Extending Graph Query Languages by Reduction

Erhard Weinell

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

Abstract: Graph grammars have a long history in visual programming dating back to the seventies. Due to their declarative nature, even complex systems can be captured by clear and concise specifications. Recently, with the OMG's standard for model transformations QVT, graph grammar concepts have also found their way into an industrial scale. Although numerous languages and tools for graph transformations exist, they have no technical basis such as an execution framework in common. Instead, graph transformation machineries are usually implemented anew for each of these tools.

The DRAGOS graph database is especially well-suited for building graph transformation systems, as it is able to store complex graph structures directly. Besides its storage functionality, the database also provides a Query & Transformation Mechanism which is able to handle complex queries upon the stored graphs, and to modify them accordingly. Being designed as a basis for graph and model transformation tools, this mechanism is required to allow a flexible adaptation and extension according to the respective applications' needs. The present paper discusses how this requirement is covered by the proposed Query & Transformation Mechanism.

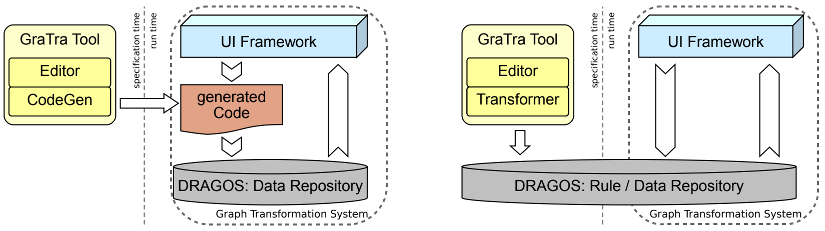
Keywords: graph database, extensibility, constraint satisfaction

1 Introduction

Model transformations are an enabling technique for model-driven software engineering, as they allow the formal definition of automated model translations. For example, model transformations can be used to enrich generic models by platform-specific informations, or to define refactoring rules at model level. The sheer amount of recent publications on the use of *graph transformations* for these purposes indicates their well-suitedness for this task. Presumably, this is caused by the fact that they rely on a mature and formally defined background with proper tool support. Nevertheless, the all-encompassing graph (or model) transformation language does not seem to exist, as new ones are proposed regularly. All of them share a common requirement: A proper data repository to store graph structures persistently, and an according execution framework to carry out the specified transformation rules.

Applications specified using graph transformation languages are called *graph transformation systems* (GTS). These system often utilize memory-based solutions as graph storage, which provide a direct access to the stored data. However, large-scaled applications usually require additional functionality, such as persistent transactional storage (instead of dedicated save actions).

Furthermore, support for concurrency isolation, and the ability to store large graph structures which are too unhandy for continuous transfer between file and memory, come into play. DRAGOS, a graph-oriented database management system (graph-database for short), is especially designed for this purpose. In contrast to traditional databases, DRAGOS provides a data model based on graphs. Therefore, graph structures can be stored directly, without any need for technical helper elements, such as tables for n-to-m relations.



(a) Generating source code (b) Interpreting rules

Figure 1: Applying DRAGOS in graph transformation systems

For implementing graph transformation systems, basically two alternatives exist (as shown in Figure 1). *First*, Figure 1a depicts an approach based on *generating* source code from the graph transformation rules. This code invokes operations on the graph database to retrieve and manipulate individual entities. *Second*, transformation rules can be executed directly by the database, as indicated by Figure 1b. As the figure suggests, the corresponding UI framework does not need to incorporate any generated code, but only relies on the functionality provided by the database. However, the database has to be able of interpreting the respective graph transformation language. This is currently not supported by the DRAGOS graph database, for which reason we develop an according mechanism. As discussed in [Wei07], this solution provides an easier integration with graph transformation tools and a larger optimization potential. To subsume the second argumentation, the code generation approach is less suitable for GTS based on databases. Due to the fact that operations utilized by the generated code are usually situated on a very low level of abstraction, they cannot consider the database-internal specifics. As result, a lot of simple operations are invoked, whereas few complex operations would cause less overhead.

In order to support not only a single, but arbitrary graph transformation approaches and their respective languages, DRAGOS only offers a *basic*, yet *extensible* core language. High-level languages are used to provide a user-friendly *concrete syntax*, which is not provided by the core language. The integration of these languages is basically performed as follows: First, rule definitions of the high-level language are *imported* into the database, e.g. by parsing textual specifications. Graph transformations then *convert* the imported rules to the DRAGOS Query & Transformation Language. The resulting graphs are *evaluated* by the underlying rule processor at runtime. Thus, application integration is achieved through graph transformations, instead of providing a complex code generation module as required in Figure 1a.

The conversion of graph transformation rules is usually complicated by complex mappings of high-level language constructs to the low-level ones provided by DRAGOS. We therefore allow to *extend* the core language by additional constructs to yield a more concise conversion. Likewise, the extended language constructs can be re-used for integrating other graph transformation languages, which is not possible for conversion rules. In this paper, we present the DRAGOS Query & Transformation Language and, as novel contribution, show how the core language can be extended. This is achieved by *reducing* new language constructs to existing ones.

The rest of this paper is structured as follows: We first introduce the basic functionality of DRAGOS in Section 2, and afterwards the Query & Transformation Language in Section 3. The following Section 4 presents how the language can be extended by additional language constructs. The paper finally discusses relations to other projects in Section 5 and gives an outlook on future work in Section 6.

2 Graph database DRAGOS

The DRAGOS database[1] allows to store and retrieve graph structures. Its data model is based on graphs, which are able to capture even complex data structures without need to introduce technical helper elements. For example, the relational data model often requires additional elements, such as extra tables to store many-to-many relations.

Architecture. 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 services' responsibilities include opening and closing of databases as well as transaction and event management.

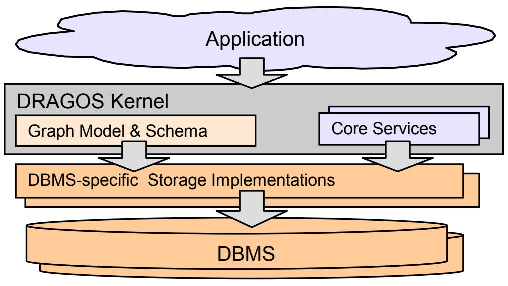
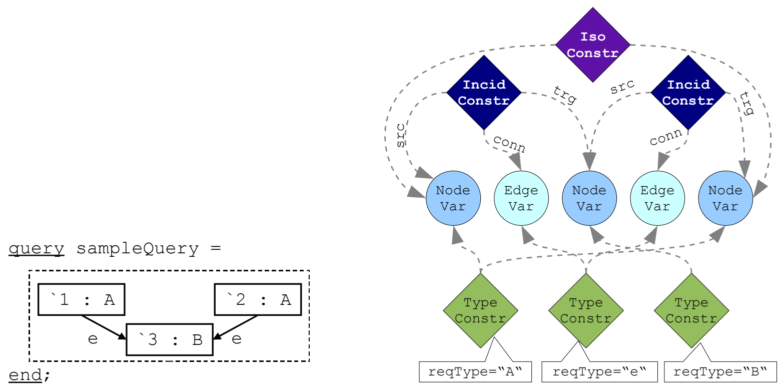


Figure 2: DRAGOS architecture

DRAGOS does not implement an own graph storage module. Instead, several implementations of the core graph model exist, which utilize 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 according to the operations invoked on the core model.

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



(a) PROGRES Query (b) DRAGOS Query

Figure 3: Sample query searching three connected nodes

Graph model. DRAGOS offers a rich graph model originally inspired by the Graph eXchange Language (GXL) [HWS00]. Among other things, DRAGOS supports hierarchical graphs including graph-crossing connections. 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 structuring is supported, including multiple inheritance.

3 Queries & Transformations for DRAGOS

In this section, we present the Query & Transformation Language by means of an example, relating it to the well-known graph transformation language PROGRES [SWZ99]. The language's abstract syntax is presented and its semantics are sketched. Unfortunately, no comprehensive definition of the DRAGOS Query & Transformation Language can be given here due to the lack of space. Also, only the *query* aspect of the language is handled in this paper. For the transformation of graphs, the reader is referred to [Wei07].

3.1 Introductory example

Figure 3a shows a simple visual query modeled using the PROGRES graph transformation language. This query checks whether three nodes connected by edges of proper type and direction exist in the host graph. Another (intuitive, but rather implicit) condition is that indeed two different nodes ‘1 resp. ‘2 exist.

As the DRAGOS graph model is a lot more complex than the PROGRES model, queries according to the PROGRES syntax would be hard to represent. Therefore, the Query & Transformation Language separates between graph entities to be *searched* from the conditions that need to be *fulfilled* by these entities. The DRAGOS query shown in Figure 3b contains a set of *variables* (middle row, depicted as circles). In order to confirm the query, each of these variables has to be bound to a graph entity from the host graph, otherwise the query fails.

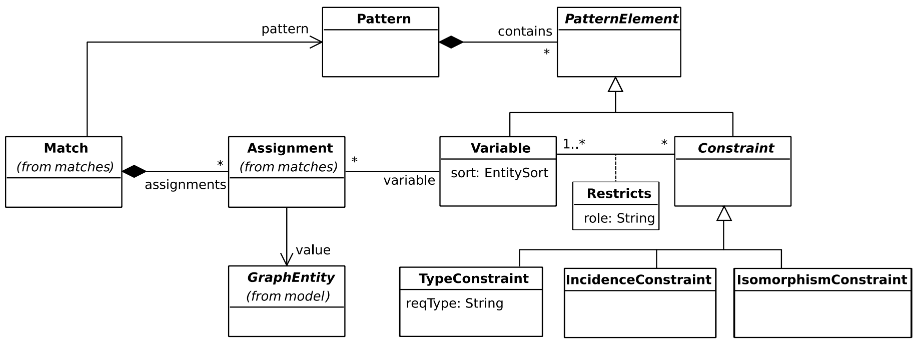


Figure 4: Meta-Model of the Query & Transformation Language (simplified)

Constraints (depicted as diamonds in Figure 3b) are used to restrict the queries' results in several ways: IncidenceConstraints demand connectivity of entities, using role names to distinguish between variables for the source, the target and the connector. This distinction is necessary as DRAGOS allows edges to be connected to other edges, and so querying these structures needs to be supported. TypeConstraints restricts legal values to a certain type, where the desired type is indicated by the reqType attribute. The IsomorphismConstraint is used to ensure that attached variables are bound to *pairwise different* entities. Its name stems from the theoretical concept of searching an isomorphic mapping of queried entities to host graph entities, although it could be called NonIdentityConstraint as well. It is only added between variables of the same type, as inheritance is not considered in the current example.

3.2 Syntax & Semantics

The language's abstract syntax is depicted in Figure 4 by means of its meta-model. According to this model, each Pattern consists of a set of PatternElements, which are sub-divided into Variables and Constraints. Constraints are connected to at least one Variable via Restricts edges, which can be distinguished using the role attribute. To support manipulation of graphs, the complete meta-model additionally provides Operators, which are not discussed in this paper.

Figure 4 only defines the basic structure of patterns, but does neither define static semantics (e.g. well-formedness of patterns) nor dynamic semantics (the actual meaning of the pattern). Here, these two kinds of semantics are introduced for a small subset of the Query & Transformation Language. We utilize the OMG's Object Constraint Language (OCL), as it allows to combine first-order predicate formulae with object-oriented concepts. Nevertheless, it should be noted that the OCL has not been comprehensively defined in a formal way, so that no unique interpretation of the presented formulae can be given. However, several research activities [BW02] strive to define the OCL's semantics, which would lead to an unambiguous understanding.

Besides the language's meta-model depicted above, several well-formedness conditions for patterns exist, which cannot be expressed using class-diagrams in a convenient way. For example, the following OCL invariant defines conditions on the IncidenceConstraint:

```
context IncidenceConstraint
  def: 𝒮: Collection(Variable) = self.restricts→select(r | r.role = "src")
  def: 𝒯: Collection(Variable) = self.restricts→select(r | r.role = "trg")
  def: 𝒞: Collection(Variable) = self.restricts→select(r | r.role = "conn")

  inv: wellformedness =
    self.𝒞→size() = 1 and self.𝒞.sort = VariableSort.EDGE and
    self.𝒮→size() ≤ 1 and
    self.𝒯→size() ≤ 1 and
    1 ≤ self.𝒮→size() + self.𝒯→size()
```

This invariant requires that the constraint is connected to exactly one Variable via a Restricts edge with role conn (connector). This variable has to specify the meta-class EDGE, i.e. it must query edges from the database. In addition, either a unique source variable (role src), or an unique target variable (role trg), or both, have to be given.

An assignment of graph entities to a Pattern's Variables not violating any Constraints is called a Match. Matches are instantiated by the language implementation according to the given Pattern and the contents of the graph database. As specified by the class diagram, each Match holds a (possibly empty) set of Assignments, each of which points to a Variable and its corresponding value. In addition, Matches have to comply to the following invariants.

```
context Assignment
  inv: validity =
    (self.variable.sort = VariableSort.NODE implies self.value.oclIsTypeOf(Node)) and
    (self.variable.sort = VariableSort.EDGE implies self.value.oclIsTypeOf(Edge)) and
        [...]
```

The *validity* invariant requires that each Assignment relates Variables to *proper* entities in the database. Therefore, i.e. a Variable of sort EDGE may only be related to an Edge in the database.

```
context Match
  inv: completeness =
    let 𝒱: Collection(Variable) =
      self.pattern.contains→select(oclIsKindOf(Constraint))→collect(c | c.variable)
    in self.assignments→collect(a | a.variable)→includesAll(𝒱)

  inv: uniqueness =
    self.assignments→forAll(a₁ | self.assignments→forAll(a₂ |
      a₁.variable = a₂.variable implies a₁ = a₂))

  inv: correctness =
    self.pattern.contains→select(oclIsKindOf(Constraint))→forAll(c | c.fulFilled(self))
```

Besides the Assignments' validity, a Match has to be complete, unique, and correct.

- For *completeness*, an Assignment has to exist for all Variables which are referred to by any of the Pattern's Constraints. Hence, all restricted Variables must have a value assigned.

- The *uniqueness* invariant demands that each Match holds at most one Assignment for each Variable. This restriction eases the definition of Constraints.

- *correctness* means that a Match fulfills every Constraint of its Pattern. Fulfilledness is defined depending on the respective Constraint's type (see below).

```
context TypeConstraint
  def: fulFilled(m: Match): Boolean =
    self.variable→
      forAll(v | m.assignments→select(a | v = a.variable).value.type = self.reqType)

context IncidenceConstraint
  def: fulFilled(m: Match): Boolean =
    let c = m.assignments→select(a | 𝒞 = a.variable).value
    in (𝒮 →isEmpty() or m.assignments→select(a | 𝒮 = a.variable).value = c.source)
      and (𝒯 →isEmpty() or m.assignments→select(a | 𝒯 = a.variable).value = c.target)
```

A TypeConstraint is fulfilled iff the values of all attached variables are of the type demanded by its reqType attribute. This definition does not consider any type hierarchy. The IncidenceConstraint demands that the edge assigned to the connector variable (the singleton collection $\mathscr{C}$) is the source resp. the target of the corresponding variables. This restriction only applies if an according variable is connected to the constraint.

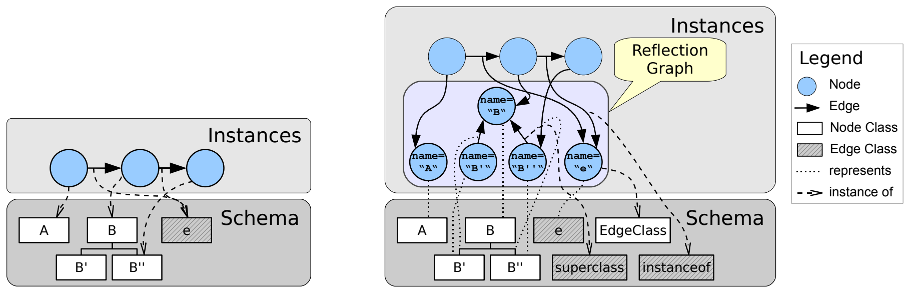
The presented invariants (partially) define the validity of Matches, but do not state how such an assignment can be computed. Language implementations therefore need to provide an operational implementation of these invariants.

4 Extending the Query & Transformation language

The core language defined in the previous section allows to model queries using a basic set of language constructs. This section introduces a technique to add additional constructs to the language, e.g. to represent special semantics of a high-level language. As example, the TypeConstraint mentioned above is extended to support *type inheritance*. This is achieved by adding an additional constraint to the language's meta-model, and by reducing its intended semantics to those of existing constraints.

4.1 Type-level reasoning

The reduction of constraints usually requires to reason on the entities' types and their relations. For this purpose, we added a mechanism which *reflects* the database graph schema into the runtime graph, as shown in Figure 5. On the left side (Figure 5a), the standard situation using separate instance and schema models is shown. Dashed arrows indicate an entities' type. However, the Query & Transformation Mechanism is not able to traverse this relation or examine the entities' types. Therefore, Figure 5b reflects the graph schema into the runtime data as special Reflection Graph. Node classes and edge classes are represented by nodes in this graph, with attributes storing the types' names. Edges model the inheritance relations. Additional edges connect entities of the regular instance graph to nodes representing their types in the Reflection Graph. The Query & Transformation Mechanism can therefore traverse and analyze this graph in the same way as regular instance graphs are handled. For the sake of clarity, some represents and instance of lines are omitted in the figure.



(a) Standard instance graph & schema

(b) Explicit type relation using reflection

Figure 5: Reflection graph to query types

4.2 Basic constraint reduction

To revive the initial example, Figure 6 (left side) shows an additional SubTypeConstraint used to check an entities type compatibility. Just like the regular type constraint, it receives the type's name by the reqType attribute. In order to evaluate this constraint based on the core language, it is *reduced* to the query on the right. Node variable ① corresponds to the original variable. An IncidenceConstraint is used to traverse the instanceof relation, as checked by the TypeConstraint of edge variable ②. The value of ③ is a node in the reflection graph representing the entities class.

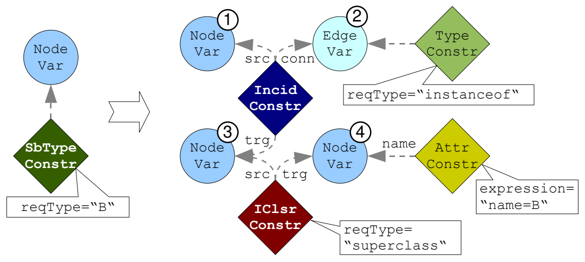


Figure 6: Definition of the SubTypeConstraint

From variable ③, a so-called IncidenceClosureConstraint traverses an arbitrary (including zero) number of edges, just like the Kleene star operator does for regular expressions. In contrast to the IncidenceConstraint, this constraint is not connected to any edge variable, as an unknown number is traversed during pattern matching. To restrict the traversed edges to a certain type, the IncidenceClosureConstraint expects an edge class passed as value of the reqType attribute. In this case, the type superclass is given, whose instances model inheritance relations in the reflection graph. According to this relation, entities assigned to the target variable ④ are again nodes of the reflection graph representing classes. Another AttributeConstraint checks the respective class name, only retaining the class named B as valid assignment.

As result of this transition, the SubTypeConstraint is fulfilled iff the pattern on the right of Figure 6 is fulfilled. For variable ①, the value's type ③ is retrieved, and all reachable supertypes are checked whether they carry the requested name B. Variables ③ and ④ may get the same entity assigned, so the case that the value of ① is an instance of class B is covered, too. Furthermore, the reachability check also supports multiple inheritance offered by DRAGOS.

The replacement shown in Figure 6 can be expressed easily by a graph transformation rule. This replacement rule is run in a pre-processing phase before invoking the resulting query.

4.3 Nested pattern matching

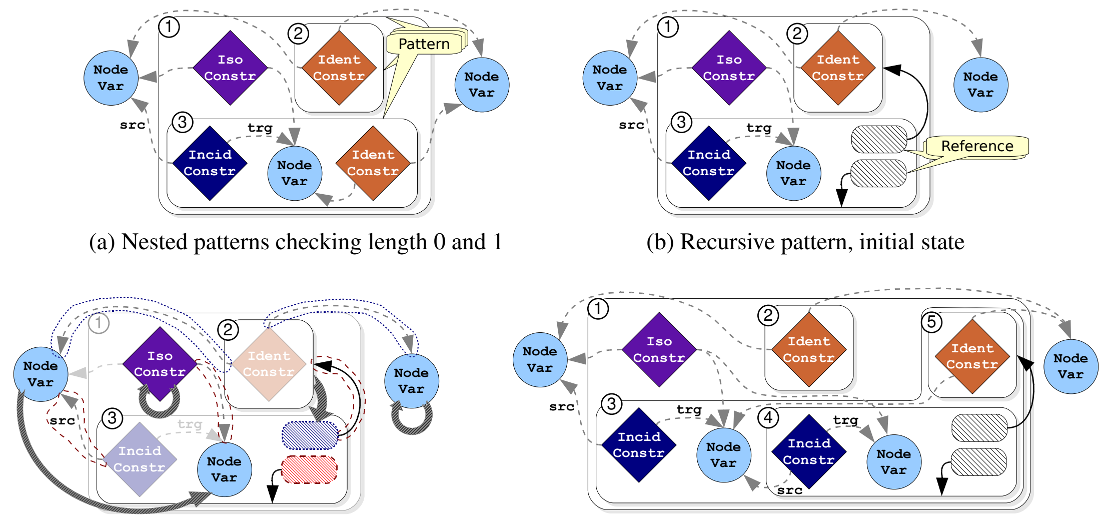
The previous subsection demonstrated a simple conversion rule to replace extended language constructs by basic ones. The DRAGOS Query & Transformation Language additionally allows to replace parts of a rule *recursively*, which is necessary to define the IncidenceClosureConstraint used above. Our approach for recursive replacements is based on the idea of *nested queries*, which is presented in the following.

On the syntactic level, the Query & Transformation Language meta-model is extended by adding Pattern to the subclasses of PatternElement (c.f. Figure 4), so that its instances may contain other patterns. Furthermore, class Match gains a reflexive association to model nested matches. For all matches, this relation has to be coherent with their respective patterns' nesting. This condition implies that a child pattern is evaluated only if a match of its parent pattern exists.

Semantically, nested patterns are matched independently from each other if constraints only refer to variables of the same pattern. The resulting set of matches (if "joining" assignments of parent and child matches) is the cross-product of matches of non-nested patterns. However, there are two possible interactions between parent and child patterns: *Firstly*, constraints can restrict variables of child patterns. As the child pattern's variable is not bound when checking fulfilledness of the parent pattern, such constraints cannot be verified. Fulfilledness of the constraint's pattern therefore only demands that no constraint is violated, thus allowing unevaluable constraints to persist. In addition, a pattern is matched only if no constraint of any ancestor pattern is violated by its variable assignments. *Secondly*, variables can be restricted by constraints of child patterns. Here, the common conditions for non-nested queries suffice, demanding fulfilledness (more generally non-violatedness) of a pattern's constraints. *However*, references between entities of *sibling* patterns are *forbidden* to keep matches independent of each other.

A final aspect on nested queries that needs to be addressed here is the *processing* of the resulting match structure. As result, we determine the validity of a match with regards to its child matches. From the application's point of view, an invalid match is treated as non-existent. Match validity can be specified w.r.t. two criteria: The *pattern condition* ensures that a match contains appropriate child matches for a *specific* child pattern. One usage of this condition is to reason on the number of these matches, e.g. *at most zero matches* to model negative application conditions. In the following, nested patterns are treated according to the intuitive *at least one* cardinality. Another approach is the *group condition*, which specifies the treatment of distinct child patterns (if any, otherwise the condition is true). Here, e.g. a boolean operator such as $\vee$ or $\wedge$ can be applied on the pattern conditions' results. In the following, we assume an $\vee$ condition, so that *at least one match* for *at least one child pattern* has to be found.

Figure 7a shows a nested pattern searching for paths of length 0 or 1. The outer variables are assumed to be bound before, in surrounding parent pattern. Pattern ① contains a single IsomorphismConstraint. As only the outer variables are bound when searching for matches of



(a) Nested patterns checking length 0 and 1

(b) Recursive pattern, initial state

(c) External links and according mappings marked

(d) After 1st replacement

Figure 7: Patterns for incidence closure

①, this constraint is always fulfilled. Therefore, a single match without any assignments exists for this pattern. The inner pattern ② checks whether the outer variables have the same value assigned, which represents a path of length 0. In contrast, pattern ③ traverses an edge (according variable and type check are omitted here), and checks whether the reached node equal the outer-right variable's value. The IsomorphismConstraint of ① requires that the target node is not identical to the outer-left variable's value, to exclude reflexive edges. Processing rules discussed above state that at least *one* of these nested patterns need to be matched to obtain a valid match for ①.

4.4 Recursive constraint reduction

Although nested queries allow to express *alternative* patterns, they can only be used to check a limited number of variables. Usually, this number cannot be given in advance, e.g. the IncidenceClosureConstraint requires to check for paths of an *arbitrary* length. The only, albeit impossible, solution would be the specification of an infinite number of patterns. Therefore, we apply a mechanism for recursive expansion of queries at *runtime*.

The language's meta-model is extended by a PatternReference class, which references a pattern defined by the developer. References are replaced by the corresponding pattern when its container pattern is matched successfully. Recursion is achieved by copying a pattern into itself, also copying the reference being expanded. If multiple references exist within the same pattern, their order of replacement is undefined. However, consistency conditions introduced below ensure that the result is indeed independent of this order. Furthermore, reference expansion should be guaranteed to terminate in recursive situations. Although this property is not ensured directly, expansion can only occur whenever a pattern is matched. Therefore, termination of reference expansion is given if only a finite number of patterns can be matched. Obviously, this can be achieved by an IsomorphismConstraint limiting at least one variable per pattern to an entity not assigned to other variables. Therefore, finiteness of the host graph implies finiteness of matched patterns and expansion steps.

The actual application of pattern references is introduced by referring to the IncidenceClosure-Constraint. Figure 7b shows a variant of the nested pattern introduced above. In contrast to Figure 7a, pattern ③ does not contain an own IdentityConstraint to check the connectedness of the path ends. Instead, two PatternReferences are given: The upper one refers to pattern ②, which means that this pattern is copied *into* pattern ③ if the latter can be matched. Furthermore, the lower reference copies pattern ③ into itself.

Reference expansion is conducted as follows: Each PatternReference is replaced by a new pattern created inside the reference's container, and filled with copies of the referenced pattern's entities. This covers the entities' types, attribute values, and connectedness to other copied entities. However, the question remains how the copied pattern's *context* should be handled. This context is defined by the edges connecting its contained entities to entities not contained in the pattern being copied, the so-called *external links*. Figure 7c highlights the external links of Figure 7b for both copied patterns. Here, this concerns Restricts edges (four times), but also the pattern referred to by the upper PatternReference.

For each pattern reference, the developer has to specify a *mapping* of entities connected to external links, relating them to entities that should be connected to the referred pattern. Identical mapping of an element to itself is a valid choice. Mappings are copied along with other pattern entities during reference expansion, which is required for recursive expansions.

In order to achieve the desired replacement in case of the IncidenceClosureConstraint, the following mappings are required (c.f. broad arrows in Figure 7c):

- The *upper reference* copies pattern ② into pattern ③ to check value-identity of the outer-right variable and the variable of ③. Therefore, the outer-left variable referenced by ② is mapped to the variable of ③, whereas the outer-right variable is mapped identically.

- Expansion of the *lower reference* should yield a query for path of length 2. Therefore, the same mapping of the outer-left variable to the variable of ③ applies here, such that the IncidenceConstraint of the *copy* of pattern ③ refers to the original's variable as source.

- The traversed node should not have been visited before, so all node variables are connected to the IsomorphismConstraint of ①, which is mapped identically for this purpose. This constraint ensures termination of the replacement, as discussed above.

- A last external link of the lower reference is the pattern referred to by the *upper* reference. Here, pattern ② is mapped to its reference, such that the copied reference will refer to the *expanded* upper reference of ③. In this case, identical mapping would lead to broken copied mappings in later expansion steps, if the lower reference is expanded first.

Using these mappings, expanding both references yields the pattern structure shown in Figure 7d. Expanding the upper reference results in ⑤, whereas the lower one is expanded to ④. The resulting query checks for paths of length 0 by matching ① and ②, and 1 by matching ①,③, and ⑤, respectively. Paths of length 2 can be found after the next step, using ①,③,④, and the expanded reference to ⑤.

This section showed how complex or application-specific language constructs (represented by constraints) can be reduced to basic ones. With the presented nested query mechanism, recursive expression can be captured as well. Although its evaluation might be inefficient, it serves as the guideline for implementing the DRAGOS Query & Transformation Language. This is required by the fact that the actual storage backend of DRAGOS is exchangable, and so is the implementation of its language. As discussed in [Wei07], such implementations may either rely on the DRAGOS core graph model, or convert rules into a backend-specific format. e.g. SQL statements. To provide an efficient implementation, language extensions might also be converted into such a backend-specific language. The modeled reduction rules in this case serve as the formal definition and as reference used in test-based validation of the specific implementations.

5 Related Work

In contrast to previous publications on DRAGOS [Böh04] and the according Query & Transformation Language [Wei07], this paper focusses on the language's extensibility. In this section, we give a brief comparison to other research in the area of graph transformations.

Graph transformations based on constraint satisfaction. The DRAGOS Query & Transformation Language is based on the theory of *constraint satisfaction problems* (CSPs) known from artificial intelligence. CSPs are well-suited to model graph pattern matching by solving the *subgraph-isomorphism problem* [LV02]. Algorithms have been proposed for this purpose which circumvent efficiency concerns arising from the problem's complexity in most situations [FSV01]. In our work, we implement the proposed language based on existing databases, and therefore extensive development of a basic constraint solver is not of crucial importance. Instead, we focus on implementations based on sophisticated storage backends like databases.

Graph transformations on databases. As briefly mentioned in Section 4, we not only provide an operational implementation of the presented language. In addition, queries and transformation rules can be converted into a language offered by the respective storage backend, e.g. SQL. Building GTS on this language has been presented by [VFV06], which is based on the construction of database views and update operations from graph transformation rules.

Ongoing work in our project generalizes this idea by deriving SQL statements from the more expressive DRAGOS Query & Transformation Language. Furthermore, its language structure allows an easier processing, as it already separates between variables and constraints. Therefore, we are able to apply an extensible rule language on various storage backends, not limited to the SQL or one of its DBMS-specific variants. Implementations do not need to cover the entire DRAGOS Query & Transformation Language, as extended language constructs may be reduced to basic ones. Moreover, evaluation may fall back on a generic implementation only based on the DRAGOS graph model, which is independent of the actual storage backend.

Complex pattern matching. A large amount of recent publications deals with the representation and semantical definition of complex graph patterns. Besides recent work in our own department, [Ba07] proposes set-valued graph patterns by grouping, and [LLP07] discusses repeated pattern structures beyond binary path expressions. [DHJ+07] utilizes graph grammars to build transformation rules, similar to two-level grammars.

The stepwise extension of pattern references can be seen as a simple graph grammar, and indeed provides similar functionality. Therefore, arbitrary repeated structures can be expressed by building nested queries, as shown in Figure 7. This is not limited to binary path expressions, but can be applied for n-ary structures as well. The definition of proper sub-pattern interfaces, expressing how sub-patterns are to be glued together, is given by a consistent element mapping.

Graph transformation languages for visual programming. Graph transformation languages like PROGRES provide similar functionality to the DRAGOS Query & Transformation Language. In fact, PROGRES can already generate code to store the runtime data persistently using DRAGOS. However, this approach leads to inefficient applications because the generated code performs many simple operations on the DRAGOS graph model. In our approach, DRAGOS interprets transformation rules itself, and hence may utilize storage backends more appropriately.

In contrast to common graph transformation languages, the low-level DRAGOS Query & Transformation Language is not feasible for direct use by a specificator. Therefore, it should not be considered as competitor to existing languages, but as a common core for existing and new languages to build on.

6 Conclusion

In this paper, we introduced the Query & Transformation Language currently being developed for the DRAGOS graph database. This language especially focuses on extensibility, which is the core aspect of this publication. Developers may choose to add new constructs to the language in case existing ones do not suffice the application's needs or do not match its semantics. These are implemented by reduction to existing ones, also allowing recursive substitutions. In addition, language constructs may be converted into a storage-specific query such as SQL statements.

The presented work is fully implemented based on the DRAGOS graph model interface, designated as *generic* implementation in [Wei07]. Currently, we are working on an SQL-based solution. Interesting problems remain in the recursive evaluation of queries, which cannot be expressed directly in many database systems[2]. Upon completion, we will conduct performance evaluations comparing the Query & Transformation Language to DRAGOS applied in the code generation approach. Furthermore, comparisons to other graph transformation solutions based on databases are of interest.

Currently, we are embedding support for control flow into the language definition and its generic implementation. Core features of this mechanism include hierarchical rule composition, optional dataflow and rule invocation. Rule application strategies will allow non-deterministic and random (with or without backtracking) processing of multiple matches. Using this mechanism, rules can be combined to complex graph transformation systems.

As next step, we will investigate which additional language constructs are required to support different approaches to graph transformations, such as the algebraic approach or hyper-edge

[2] Altough recursive SELECT statements are defined by SQL3, support is optional and obviously not very popular.

replacement grammars [Hab93]. This way, DRAGOS can serve as a platform to develop new constructs for graph transformation languages by offering a high-level extension mechanism.

Bibliography

[Ba07] D. Balasubramanian, et al. Applying a Grouping Operator in Model Transformations. Pp. 406–421 in [SNZ07].

[Böh04] B. Böhlen. Specific Graph Models and Their Mappings to a Common Model. In Pfaltz et al. (eds.). LNCS 3062, pp. 45–60. Springer, 2004.

[BW02] A. D. Brucker, B. Wolff. A Proposal for a Formal OCL Semantics in Isabelle/HOL. In Muñoz et al. (eds.), *Theorem Proving in Higher Order Logics*. Lect. Notes in Comp. Sci. 2410, pp. 99–114. Springer, Hampton, VA, USA, 2002.

[DHJ$^+$07] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, N. V. Eetvelde. Shaped Generic Graph Transformation. Pp. 197–212 in [SNZ07].

[FSV01] P. Foggia, C. Sansone, M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*. Pp. 188–199. 2001.

[Hab93] A. Habel. *Hyperedge Replacement: Grammars and Languages*. Lect. Notes in Comp. Sci. 643. Springer, 1993.

[HWS00] R. Holt, A. Winter, A. Schürr. GXL: Towards a Standard Exchange Format. In *Proc. of the* 7^{th} *Working Conference on Reverse Engineering (WCRE '00)*. Pp. 162–171. IEEE Computer Society Press, 2000.

[LLP07] J. Lindqvist, T. Lundkvist, I. Porres. A Query Language With the Star Operator. In Ehrig and Giese (eds.), *Graph Transformation and Visual Modeling Techniques*. ECEASST 6, pp. 69–80. 2007.

[LV02] J. Larrosa, G. Valiente. Constraint Satisfaction Algorithms for Graph Pattern Matching. *Mathematical Structures in Computer Science* 12(4):403–422, 2002.

[SNZ07] A. Schürr, M. Nagl, A. Zündorf (eds.). *Applications of Graph Transformation with Industrial Relevance (AGTIVE'07), preliminary proceedings*. 2007.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. (eds.). Pp. 487–550. Volume 2. World Scientific, 1999.

[VFV06] G. Varró, K. Friedl, D. Varró. Implementing a Graph Transformation Engine in Relational Databases. *Journal on Software and Systems Modeling* 5(3):313–341, 2006.

[Wei07] E. Weinell. Adaptable Support for Queries and Transformations for the DRAGOS Graph-Database. Pp. 390–405 in [SNZ07].