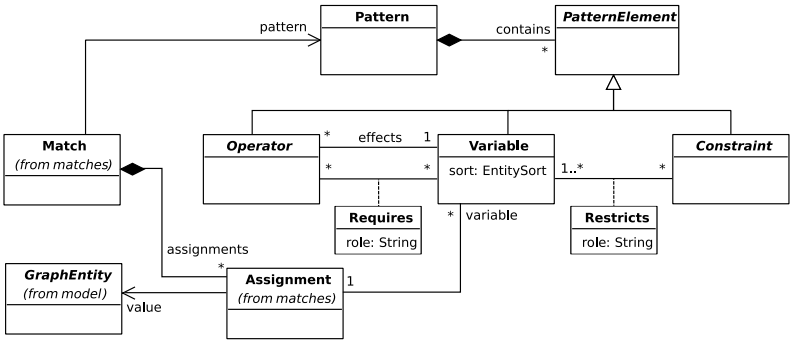


Fig. 4: Core language meta-model (excerpt)

their desired properties. This idea has been introduced previously, e.g. for efficiently implementing pattern matching algorithms [4] or search-plan generation [5]. As result, distinct properties such as containment within a graph or restriction to a specific type can be modeled by attaching constraints to a variable. These properties can be neglected by not adding these constraints, e.g. to query connections of an *arbitrary* type between graph entities.

Figure 4 shows an excerpt of the meta-model of the generic core language. A Pattern contains a set of PatternElements, i.e. Variables, Constraints and Operators. Variables are placeholders for entities found during pattern matching. Constraints restrict legal assignment of entities to Variables. Among others, Constraints restrict entities to a specific type, check connectivity between entities, or containment within a graph. The role attribute of the Restricts association distinguishes between Variables a Constraint restricts. For example, the ContainmentConstraint needs to distinguish between Variables holding the parent graph and the child entity, as both may refer to graphs.

An assignment of graph entities to Variables of a Pattern not violating any of its Constraints is called a Match. Each Match aggregates a set of Assignments, which relate a Variable to exactly one GraphEntity. We require that each Variable with an attached Constraint has to be present in a Match, so partial matches are not allowed. Unconstrained Variables are not bound during pattern matching.

Operators define how entities bound to a Match should be transformed. Each Operator effects exactly one entity assigned to a Variable. To do so, the Operator may Requires values of other variables as parameter. Required variables are distinguished by a role name. Creation and deletion of an entity extend and reduce the match by the effected variable, respectively. Operators can be executed only if all required Variables are bound. This indirectly imposes an order on the operator's execution, as required variables need to be bound in advance.

In general, the execution order of operators might influence the result of the transformation. To avoid ambiguities, we require that variables are either bound to an entity, or a new entity is created by a transformation rule. Furthermore,

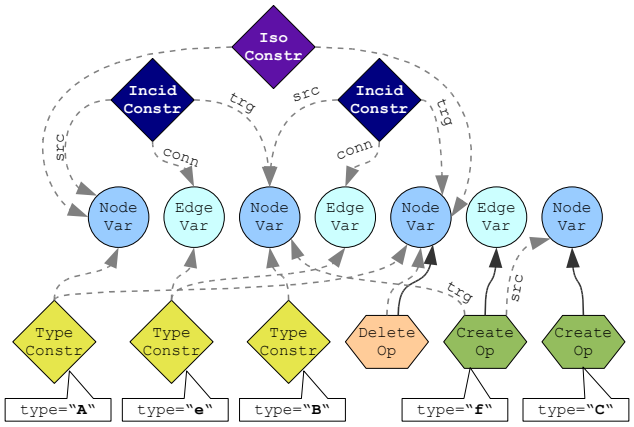


Fig. 5: Example transformation rule in the DRAGOS core language

only entities bound to a variable may be deleted. In addition to these restrictions, all operators must act in a single-step manner, i.e. only results of the pattern matching phase may be taken into account when modifying graph entities. For example, computing an attribute value may only be based on the attribute values *before* the first operator invocation. Therefore, the computed result does not depend on other attribute updates within the same transformation.

Figure 5 shows the transformation rule from Figure 3 after translation to the DRAGOS core language. *Circles* represent Variables for each node or edge queried or created by the transformation rule. *Rhombs* depict constraints which a value of a variable has to fulfill. Dashed lines relate constraints to variables, e.g. the `IncidenceConstraint` puts three variables into relation as `source`, `target` and `connector`. The `IsomorphismConstraint` ensures that different entities are bound to the attached variables. `TypeConstraint`s restrict the variables' values to entities of the type denoted by the `type` attribute. *Hexagons* represent operators which denote modifications of the matched graph entities. Here, the entity assigned to the third `NodeVariable` is deleted and removed from the match. The deletion of an entity also causes the deletion of incident edges and relation ends to prevent dangling connections. Therefore, explicit deletion of the retrieved edge is not required. In contrast, newly created entities are assigned to the two variables on the right, with their type passed via the `type` attribute of the operator. The variable effected by an operator is attached using a solid line, whereas required variables are connected by a dashed line. To create an edge, its `source` and `target` entities are required. The right-most variable is not connected to any constraint, so it is not bound during pattern matching. Hence, creation of the edge is postponed until its designated source node has been created, as the corresponding variable is bound to the new entity afterwards.

This basic representation of the example rule does no longer contain any specifics of the PROGRES language. For example, the implicit condition that

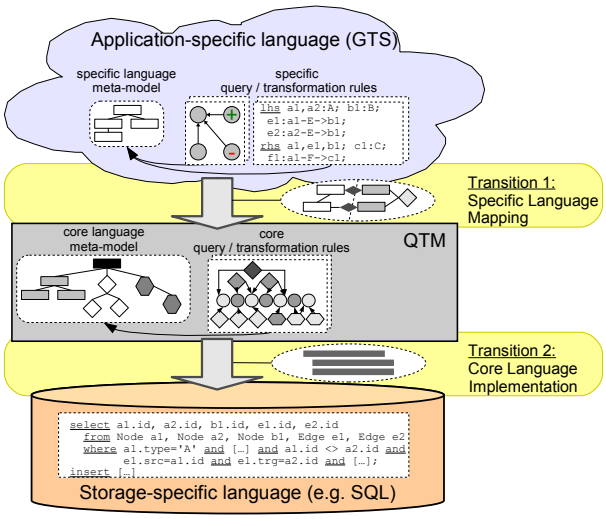


Fig. 6: Application integration with the DRAGOS QTM

distinct entities have to be assigned to the node variables of Figure 3 is explicitly expressed by an `IsomorphismConstraint`. LHS and RHS are condensed into a single pattern, using operator nodes to express actions. Note, that this pattern could be changed easily to query connections between edges instead of nodes, by changing the variables' sorts. Such queries are not possible in common graph transformation languages.

4 Query & Transformation Mechanism

In this section, we clarify how GTS access the Query & Transformation Mechanism and how the QTM is embedded into the DRAGOS architecture.

4.1 Application integration

Figure 6 shows how GTS modeled in an application-specific language interact with the DRAGOS QTM. As mentioned in Section 1, the corresponding rules are converted to the DRAGOS core language. This can be achieved by importing the corresponding rules's ASGs into the graph-database, e.g. by parsing a textual representation. Furthermore, we currently implement an import mechanism to access EMF-based model repositories. The translation results in a graph structure representing a set of QTL rules, which are processed by the QTM's language implementation.

In order to map the application-specific graph language to the QTL, a set of *model transformation rules* translate the increments of the GTS' rules to patterns of the QTL. By this translation, the respective language's semantics are

mapped to the core language. For example, the two graph transformation languages PROGRES and GROOVE treat rules differently regarding isomorphism: In PROGRES, each node may be bound to only one variable during pattern matching, but allows exceptions using *folding groups*. The default treatment is the other way around in GROOVE, where exceptions are specified using *merge embargo* edges. To cover the former case, an isomorphism constraint is connected to all variables whose values should be pairwise disjoint. For the latter, isomorphism constraints are simply added between those variable connected by a merge embargo edge.

As the model transformation process operates on the graph-database exclusively, its corresponding rules can again be specified using the QTL. Therefore, a convenient representation of these rules is desirable, which can be achieved by an own application-specific language. Its purpose is to allow a convenient development of application-specific language mappings. Such an integration language is currently being developed, which naturally profits from existing work in the field of model transformations.

4.2 Embedding into the DRAGOS architecture

The fact that DRAGOS can utilize relational databases as storage backend suggests that the QTM should use their sophisticated query functionality to enable efficient execution. Regarding SQL databases, rules modeled in the QTL can be translated into corresponding SQL queries and update operations, as indicated in the lower part of Figure 6. This transition is currently implemented based on templates. Basically, they create increments in the `FROM` part of the SQL query for each variable, and increments in the `WHERE` part for each constraint.

As not all available backend implementations provide database-like functionality, a backend-independent solution is required, too. As the DRAGOS graph model is the common interface of all available implementations, we provide a rule interpreter based on this model. This interpreter is called the Generic Implementation in Figure 7, where the architecture of the QTM is depicted. Gray arrows indicate the approach of processing QTL rules using the Generic Implementation, White arrows show the processing by a backend-specific language implementations, such as by deriving SQL code. Although the latter choice allows a more efficient processing, it can only be used if a corresponding language implementation is available for the applied DRAGOS backend. The Controller Service therefore selects the appropriate processing path at runtime.

Although it might appear questionable why not all existing backend implementations can be augmented with an appropriate QTL implementation, this approach is especially necessary for adding language extensions. Also note that the chosen processing step, e.g. rule interpretation or query generation, is completely independent of the actual application-specific language. Therefore, the two transitions shown in Figure 6 can be combined arbitrarily. In this sense, the QTL acts as an interface separating backend functionality from the application-side GTS.

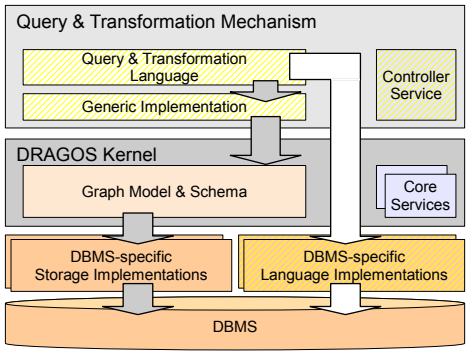


Fig. 7: Extended DRAGOS architecture

4.3 Adding language extensions

The mapping of rules from an application-specific language to QTL usually cannot be represented directly, causing the required mapping to become complex and hard to read. Although this might be acceptable for supporting a single graph language, the developed mapping cannot be re-used for other application-specific languages. Therefore, we offer to *extend* the core language by additional language constructs by defining new constraint classes or operator classes. However, the core graph model cannot be extended by additional variable classes, as this would require adaptations to existing constraints and operators. In contrast, constraints and operators only refer to their attached variables, so new types of these elements can be added without effecting existing ones.

Basically, we offer two options to implement added constraint and operator classes: *First*, an extension of the existing backend-specific implementation of the QTL can be provided, e.g. by generating fragments of SQL code. This choice generally yields the more efficient implementation, as backend functionality can be utilized directly. Regarding the architecture in Figure 7, another column is added which allows to bypass the basic QTL and the DRAGOS graph model. However, this approach would require to implement each language extension for all existing storage backends. Even worse, supporting additional storage backends would, the other way around, require to provide variants for all QTL extensions. Obviously, this tight interrelation is not desirable.

As solution for this dilemma, we provide a *second* option for implementing language extensions, which is split into two variants: A QTL extension should *additionally* be implemented by extending the existing Generic Implementation, thus combining the two approaches of Figure 7: If the utilized storage backend does not provide a corresponding extension implementation, the extension's Generic Implementation can always be used. Nevertheless, the backend-specific QTL implementation can be used to process the remaining parts of the query.

In addition, extended language constructs can be *reduced* to basic QTL constructs by providing a corresponding transformation rule. Using this approach, constructs can be implemented in a transformation-based way, in contrast to the programmed approach discussed above. The strictly graph-based representation of query and transformation rules provide a sound basis for such a high-level extension mechanism. As we cannot introduce this aspect of the QTL in detail here, the reader is referred to [6], where the model-based extension mechanism is elaborated in detail.

4.4 Experimental evaluation

To conclude the presentation of the QTM, we present initial performance comparison against DRAGOS and its predecessor, GRAS [7]. The original GRAS (GRAph Storage) database had been developed at our department since the late eighties. A replacement of the GRAS system became necessary because its tight platform dependency and severe restrictions on both the number of manageable graph entities and the total amount of stored attribute values. As DRAGOS relies on existing solutions like relational databases as graph storage, the amount of manageable data is only limited by the underlying storage backend. This limit is sufficiently high in common relational databases, e.g. PostgreSQL restricts tables to 32 terabytes.

The obvious disadvantage of the DRAGOS approach is the immanent performance penalty, as each atomic graph operation is implemented by at least one SQL statement. Comparisons of real-world examples indicate a factor of around $1 : 12$ relating GRAS to DRAGOS operating only on the in-memory storage, which does not provide transactions support. Using PostgreSQL as transactional storage backend, the factor increases up to $1 : 120$. The reason for this massive overhead is that the DRAGOS architecture does not allow the adequate use of complex query mechanisms, such as SQL. In fact, only a limited amount of simple statements are used to query the storage facility.

The QTM introduced in this paper, however relieves this architectural disadvantage, as complex graph patterns can be transformed into backend-specific queries. Initial experiments underline this thesis, as shown in Figure 8: We measured the required time to test for circles of a given size (3 resp. 10) in an n-complete graph, comparing the SQL code generated from the QTM, DRAGOS operating on PostgreSQL, and GRAS. All tests were run on a 3 GHz Intel CPU and 2 GB of main memory, showing the median of several test runs.

We can conclude that GRAS outperforms DRAGOS and the QTM for small queries, although results become less clear for larger ones. Interestingly, although DRAGOS and GRAS are controlled by comparably generated (and optimized) code, time consumption behaves differently for these systems: For small queries, the required time remains almost constant with larger graphs for GRAS, whereas it noticeably increases for DRAGOS. The QTM approach performs quite good, given its early stage of development. For larger queries, the PostgreSQL query optimizer does not seem to derive optimal search plans, which needs to be addressed to improve the QTM performance. It should be noted that the specific

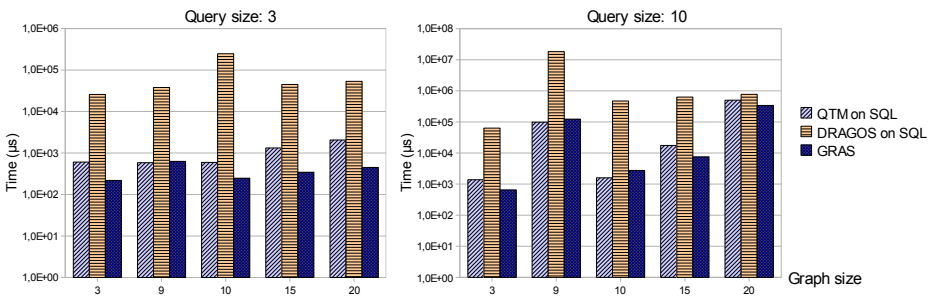


Fig. 8: Timings for querying n-complete graphs (microseconds, logarithmic scale)

results of GRAS varied significantly for the large query, so the overall result is less reliable.

Unfortunately, the generation of SQL-code from the QTL is not fully implemented up to now, so that standardized comparisons, e.g. presented in [8], could not be performed. Once the implementation reaches a proper level, we will also evaluate the performance impact on real-world scenarios, such as model transformations.

5 Related Work

This section covers comparisons to existing work, considering model repositories, graph transformations, and related optimization techniques.

Model repositories. Besides being designed as graph-oriented database, DRAGOS can be considered a model repository or data binding tool as well. Both aspects are covered by a wealth of existing standards and tools. An example for such a tool is the Universal Data Model (UDM) [9]. In contrast to DRAGOS, which uses a complex core graph model for data representation, UDM relies on a limited set of base classes. However, UDM is able to generate APIs from provided metadata, such as UML class diagrams, to allow convenient and type-safe use by developers. Such functionality is currently not provided for DRAGOS, although code-generating graph transformation tools can be applied for this purpose. Similar to DRAGOS, the UDM environment provides persistent storage using databases through the Generic Modeling Environment (GME) [10]. UDM does not incorporate a model processing engine, but can be used by the GReAT transformation engine [11].

Graph transformations on relational databases. Implementing GTS on established relational databases has been presented initially by the authors of [12]. Basically, the authors transform a graph schema to a set of database relations, and implement pattern matching by deriving views on these tables. One difference to our approach is the applied meta-level (M 1), as the DRAGOS graph

model constitutes a common meta model for all applications (thus M 2). Furthermore, we apply the basic idea of generating SQL code in a language-independent environment, with the QTL forming a common basis. The separation between variables, constraints, and operators applied in the QTL is indeed closer to the SQL than traditional graph languages considered by [12], which simplifies the translation process for us.

The authors also mention a specialized query optimizer developed for the applied relational databases, which, unfortunately, is not discussed any further. We agree that an optimizer specialized on graph queries is indeed necessary. Inspection of the internal search plans of the database backend showed that the standard optimizer already prefers table joins with small result sets, e.g. traversing edges instead of global searches over the graph. However, the order of edge traversal is not optimized effectively, which causes inefficient behavior for the larger query discussed above. To relieve this drawback, a specialized query optimizer should adapt results from search-plan driven code generation found in common graph transformation tools.

Optimization techniques. Traditionally, code generated for graph transformation rules is optimized using search plan techniques to find an efficient order of variable assignments. Among others, PROGRES and Fujaba apply this technique in a *static* environment, i.e. code is generated once before the system is run. This only allows to optimize according to the graph schema, as the actual host graph structure is unknown during code generation. Recently, the authors of [13] proposed an approach to generate differently generated variants of code. Depending on the host graph, an appropriate variant is chosen at runtime.

Our approach is not tied to a code generation step, as transformation rules are stored in the database and executed in a backend-specific way. Storage solutions may decide whether to interpret these rules (e.g. the generic implementation follows this approach) or to generate implementation-specific code (e.g. SQL statements). Nevertheless, our approach still requires search-plan based optimization techniques, as common SQL query optimizers do not recognize incident structures.

Graph transformations based on constraint satisfaction. The DRAGOS Query & Transformation Language is based on the theory of *constraint satisfaction problems* (CSP) known from the area of artificial intelligence. CSPs are well-suited to model graph pattern matching by solving the *subgraph-isomorphism problem* [14]. In our work, we aim to implement the Query & Transformation Language based on existing systems, and therefore extensive development of a basic constraint solver is not of crucial importance. This would only improve the generic implementation based on the core graph model, which should be considered as fallback solution only. Instead, we focus on implementations based on sophisticated storage backends like databases.

CSP-like representations of graph transformation rules have also been applied in [5], where search-plan optimization is discussed for such a rule model. As this approach is not concerned with the evaluation of expressions, dynamic

aspects such as matches need not to be considered. In contrast, our approach also incorporates matches to model the result of a query. Furthermore, the cited work includes negative application conditions directly into the language, which are treated differently in the QTL [6].

Model transformations. As mentioned in Section 4.1, model transformations are used to map an application-specific language to the DRAGOS Query & Transformation Language. A wide range of languages and tools for model transformations have already been presented in the literature, several of which are compared in [15]. Available approaches differ with regard to expressiveness, concrete representation (textual vs. visual), usage (batch vs. user-interactive), traceability, and directness (uni- vs. bidirectional).

Currently, we investigate in how far existing solutions can be applied in the DRAGOS system. For our purposes, a very simple batch-oriented uni-directional system suffices. However, we did not collect experiences on the required language constructs yet. In the future, commonly required constructs will form a language specifically tailored for modeling language mappings.

Graph transformations for visual programming. Graph transformation languages like PROGRES and Fujaba provide functionality similar to the presented QTL. In fact, both systems can already generate code to store the runtime data persistently using DRAGOS. As discussed in Section 1, this approach leads to inefficient applications because it is only based on atomic operations. In our approach, DRAGOS executes transformation rules itself, either by interpreting or backend-specific code generation.

Projects at our department recently encountered the need for more advanced language constructs in the PROGRES language. However, extending PROGRES not only requires a visual representation, but also an enhanced code generation. Shortcomings in the architecture of the PROGRES environment caused the authors of [16] to embed new language constructs in a pre-processing phase generating the actual PROGRES specification. Again, executing transformation rules inside DRAGOS and embedding language extension therein would lead to a more concise application development.

In contrast to common graph transformation languages, the DRAGOS core language is not feasible for direct use by a specificator. Due to the very low level of abstraction, even simple queries tend to become quite large and hard to read. Therefore, the presented language should not be considered as competitor to existing languages, but as a common core for existing and new ones to build on.

Unified graph languages. Besides the *GRAph and Rule CEntered specification language* (GRACE) [17] and the *Graph Transformation eXchange Language* (GTXL) [2], little work can be found on providing a common platform for graph transformations. Probably this is caused by the need to establish a standardized unified language, and persuade tool developers to achieve compliance. We therefore do not try to establish such a standard, but offer a flexible and extensible

base layer. Graph languages are integrated by transforming the rules' ASGs, which allows to use concepts independent from the QTL.

6 Conclusion

In this paper, we motivated the need for a QTM in the graph-oriented database DRAGOS. Benefits gained from this mechanism comprise easier use by application developers, a high-level integration of application-specific languages, and a more efficient execution.

The newly developed language is especially designed for extensibility regarding two aspects: First, the core language can be extended by adding new language constructs in the form of constraints or operators. Second, the implementation of the core language may use storage-specific functionality for selected language constructs, and refer to a generic implementation otherwise. Therefore, the language's implementation is extensible, too.

Current & Future Work

The core language's expressiveness is currently restricted to graph patterns with a fixed size, hindering its use e.g. for matching path expressions with the Kleene star operator. We therefore offer to dynamically expand a pattern using a template-based mechanism, allowing for *recursive queries*.

In order to model large-scale graph transformation systems, the interaction between different transformation rules needs to be captured. For this purpose, the introduction of *control structures* is necessary, e.g. for iterations or conditional branching. Currently, a minimal amount of control structures is being integrated into the core language to support arbitrary graph transformation systems. This is inspired by the results of [18], where the authors show that a very small set of structures suffices. Extended control structures can be supported by a language extension or by a proper mapping of application-specific languages to the core QTL.

In the future, we will investigate how existing approaches to graph transformations can be mapped to the DRAGOS language, such as the algebraic approach or hyper-edge replacement grammars. This way, DRAGOS can serve as a platform to develop new constructs for graph transformation languages by offering a high-level extension mechanism.

References

1. Böhlen, B.: Specific graph models and their mappings to a common model. Volume 3062 of LNCS., Springer (2004) 45–60
2. Lambers, L.: A new version of GTXL. In: Graph-Based Tools (GraBaTs 2004). Volume 127 of Elec. Notes in Theoretical Comp. Sci., Elsevier Science (2004)
3. Holt, R., Winter, A., Schürr, A.: GXL: Towards a standard exchange format. In: Proc. of the 7^{th} Working Conference on Reverse Engineering (WCRE), IEEE Computer Society Press (2000) 162–171

4. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. [19] 238–251
5. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In Ehrig, K., Giese, H., eds.: Graph Transformation and Visual Modeling Techniques. Volume 6 of ECEASST. (2007) 57–68
6. Weinell, E.: Extending graph query languages by reduction. In: Proc. of the 7^{th} Intl. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT). (2008) to appear.
7. Lewerentz, C., Schürr, A.: GRAS, a management system for graph-like documents. In: Proc. of the 3rd International Conference on Data and Knowledge Bases, Morgan Kaufmann (1988) 19–31
8. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE (2005) 79–88
9. Magyari, E., et al.: UDM: An infrastructure for implementing Domain-Specific Modeling Languages. In: 3^{rd} OOPSLA Workshop on Domain-Specific Modeling. (2003)
10. Davis, J.: GME: the generic modeling environment. In: Companion of the 18^{th} annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA), New York, NY, USA, ACM (2003) 82–83
11. Agrawal, A., et al.: The design of a language for model transformations. Software and Systems Modeling **5**(3) (September 2006) 261–288
12. Varró, G., Friedl, K., Varró, D.: Implementing a graph transformation engine in relational databases. Journal on Software and Systems Modeling **5**(3) (2006) 313–341
13. Varró, G., Friedl, K., Varró, D.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. ENTCS **152** (2006) 191–205
14. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical Structures in Computer Science **12**(4) (2002) 403–422
15. Taentzer, G., et al.: Model Transformation by Graph Transformation: A Comparative Study. In: Proc. of the Intl. Workshop on Model Transformations in Practice (MTiP'05). (2005)
16. Fuss, C., Tuttlies, V.: Simulating set-valued transformations with algorithmic graph transformation languages. In: this volume. (2008)
17. Kreowski, H.J., Busatto, G., Kuske, S.: GRACE as a unifying approach to graph-transformation-based specification. In Ehrig, H., Ermel, C., Padberg, J., eds.: UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques. Volume 44 of ENTCS., Elsevier Science (2001)
18. Habel, A., Plump, D.: A core language for graph transformation. In: Proc. of the APPLIGRAPH Workshop on Applied Graph Transformation. (2002) 187–199
19. Ehrig, H., et al., eds.: Theory and Application of Graph Transformations, 6^{th} Intl. Workshop (TAGT). In Ehrig, H., et al., eds.: Theory and Application of Graph Transformations, 6^{th} Intl. Workshop (TAGT). Volume 1764 of LNCS., Springer (2000)