# Recursive Graph Pattern Matching ⋆

## With Magic Sets and Global Search Plans

Gergely Varró[1], Ákos Horváth[2], and Dániel Varró[2]

[1] Department of Computer Science and Information Theory
Budapest University of Technology and Economics
gervarro@cs.bme.hu
[2] Department of Measurement and Information Systems
Budapest University of Technology and Economics
[ahorvath,varro]@mit.bme.hu

**Abstract.** We present core data structures and algorithms for matching graph patterns with general recursion. Our approach uses magic sets, a well-known technique from deductive databases, which combines fixpoint-based bottom-up query evaluation with top-down handling of input parameters. Furthermore, this technique is enhanced with the global search plans, thus non-recursive calls are always flattened before elementary pattern matching operations are initiated in order to improve performance. Our approach is exemplified using VIATRA2.

## 1 Introduction

Graph transformation (GT) [1] is a frequently used means to capture model transformations in the context of model-driven software development. Graph transformation rules provide a declarative, rule and pattern-based language for specifying both inter-language and intra-language model manipulations for model analysis, refactoring or simulation.

GT rules consist of a left-hand side (LHS) and a right-hand side (RHS) graph pattern. The LHS specifies contextual conditions which should hold as a precondition for applying the rule, which is checked by graph pattern matching. Then the model is manipulated by calculating the difference of the RHS and the LHS in the model.

However, in order to design complex transformations, the core GT formalism has been extended to address reusability or maintainability. For instance, graph transformation units [2], modules [3] or programs [4, 5] have been introduced where elementary GT rules are enriched with control structures.

An alternate, and more declarative way for reusability has also been introduced (in systems like VIATRA2 [6] or Tefkat [7]) where graph patterns are stand-alone concepts, which can be assembled into more complex patterns and/or transformation rules by pattern composition (or pattern call). This concept is quite similar to other popular

⋆ This work was partially supported by the SENSORIA European IP (IST-3-016004), the Hungarian National Research Fund and the National Office for Research and Technology (grant No. 67651, OTKA), and the János Bolyai scholarship

declarative formalisms in logic programming or deductive database systems (like Prolog or Datalog), where basic facts and complex predicates are treated identically when evaluating a query. A key performance issue when matching graph patterns in case of pattern composition is to generate a single global search plan for the flattened pattern, which is discussed in [8].

A natural extension for pattern-level reuse is to allow recursive calls in case of pattern composition, i.e., when a predefined graph pattern may call itself or other patterns recursively. Investigating recursion in graph transformation rules has recently become very popular with several approaches [9, 10] targeting mainly its specification aspects. However, these approaches mostly assume simple recursion where a pattern may call itself only once in a single execution branch.

In the current paper, we make only a single, very general assumption on recursion, namely, parameters of negative application conditions must be bound at the time of their invocation , but otherwise arbitrary recursive calls are allowed. As the main contribution, we define data structures and sketch core algorithms how recursive graph patterns can be matched based upon *magic sets* [11], a well-known technique from deductive databases, which combines fixpoint-based bottom-up query evaluation with top-down handling of input parameters. Furthermore, this technique is enhanced with the global search plans of [8, 12] thus non-recursive calls are always flattened before elementary pattern matching operations are initiated to improve performance.

The remainder is structured as follows. First, Section 2 briefly introduces a combined graph-based representation for models and metamodels used in the paper (and in the VIATRA2 framework). Then Section 3 describes the overview of our approach, while Section 4 and Section 5 propose our data structures and algorithms used in the compile and run-time phases of the recursive pattern matcher, respectively. Related work is discussed in Section 6, while Section 7 concludes our paper.

## 2 Background

First we informally introduce models, metamodels and graph patterns used in the paper, using the object-relational mapping defined in the model transformation contest of [13] as a running example. This transformation was captured by graph transformation rules using recursive patterns in [7, 14].

### 2.1 VIATRA Models and Metamodels

Metamodeling provides the structural definition (i.e., abstract syntax) of modeling languages.

In the paper, we use a unified directed graph representation [6] which stores metamodels and models in a combined model space. Intuitively, the morphisms from instance nodes (and edges) to their respective node (edge) types are stored explicitly in our graph model. As a summary, nodes represent basic concepts of a (modeling) domain, while edges represent the relationships between model elements. This unified graph representation serves as the underlying model of the VIATRA2 framework.

This way, graph nodes (called entities in VIATRA2, depicted as a rectangle in Fig. 1) uniformly represent MOF packages, classes, or objects on different metalevels, while graph edges with identities (called relations in VIATRA2, depicted as a solid line in Fig. 1) denote MOF association ends, attributes, link ends, and slots in a uniform way. Nodes are arranged into a strict containment hierarchy to denote model element containment either on the metamodel or model-level.

*Example 1.* Figure 1 presents the joint representation of a simplified UML metamodel and an instance model. The metamodel is depicted on the right side. Both the classes of the metamodel (such as `cls`, `assoc`, etc.) and the objects of the instance model (such as `car`, `plt`) uniformly appear as nodes (entities), while relations between nodes are illustrated by solid edges.

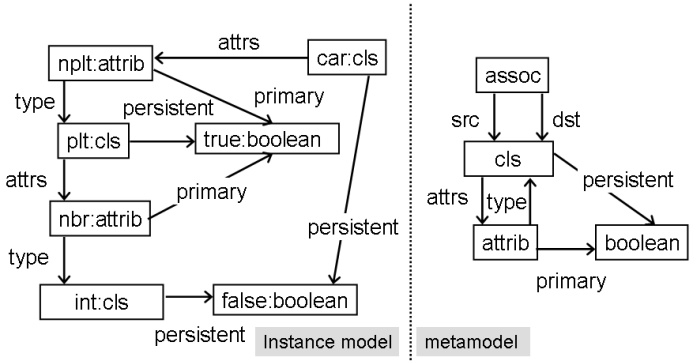



**Fig. 1.** Sample instance model and metamodel

### 2.2 Graph Patterns

Graph transformation (GT) is a rule and pattern-based paradigm frequently used for describing model transformations. A graph transformation rule contains a left-hand side graph LHS (or graph pattern) and a right-hand side graph RHS, and (one or more) negative application condition graphs (NAC) connected to LHS. Graph patterns (precondition pattern) consist of the LHS pattern, the NAC pattern, and the mapping between them. They are describe by pattern bodies consist of a set of constraints that have to be fulfilled by a model to apply graph transformation rule. In order to define recursive pattern we allow alternate (OR) *pattern bodies* for a pattern, with a meaning that the pattern is fulfilled if at least one of its bodies is fulfilled.

As different graph transformation languages allow different language constraints (e.g., containment between model elements), in the following we use the constraints of the VIATRA2 framework containing (i) *structural constraints* prescribing the existence of nodes and edges of a given type, (ii) *check constraints* capturing term evaluation

over the attributes of the matched elements (using the `check` keyword), and (iii) *pattern invocation constraints* allowing pattern composition (invocation) of other patterns supported in a declarative way (using the `find` keyword). The semantics of this reference is similar to that of the declarative Horn clauses, where the caller pattern can be fulfilled only if their local constructs can be matched, and if the pattern invoked with the *actual parameters* is also fulfilled.

*Example 2.* The graphical and VIATRA2 textual representation of the graph patterns of the object-relational mapping are depicted in Fig. 2, 3, and 4. Both `classHasAttr` and `classHasIncludedAttr` contain all recursive features offered by VIATRA2, from which we use `classHasAttr` as our running example. The meaning and purpose of the `classHasAttr` that a `Cls` "has" an `Attr` for the purpose of the mapping if, (i) it is directly owned and a primitive type, or (ii) a referenced `Cls` (via an `Attr` or an `Assoc`) "has" the `Attr`, or (iii) the children of the `Cls` "have" the `Attr`. Informally the meaning of the `classHasIncludedAttr` to map additional attributes along references and inheritance, while `classHasReference` is a helper pattern for matching "references" (attributes and associations) between classes.

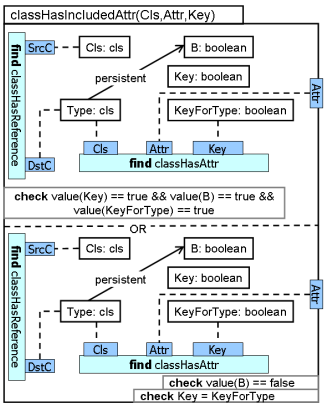
The *pattern* `classHasAttr` contains three *formal parameters*: `Cls`, `Attr` and `Key`, and it consists of three pattern bodies. The first body prescribes that there exists a `Cls` class with an `Attr` attribute, which has a `Boolean` value `Key` denoting whether attribute `Attr` is a primary key. The second prescribes the `classHasIncludedAttr` pattern invocation with formal parameters (`Cls`, `Attr`, `Key`) of the caller pattern as actual parameters (depicted by grey boxes on the invoked pattern). The last body prescribes three constraints: (i) there exists two classes `SubCls` and `Cls` with a parent relation between them, (ii) the `classHasAttr` pattern invocation where the first parameter is the local `SubCls` element (depicted by dashed line), while the two other (`Attr` and `Key`) are the formal parameters of the caller pattern, and (iii) that the value of `Key` is false.

### 2.3 Graph Pattern Matching

The most critical step of graph transformation is graph pattern matching, i.e., to find a matching of the LHS pattern in the model, that is not invalidated by a matching of the negative application condition graph NAC, which prohibits the presence of certain combinations of nodes and edges. Thus we restrict our investigations only to graph patterns and graph pattern matching for the current paper.

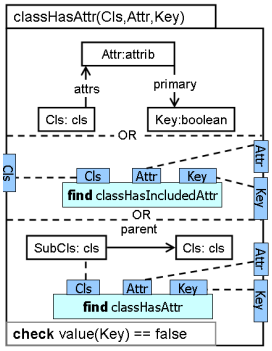
During pattern matching each variable of a graph pattern is bound to a node in the model such that this matching (binding) is consistent with edge labels, and source and target nodes of the model.

Traditional model transformation approaches handle recursive invocation in a top-down imperative way, usually integrated into the control flow rather than the patterns themselves. We propose a fixpoint-based bottom-up evaluation approach combined with a top-down handling of input parameters following deductive database techniques. As [11] states, such a technique benefits from the advantages (e.g., convergence detection, strong focus on relevant facts) of both the semi-naive bottom-up and the traditional top-down style, queue-based rule/goal tree expansion methods. Furthermore, it



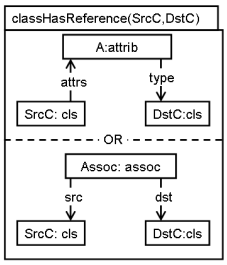

```
pattern classHasIncludedAttr(Cls, Attr, Key) =
{cls(Cls);
 find classHasReference(Cls, Type);
 find classHasAttr(Type,Attr, KeyForType);
 cls(Type);
 cls.persistent(Type, B);
 Boolean(B);
 Boolean(Key);
 Boolean(KeyForType);
 check(value(B) == "true" &&
 value(Key) == "true" &&
 value(KeyForType) == "false");
}or{
 cls(Cls);
 find classHasReference(Cls, Type);
 find classHasAttr(Type, Attr, KeyForType);
 cls(Type);
 cls.persistent(Type, B);
 Boolean(B);
 check(value(B) == "false");
 Boolean(Key);
 Boolean(KeyForType);
 Key = KeyForType;
}
```

**Fig. 2.** classHasIncludedAttr pattern




```
pattern classHasAttr(Cls, Attr, Key) =
{cls(Cls);
 cls.attrs(Cls, Attr);
 attrib(Attr);
 attrib.primary(Attr, Key);
 datatypes.Boolean(Key);
}or{
find classHasIncludedAttr(Cls, Attr, Key);
}or{
 cls(SubCls);
 cls.parent(SubCls, Cls);
 cls(Cls);
 find classHasAttr(SubCls, Attr, Key);
 datatypes.Boolean(Key);
 check(value(Key) == "false");
}
```

**Fig. 3.** classHasAttr pattern




```
pattern classHasReference(SrcC, DstC) =
{cls(SrcC);
 cls.attrs(SrcC, A);
 attrib(A);
 attrib.type(A, DstC);
 cls(DstC);
}or{
 assoc(Assoc);
 cls(SrcC);
 assoc.src(Assoc, SrcC);
 cls(DstC);
 assoc.dest(Assoc, DstC);
}
```

**Fig. 4.** classHasReference pattern

terminates after the same number of iterations (up to a constant factor), and it provenly produces [11] the same results as the others.

## 3 Overview of the Approach

The proposed workflow of implementing recursive graph pattern matching is summarized in Fig. 5.

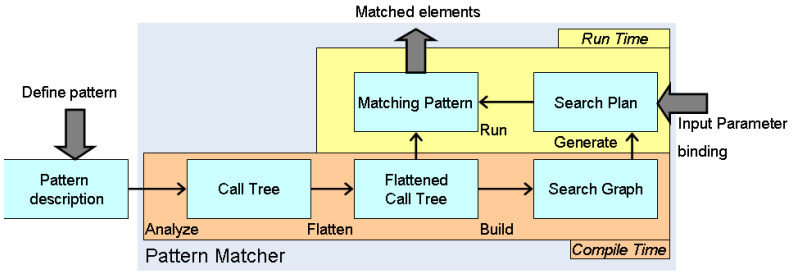



**Fig. 5.** Overview of the recursive pattern matching approach

We separate compile time parts from run-time parts, where each part consists of the following steps:

- At compile time each step is calculated once for each pattern description.
  - First, for each pattern description a *call tree* is generated capturing how patterns call other patterns.
  - Then for each *call tree flattened patterns* are generated. The use of flattened patterns allows the optimization of pattern matching in a global scope (e.g., edges that are defined in different patterns can be traversed one after the other).
  - For each *flattened pattern* a corresponding *search graph* is generated. The search graph is glued from the patterns of the flattened pattern body according to the passed parameters of the calls.
- After initializing the previous data structures at compile time, run-time steps have to be calculated for each separate pattern invocation.
  - *Search plan* is generated from the *search graph* based on the parameter binding to drive the pattern matching process.
  - Then matchings are calculated by an iterative bottom-up recursion evaluation using magic sets, helping the pattern matcher to focus only on matches relevant to the input parameter binding.

## 4 Compile Time Steps of the Recursive Pattern Matcher

In this section we briefly introduce the data structures and algorithms needed for the compile time tasks of the recursive pattern matcher.

### 4.1 Call Tree

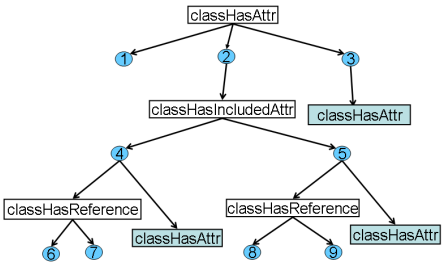



**Fig. 6.** Call tree of the classHasAttr graph pattern

A *call tree* is a directed bipartite tree describing the structural dependencies of a given pattern. It is constructed by a traversal process, which explores the possible body alternatives of a pattern and all the pattern invocations in a depth first manner.

Nodes on the odd levels of the call tree represent *pattern heads* (denoted as simple rectangles) and (pattern) *references* (illustrated by grey rectangles), while nodes on the even levels denote (pattern) *bodies* (symbolized with numbered circles). The fact that a *body* is a disjunctive alternative of a *pattern head* is expressed by an edge connecting the corresponding *pattern head* to the *body*. Edges connecting bodies to pattern heads and references represent non-recursive and recursive pattern invocations, respectively.

*Example 3.* The call tree of pattern `classHasAttr` of Fig. 3 is illustrated in Fig. 6.

The `classHasAttr` pattern (head) has three pattern bodies depicted by circles with numbers `1`, `2` and `3`. Pattern body `2` invokes the `classHasIncludedAttr` pattern head which has pattern bodies `4` and `5`. Both of these bodies have similar subtrees, as they differ only in the check constraint. The `classHasReference` contains two pattern bodies, and contained twice in the *call tree* as it is invoked separately from `4` and `5`.

### 4.2 Flattening

In order to provide better performance for pattern matching, we use search plan optimization techniques, where optimization can be considered as a process that orders constraints to provide an efficient evaluation plan for their run-time execution.

As current optimization techniques [4, 8, 12] have been developed for non-recursive use, they operate on the scope of pattern bodies, which means that a separate optimization procedure is executed for the set of constraints defined by a given body. This approach often results in poor search plans for a recursive pattern matcher due to the lack of global view for the optimizer on the overall set of structural constraints.

In order to get better search plans, the operation scope of the optimizer module is increased by flattening the call tree and by merging pattern bodies and recursive invocations resulting in a larger set of constraints to be processed by the optimizer.

In the flattening process each pattern body or pattern reference node is recursively merged to the closest ancestor pattern body and mapped to *flattened pattern bodies* (FPB). As a result, a *flattened call tree* is obtained in which the new flattened pattern bodies are direct children of the root pattern head node.

*Example 4.* The flattened version of the call tree of Fig. 3 is depicted in Fig. 7. The `classHasAttr` pattern has six flattened pattern bodies denoted by vectors, containing the numbers of the constituting body nodes. The flattened pattern bodies (`#2460`,

#2470, #2580, #2590 and #30) are recursive as depicted by grey square with circle, while flattened pattern body #1 is non-recursive.

For example the #2460 flattened pattern is constructed by starting from the root (disjunctive) node selecting the pattern body 2. From 2 the `classHasIncludedAttr` pattern head is traversed and the pattern body 4 is selected. The traversal continues on both branches of body 4 adding pattern body 6 and `classHasAttr` pattern call to the flattened pattern.

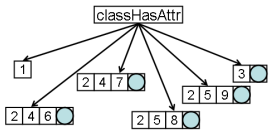


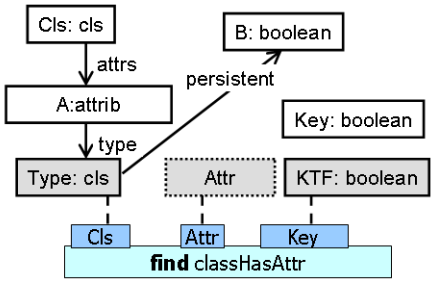

**Fig. 7.** Flattened patterns

### 4.3 Search Graph

Informally a *search graph* is a common representation of constraints (e.g., there is a relation between two elements) that drives the pattern matching process. For each flattened pattern body a separate *search graph* is generated, where a search graph is built by merging the constraint of the contained pattern bodies of a flattened pattern body, i.e., all formal parameters of the invoked pattern head are substituted with the corresponding actual parameters of the caller.

*Example 5.* For easier readability an extract of the search graph — without all constraints — of #2460 from Fig. 7 is depicted on the left side of Fig. 8, with the simplified VIATRA2 textual representation on the right hand side.

The search graph created from the combination of pattern bodies 2, 4, 6 and the pattern reference `classHasAttr` contains 7 entities all denoted as rectangles. `Relations` are captured by solid lines (e.g., `attrs` relation source is `Cls`), while binding between the actual and formal parameters of the recursive invocation are highlighted by dashed lines between the corresponding elements (e.g., `Type` is the actual parameter of the `Cls` formal parameter). While passing the formal parameter `Attr` of the caller pattern is denoted by a dotted box.




```
classHasAttr_2460(Cls,Attr,Key) ={
 //Local goal
 Boolean(Key);
 Boolean(KTF);
 Boolean(B);
 cls(Cls); cls.attrs(Cls,A);
 attrib(A); attrib.type(A,Type);
 cls(Type); cls.persistent(Type, B);
 check(value(Key) == "true" &&
 value(B) == "true" &&
 value(IKTF) == "true"); //TERM
 //Remote goal
 find classHasAttr(Cls,Attr,KTF); }
```

**Fig. 8.** Search graph of classHasAttr flattened pattern #2460

In order to present our concepts, we use an intuitive database like notation, where *search graphs* are defined as a set of natural joins over tables formed by the structural

and invocation constraints of the FPB, while check constraints are mapped to filters on matching candidates. Tables defined for entities and relations (structural constraints) are illustrated with tables of one and two columns, respectively, while pattern references and heads are captured by tables containing a column for each formal and actual parameter of the pattern. Note that, pattern references and heads of the same pattern are mapped to the same table.

This representation allows to define the matching process as a least-fix point evaluation ($tableofmatchings = lfp(structuralconstraints \bowtie patternreference)$) over the joined tables, where the $tableofmatchings$ holds the matchings of the invoked pattern head.

As a result this representation pin-points the crucial parts of recursive pattern matching, namely (i) optimized ordering of natural joins, and (ii) effective evaluation of least-fix points for which our solutions are introduced in Sec. 5.

*Example 6.* The extract database like representation of Fig. 8 is depicted in Fig. 9. Structured constraints (boxed in dashed line) are illustrated by tables of one and two columns, where the first row holds the type of the element, while the second represents the corresponding name of the involved search graph elements (e.g., the `attrs` table with two columns represent the `attrs` relation between the `Cls` and `A` entities). While the `classHasAttr` pattern recursive invocation is captured by a table of rows `Type`, `Attr`, and `KFT`. Finally, the search graph described as a least-fix point evaluation is $classHasAttr = lfp(structuralconstraints \bowtie classHasAtrr)$.

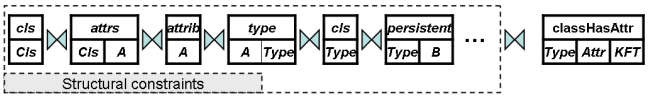



**Fig. 9.** Natural join representation of the classHasAttr pattern

## 5 Run-time behavior of the Recursive Pattern Matcher

After calculating and initializing the previous data structures at compile time, the rest of the recursive pattern matching process is carried out at run-time.

### 5.1 Ordering constraints of the flattened pattern body

When a pattern matching process is initiated for a given pattern at run-time, a user may supply input parameters. Depending on the binding of the formal parameters of the pattern head we define an *adornment* which denotes if the pattern parameter is `bound` (B) or `free` (F).

For an efficient query evaluation process, the execution order of natural joins should be determined by sequencing its constituting constraints. This sequence of constraints in a flattened pattern body is called a *search plan*, and it is produced by the algorithms of [8, 12], which also use the adornment information during the generation process.

### 5.2 Recursion evaluation techniques

Approaches for efficiently calculating the fix-point for the table of matchings can be categorized as follows.

The queue-based top-down recursion evaluation technique performs a breadth-first traversal for collecting matchings by alternately using the flattened call tree and navigating along pattern invocation constraints to explore the recursion in depth. As an advantage, this technique is able to focus only on exactly those "relevant" matchings that can provide solution for the actual binding of the pattern head at the topmost recursion level. On the other hand, as the matchings found in a deeper level of recursion are always immediately propagated upwards by performing a series of natural joins, this approach requires the proper maintenance of the pattern heads that have actually been invoked during the traversal including one local copy for their actual bindings and one for their matchings resulting in a decentralized solution.

The bottom-up recursion evaluation technique directly follows the fixpoint calculation approach, and in this sense, it iteratively extends one global table of matchings by repeatedly evaluating the query of each flattened pattern body. As a consequence, compared to the top-down approach, queries are executed fewer times and on larger blocks of data resulting in a faster solution. On the other hand, the bottom-up technique always calculates all matchings independently of the initial bindings, which unavoidably produces a table of matchings that is significantly larger than the final result set.

### 5.3 Magic sets

In order to preserve the fast and centralized bottom-up evaluation technique and to simultaneously minimize the gap between the number of calculated matchings and the size of the final result set, the concept of magic sets is introduced, which helps avoiding the generation (and temporary storage) of irrelevant matchings by restricting calculations only on such input parameters that might be produced during the actual pattern matching process.

For each pattern head, a *magic set (MS) table* is allocated, which stores such tuples of the bound parameters of the pattern head that have ever been passed downwards (i.e., to a deeper level of recursion) as input parameters during the evaluation. Note that the adornment (or binding pattern) of the pattern head determines, which columns must be contained by the magic set.

A *magic set transformation* is performed to introduce the MS table in the query calculation by placing it into the first (i.e., leftmost) position. Additionally, queries for extending the MS table are defined. As it is difficult to give a short and intuitive explanation for specifying these queries, the process of MS table extension is only exemplified in the current paper.

### 5.4 Execution

Recursive graph pattern matching is an iterative process, in which a fix-point is calculated for each MS table and each table of matchings.

Tuples can be classified based on the number of iterations passed since they got into a given table. Based on this categorization, tuples that joined just before the current iteration are called *recent*. All other tuples already contained by the tables are referred as *old*. Tuples being calculated in the current iteration are called *new*.

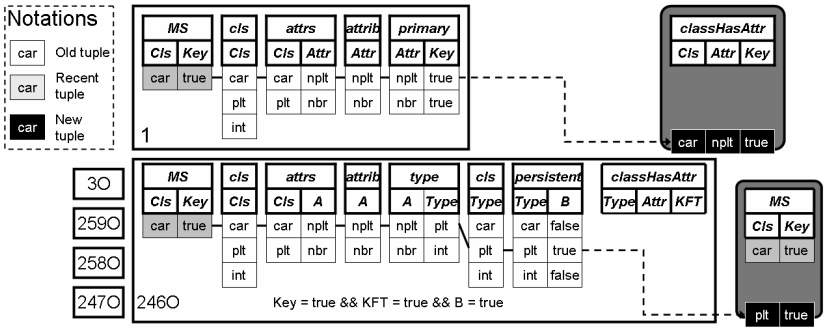
The exact process of fix-point calculation is as follows.

- **Initialization.** The table of matchings is initially empty, and the MS table is initialized with a single recent tuple containing the input parameters of the original pattern invocation.
- **Calculation tasks of each iteration.** In each iteration, all queries are executed once to possibly generate new tuples for the MS table and the table of matchings, which, in turn, represent new input parameters passed downwards and new matchings passed upwards, respectively. In order to avoid unnecessary recalculations on old tuples, only recent tuples of the MS table and the table of matchings are involved in the natural joins. The tuples calculated by the natural joins are filtered by check constraints. If all the constraints are fulfilled, then the result tuple is projected on the formal parameters of the pattern head, and scheduled to be added to the corresponding table as a new tuple.
- **Synchronization after each iteration.** Synchronization is performed after each iteration by an ageing process, which (i) keeps old tuples, (ii) makes all recent tuples old, (iii) collects new tuples from flattened pattern bodies, (iv) adds these new tuples to the corresponding table, if they are not yet contained, and (v) marks all the collected new tuples recent.
- **Termination.** Pattern matching is terminated when neither the MS table, nor the table of matchings is extended during an iteration. Based on analogy to [15], termination can be guaranteed, if negative application condition checks are invoked only with bound parameters, which is typically fulfilled in graph transformation approaches.
- **Postprocessing.** Finally, in a postprocessing phase, the table of matchings is filtered by checking whether the result tuples in the bound parameter positions match the input parameters passed at the original pattern invocation.

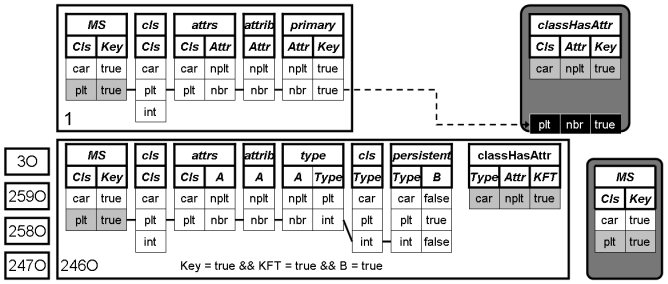
*Example 7.* The iterative pattern matching process is illustrated in Fig. 10. It calculates such matchings for pattern head `classhasAttr`, in which formal parameters `Cls` and `Key` are bound to `car` and `true`, respectively.

Each subfigure shows (i) the table of matchings for the pattern head `classhasAttr` in its top-right corner together with the corresponding `MS` table beneath, (ii) the detailed search plans of flattened pattern bodies `#1` and `#2460` in the middle, and (iii) the flattened pattern bodies (`#30`, `#2590`, `#2580` and `#2470`) not involved in the calculations on the left.

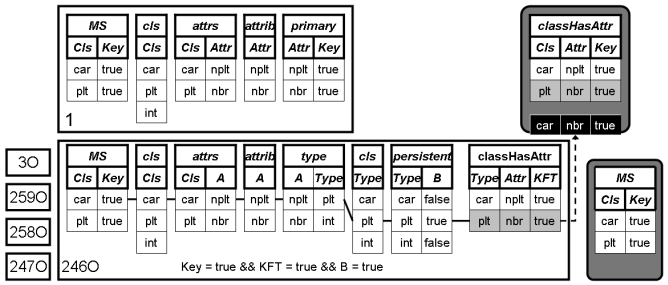
Fig. 10(a) illustrates the state of the runtime execution after the calculation tasks of the first iteration, during which (i) the `MS` table is initially loaded with recent tuple `(car,true)`, (ii) the query of non-recursive flattened pattern body `#1` is evaluated by natural joining all its tables to the recent tuple `(car,true)` of the `MS` table producing a new matching `(car,nplt,true)`, and (iii) a new tuple `(plt,true)` of input parameters to be passed downwards later is generated by calculating natural joins of



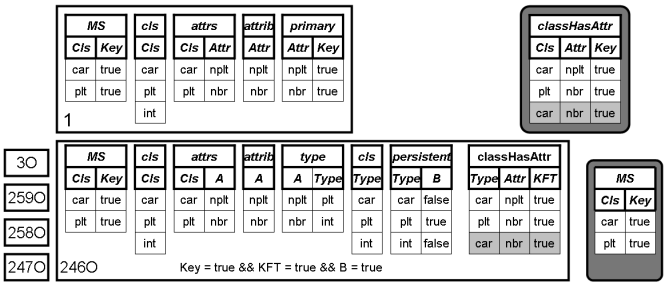

(a) First iteration




(b) Second iteration




(c) Third iteration




(d) Fourth iteration

**Fig. 10.** Runtime iterations of the classHasAttr pattern

tables up to (but excluding) the recursive invocation constraint in flattened pattern body `#2460`.

At the second iteration in Fig. 10(b), the previously matched tuples `(car,nplt,true)` and `(plt,true)` appear as recent tuples in the table of matchings and the `MS` table, respectively. In this iteration, the query of flattened pattern body `#1` produces a new matching `(plt,nbr,true)` for the pattern head `classHasAttr`, while queries of flattened pattern body `#2460` fail on checking constraints as the natural joins produce such results by starting from either recent tuple, in which the value in column `B` is false.

In the third iteration (shown by Fig. 10(c)), the table of matchings for pattern head `classHasAttr` is extended by a new matching `(plt,nbr,true)` produced by the query of flattened pattern body `#2460`, which uses the recent tuple `(plt,nbr,true)` for performing the natural joins.

The fixpoint calculation algorithm terminates after the fourth iteration (depicted in Fig. 10(d)) as neither the `MS` table, nor the table of matchings is further extended.

Finally, in the postprocessing phase, matching `(plt,nbr,true)` is filtered out as it does not have value `car` in its column `Cls` as it would have been required by the initial binding of input parameters. However, matchings `(car,nplt,true)` and the `(car,nbr,true)` remain in the final result set.

## 6 Related Work

The concept of recursion has already been used by several powerful, graph transformation related algorithms, tools, and approaches including [9], which presents valuable theoretical foundations of handling recursion in graph transformation. Since our approach focuses on the *implementation* of a pattern matching engine, only practical considerations are examined in the following.

Many advanced graph transformation tools support recursion in their control flow language (like GReAT [16] and VMTS [17]) or use it in the control structure implementation (like MOLA [18]). In all these approaches, recursion appears in the *imperative control flow part* of the graph transformation engine, in contrast to our approach, in which *fully declarative and recursive pattern* specifications are given to the *pattern matching module* as input.

In the following, only such pattern matchers are surveyed in the order of increasing expression power of their specification language, which are able to handle recursive patterns. PROGRES [4] and Fujaba [5] use the concept of path conditions and expressions, which can be considered as a form of recursion, as a path can define a set of connected edges of arbitrary length. Paths are computed only in forward direction in PROGRES, which may cause performance degradation when the end point of a path condition is fixed as reverse path navigation is not part of the otherwise, highly sophisticated search plan generation algorithm. The expression power of path conditions is strongly limited by their nature due to the fact that only linear graph structures are allowed to be repeated.

A recent paper [10] presents the concept of star regions for expressing repetitive graph structures, which can be considered as an alternative representation of recursion.

The authors provide a valuable and detailed description of their algorithm, which evaluates recursion in a top-down manner, in contrast to our approach, which performs bottom-up evaluation. Since arbitrary graph structures can be contained by star regions (undoubtedly providing support for any form of simple recursion) this indicates a more expressive language compared to the ones that only handle path conditions. However, e.g., mutual recursion is still an unsupported feature.

From a graph transformation point of view, the implementation of Tefkat [7] shows the largest similarity to our approach. Both are able (i) to handle complex forms of recursion (providing a stronger expression power compared to all the previous approaches), and (ii) to reorder terms (i.e., search plan constraints) for efficiency and on semantic correctness backgrounds. Tefkat uses the technique of top-down recursion evaluation with memoing, while our approach performs a magic set transformation followed by a bottom-up evaluation. Additionally, our approach provides support for flattening, which allows an inter-pattern search plan optimization for such patterns that can be evaluated by a single non-recursive pattern matching algorithm.

The technique of combining bottom-up evaluation with magic set transformation [11] is well-known in the knowledge-base system community for over a decade. This technique is intentionally used by our approach as several important theorems (including statements about algorithm termination) have already been proven. Arguments for preparing an own implementation include (i) the lack of support for flattening by any existing general-sense knowledge-base systems, and (ii) a vision to build further runtime optimizations by using graph pattern matching specific knowledge.

The popularity of recursive graph pattern matching has been demonstrated at the AGTIVE workshop by several contributions discussing its specification issues. [19] proposed query support for the DRAGOS graph database, and mentioned the handling of recursive queries by nested subgraph as future work. [20,21] examined different aspects of set-valued graph transformation by using the PROGRES tool. Note that these contributions can be considered as application domains for our approach as it (over)fulfills their specification criteria by providing a larger expression power.

## 7 Conclusion

In the current paper we proposed a pattern matching framework for matching recursive patterns by using fixpoint-based bottom-up query evaluation with top-down handling of input parameters. The essence of the matching process is to flatten non-recursive pattern compositions for global optimization and execute recursive invocations in an iterative manner by using magic set transformation.

Finally, it is worth pointing out that the proposed approach has been fully implemented, and it will be part of the upcoming VIATRA2 release.

## References

1. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. World Scientific (1997)

2. Kreowski, H.J., Kuske, S.: Graph transformation units and modules. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools. World Scientific, Singapore (1999) 607–638
3. Heckel, R., Ehrig, H., Engels, G., Taentzer, G.: Classification and comparison of module concepts for graph transformation systems (1999)
4. Zündorf, A.: Graph pattern-matching in PROGRES. In: Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science. Volume 1073 of LNCS., Springer-Verlag (1996) 454–468
5. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proc. of the 22nd International Conference on Software Engineering, ACM Press (2000) 742–745
6. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: Proc. of the 21st ACM Symposium on Applied Computing, Dijon, France, ACM Press (April 2006) 1280–1287
7. Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In Bézivin, J., Rumpe, B., Schürr, A., Tratt, L., eds.: Proc. of the International Workshop on Model Transformation in Practice (MTiP 2005). (October 3rd 2005)
8. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Proc. of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), Braga, Portugal, Electornic Communications of the EASST (March 31- Apr. 1 2007) 57–68
9. Guerra, E., de Lara, J.: Adding recursion to graph transformation. In: Proc. of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), Braga, Portugal, Electornic Communications of the EASST (March 31- Apr. 1 2007) 107–120
10. Lindqvist, J., Lundkvist, T., Porres, I.: A query language with the star operator. In: Proc. of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), Braga, Portugal, Electornic Communications of the EASST (March 31- Apr. 1 2007) 69–80
11. Ullman, J.D.: Principles of database and knowledge-base systems, Vol. II. Computer Science Press, Inc., New York, NY, USA (1989)
12. Varró, G., Varró, D., Friedl, K.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. In Karsai, G., Taentzer, G., eds.: Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of ENTCS., Tallinn, Estonia (September 2005) 191–205
13. Bézivin, J., Rumpe, B., Schürr, A., Tratt, L.: Challenge of the model transformations in practice workshop (October 3rd 2005)
14. Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varró, D., Varró-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005). (2005)
15. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies. Computer Science Press (1989)
16. Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G.: Reusable idioms and patterns in graph transformation languages. In Mens, T., Schürr, A., Taentzer, G., eds.: Proc. of the International Workshop on Graph-Based Tools. Volume 127 of ENTCS., Rome, Italy, Elsevier (October 2004) 181–192 `http://tfs.cs.tu-berlin.de/grabats/`.
17. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model transformation with a visual control flow language. International Journal of Computer Science **1**(1) (2006) 45–53

18. Kalnins, A., Celms, E., Sostaks, A.: Model transformation approach based on MOLA. In Bézivin, J., Rumpe, B., Schürr, A., Tratt, L., eds.: Proc. of the International Workshop on Model Transformation in Practice (MTiP 2005). (October 2005) http://sosym.dcs.kcl.ac.uk/events/mtip05/.
19. Weinell, E.: Adaptable support for queries and transformations for the DRAGOS graph-database. In Schürr, A., Nagl, M., Zündorf, A., eds.: Proc. of the 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance, Kassel, Germany (October 2007)
20. Fuss, C., Tuttlies, V.E.: Simulating set-valued transformations with algorithmic graph transformation languages. In Schürr, A., Nagl, M., Zündorf, A., eds.: Proc. of the 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance, Kassel, Germany (October 2007)
21. Körtgen, A.T.: Modeling successively connected repetitive subgraphs. In Schürr, A., Nagl, M., Zündorf, A., eds.: Proc. of the 3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance, Kassel, Germany (October 2007)