Electronic Communications of the EASST
Volume 10 (2008)

Proceedings of the
Seventh International Workshop on
Graph Transformation and Visual Modeling Techniques
(GT-VMT 2008)

Extending Graph Query Languages by Reduction

Erhard Weinell

14 pages

Guest Editors: Claudia Ermel, Reiko Heckel, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

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

Abstract: Graph grammars are a well-founded technology for visually specifying computations or the processing of complex data structures. Up to now, numerous languages and tools for graph transformations exist, whilst new ones are proposed regularly. However, these tools have no technical basis such as an execution framework or data storage in common. Instead, graph transformation machineries are usually implemented anew each time.

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

Graph transformations have shown to provide appropriate means to approach various challenges related to software engineering. They not only support the visual programming paradigm, but are also able to express model-to-model transformations, just to mention two prominent scenarios. Up to now, a lot of languages and tools related to graph transformations have been developed, differing w.r.t. application domains and provided language expressiveness. Nevertheless, the all-encompassing system does not seem to exist, as new ones are proposed regularly. All of these tools share common requirements: A data repository to store graph structures persistently, and an according execution framework to carry out the specified transformation rules.

In our project, we strive to support the construction of tools related to graph transformations by means of a common platform. Persistent storage of graph structures is provided by the DRAGOS database [Böh04], which continues a series of graph databases dating back to the eighties [LS88]. The ability to query and transform the stored graphs is currently being added [Wei08]. In this regard, we do not provide an enclosed language and environment, e.g. as applied by PROGRES [SWZ99]. Instead, a *core* Query & Transformation Language (QTL) is provided which can be easily applied as base-layer for other graph languages. This not only enables to specify graph transformations e.g. using PROGRES syntax, but also allows to construct specialized languages comprising sophisticated query functionality, or model-to-model translations.

Tool developers therefore can refer to high-level functionality provided by the QTL, instead of developing proprietary interpreters or code generation modules required without such a platform.

Figure 6c gives an architectural overview of the pursued approach. As the figure suggests, Graph Transformation Tools have to provide a language-specific editor, and a transformation module. The latter's responsibility is to *translate* rules of the specialized language to the QTL. These rules are stored in a Rule Repository, from where they are loaded into the Query & Transformation Mechanism (QTM) for processing. The QTM can be controlled by UI frameworks to invoke rules on the stored data. Furthermore, the QTM is able to handle multiple data storages at a time, although all depicted DRAGOS instances can be integrated into a single one at runtime.

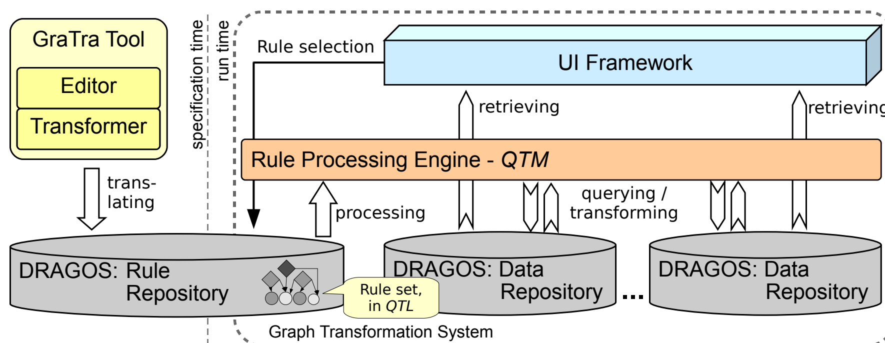


Figure 1: DRAGOS / QTL interaction

The critical step in constructing specialized graph languages is the concise representation of language transformations, e.g. an appropriately modeled transformation module. Despite the QTL's ability of declarative modeling, such translations can become complex and hard to maintain, depending on the specialized language's complexity. Therefore, we allow to *extend* the QTL by additional constructs to align both languages' conceptual levels. For example, an extension can comprise constructs to handle traceability links, which is useful if a model-transformation language is considered as specialized graph language.

Language constructs can be realized by extending the QTL's implementation, i.e. by textual programming. However, this approach requires precise technical knowledge on the QTM's API, whilst the added constructs' effects are hard to validate. Therefore, we alternatively allow to define language constructs by *reducing* them to basic ones. As result, extensions can be specified in a model-based way, easing development and making their effects easier to comprehend due to the known basic constructs' effects.

The present paper introduces this extension mechanism by means of a simple example, namely the detailed modeling of type checking in graph languages. This represents another application of language extensions, besides conceptual alignment: Core language extensions can also be applied to precisely model a specialized language's semantics, e.g. which types of instances are considered *valid* for a specific query.

The rest of this paper is structured as follows: We first introduce the basic functionality of DRAGOS in Section 2, and afterwards the QTL 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] is a data repository for the management of graph structures. Using graphs as fundamental data model, even complex data structures can be represented without need for technical helper elements. In contrast, the relational data model often requires additional elements, such as extra tables to store many-to-many relations.

Architecture. The DRAGOS architecture allows flexbile adaption to a given application domain, amongst other things by exchanging the underlying graph storage module. This way, developers may choose between fast in-memory solutions, and transactional storages based on relational or object-oriented databases. Furthermore, the graph storage module can be adapted to use existing model repositories, too.

Graph model. DRAGOS offers a rich graph model originally inspired by the Graph eXchange Language (GXL) [HWS00]. Among other things, the graph model 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

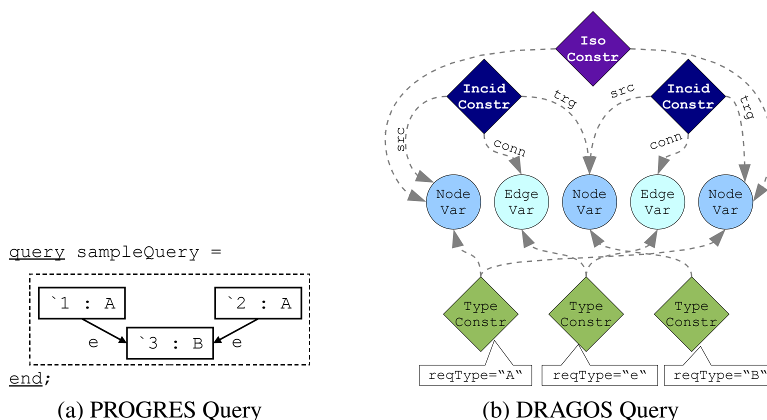
In this section, we present the Query & Transformation Language by means of an example, relating it to the well-known graph transformation language PROGRES [SWZ99]. The language's abstract syntax is presented and its semantics are sketched. Unfortunately, no comprehensive definition of the QTL can be 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 [Wei08].

3.1 Example Query

Figure 2a 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 QTL separates between graph entities to be *searched* from the conditions that need to be *fulfilled* by these entities. The DRAGOS query shown in Figure 2b 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.

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



(a) PROGRES Query (b) DRAGOS Query

Figure 2: Sample query searching three connected nodes

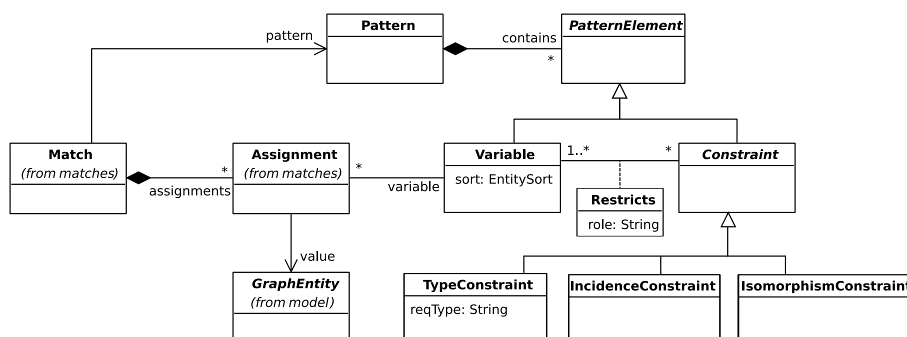


Figure 3: Meta-Model of the Query & Transformation Language (simplified)

Constraints (depicted as diamonds in Figure 2b) 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 restrict 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 3 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 3 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 QTL. 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.

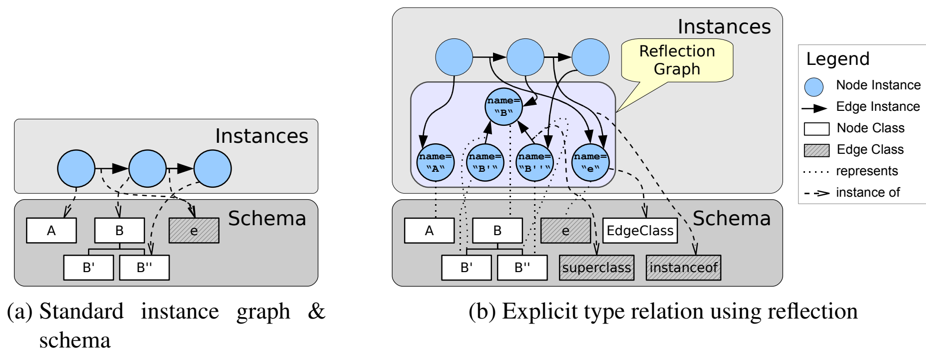
The presented invariants (partially) define the validity of Matches, but do not state how such an assignment can be computed. Language implementations therefore need to provide an operational implementation of these invariants.

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 4. On the left side (Figure 4a), the standard situation using separate instance and schema models is shown. Dashed arrows indicate an entities' type. However,



(a) Standard instance graph & schema

(b) Explicit type relation using reflection

Figure 4: Reflection graph to query types

the QTM is not able to traverse this relation or examine the entities' types. Therefore, Figure 4b 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 QTM 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.

4.2 Basic constraint reduction

To revive the initial example, Figure 5 (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.

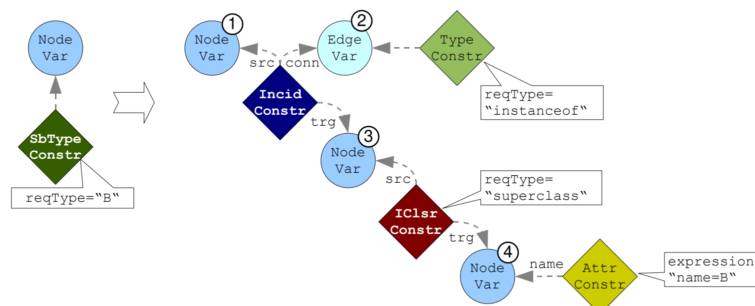


Figure 5: Definition of the SubTypeConstraint

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 5 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 5 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 proposed QTL 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 QTL meta-model is extended by adding Pattern to the subclasses of PatternElement (c.f. Figure 3), 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 6a 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 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.

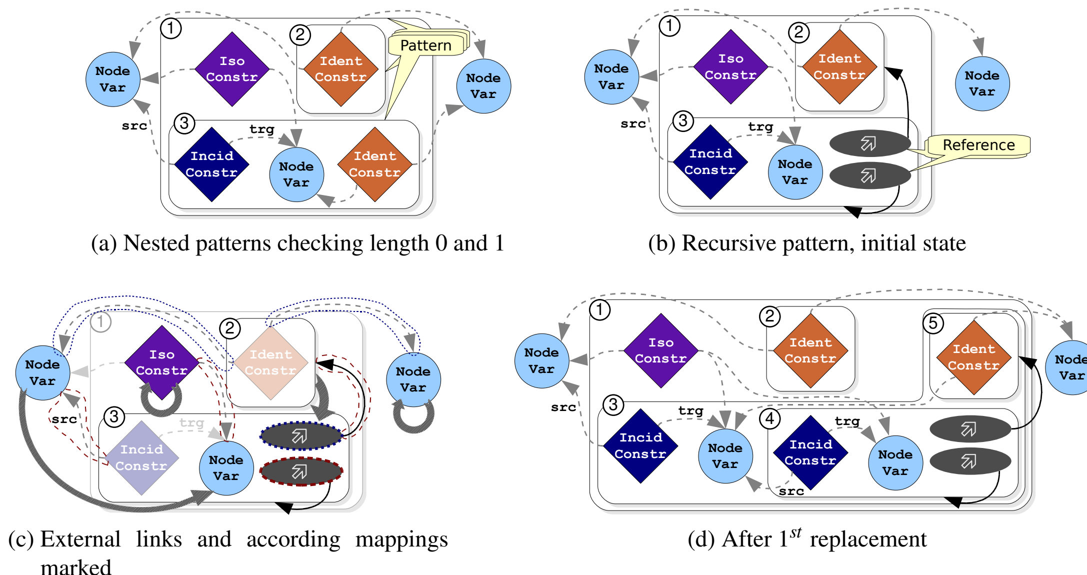
4.4 Recursive constraint reduction

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 IncidenceClosureConstraint. Figure 6b shows a variant of the nested pattern introduced above. In contrast to Figure 6a, 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



(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 6: Patterns for incidence closure

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 6c highlights the external links of Figure 6b 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 6c):

- 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 6d. 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 QTL. This is required by the fact that the actual storage backend of DRAGOS is exchangeable, and so is the implementation of its language. As discussed in [Wei08], 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 QTL [Wei08], 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 QTL 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* [LV02]. In our work, we aim to implement the QTL 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 [HVV07], 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, whereas NACs are treated as special case of nested patterns in our QTL.

Graph transformations on relational databases. Implementing GTS on established relational databases has been presented initially by the authors of [VFV06]. 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 [VFV06], 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 large queries. To relieve this drawback, a specialized query optimizer should adapt results from search-plan driven code generation found in common graph transformation tools.

Graph transformations for visual programming. Graph transformation languages like PROGRES provide similar functionality like the DRAGOS QTL. However, the latter has been designed as *base layer* of specialized graph languages, and therefore especially focuses on two aspects: First, the language has provide common functionality, yet being extensible, as discussed in Section 1. In contrast, the architecture of PROGRES has not been designed for extensibility at all. Second, the provided platform has to be able to represent and translate specialized languages. The QTM meets this requirement by applying graph-oriented data storages not only for the runtime data, but also for representing rules modeled using specialized languages and there counterparts in the QTL.

In contrast to common graph transformation languages, the low-level DRAGOS QTL is not feasible for direct use by developers. 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 QTL 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 [Wei08]. 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

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

evaluations comparing the QTL 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 support the development of specialized graph languages on top of the QTL. For this purpose, a set of parser specifications are being developed to import textual documents from graph languages into the DRAGOS database. Currently, we are working on support for the general-purpose rewriting language PROGRES and a derivate of the query language GReQL [KW99]. To enact the corresponding specifications, we also develop a transformation language to translate them to the QTL.

Bibliography

[Böh04] B. Böhlen. Specific Graph Models and Their Mappings to a Common Model. In Pfaltz et al. (eds.), 2nd *Intl. Workshop on Applications of Graph Transformations with Industrial Relevance, (AGTIVE)*. Lect. Notes in Comp. Sci. 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.

[HVV07] Á. Horváth, G. Varró, D. Varró. Generic Search Plans for Matching Advanced Graph Patterns. In Ehrig and Giese (eds.), *Graph Transformation and Visual Modeling Techniques, 6th Intl. Workshop*. Elec. Comm. of the EASST 6, pp. 57–68. 2007.

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

[KW99] B. Kullbach, A. Winter. Querying as an enabling technology in software reengineering. In *Proc. of the 3rd Europ. Conf. on Software Maintenance and Reengineering*. Pp. 42–50. IEEE Computer Society Press, 1999.

[LS88] C. Lewerentz, A. Schürr. GRAS, a Management System for Graph-Like Documents. In *Proc. of the 3rd International Conference on Data and Knowledge Bases*. Pp. 19–31. Morgan Kaufmann, 1988.

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

[SNZ08] A. Schürr, M. Nagl, A. Zündorf (eds.). *Proc. of the 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE)*. Lect. Notes in Comp. Sci. 5088. Springer, 2008.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*. 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.

[Wei08] E. Weinell. Adaptable Support for Queries and Transformations for the DRAGOS Graph-Database. Pp. 369–411 in [SNZ08].