Electronic Communications of the EASST
Volume 50 (2011)

Recent Advances in Multi-paradigm Modeling (MPM 2011)

An Approach for Model Querying-by-Example Applied to Multi-Paradigm Models

Alek Radjenovic, Richard F. Paige

13 pages

Guest Editors: Vasco Amaral, Cécile Hardebolle, Hans Vangheluwe, László Lengyel, Peter Bunus
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

An Approach for Model Querying-by-Example Applied to Multi-Paradigm Models

Alek Radjenovic[1], **Richard F. Paige**[1]

Department of Computer Science, University of York, United Kingdom[1]

Abstract: Scenarios for industry-scale multi-paradigm modelling involve analysis, transformation, or fine-grained manipulation of models. These models are often treated wholly or in part as trees (e.g. XML or XMI documents, or source code). However, existing facilities for accessing and manipulating models as trees is limited. We present a novel approach to model querying-by-example, treating models as trees. The approach abstracts away from platform-specific concerns (e.g. XML), and exploits tree-based patterns in expressing queries; the results of queries are also trees, thus providing means to compose (conjoin) queries without requiring intermediate manipulations.

Keywords: trees, tree queries, tree patterns, model analysis, model querying-by-example, multi-paradigm modelling

1 Introduction

A majority of software specifications that are written using programming or modelling languages can be represented as trees. For example, all those languages whose syntax is specified using Backus–Naur Form (BNF) or XML Schema can readily express their documents in a tree–like form. Queries that are used to retrieve information from trees typically specify patterns of selection predicates (on multiple elements) that have some specified tree structured relationships. For example, a core operation for XML query processing is finding all occurrences of tree structured relationships (e.g. parent–child and ancestor–descendant) in an XML database.

In Model–Driven Engineering (MDE), queries on models are important in almost all of the activities related to model management, such as model analysis, comparison, differencing, or transformation. Model querying is vital in scenarios involving multi–paradigm modelling, where there is a requirement to support interoperation or integration between heterogeneous models. In such situations, queries on models allow us to isolate relevant information from heterogeneous models and provide the results as input to model management operations.

In many scenarios where models can be expressed as trees, the ability to query the models and *obtain query results* using tree structures is useful; however, many existing querying approaches produce query results that are *flat*. Queries that result in trees are beneficial: for providing a clear conceptual approach to querying; to allow composition/conjunction of queries (without having to introduce intermediate unflattening transformations); and to enable down–stream model management tasks (like system integration or code generation) that benefit from tree–based representations. This is especially valuable if the query results can preserve structural properties of the source models (e.g., containment relationships).

There are limitations in the existing querying approaches that treat models as trees:

- few abstract constructs for specifying queries and parameters can reduce the usability of model querying, though some approaches have proposed to raise the level of abstraction by introducing new operators (e.g. star [LLP07] and recursion [GL07])
- techniques in widespread use (e.g. XQuery [W3C10b]) generate data sets that are typically flat (items returned are of one particular type, e.g. books in a library); obtaining structural information from the query results is generally not straightforward
- in some approaches, it is possible to include structural information in the result, but its size increases significantly even if only a small amount of such information is needed
- in others, structural information can be obtained by additional processing (e.g. using axis steps in [W3C10a]), but such approaches are more performance–demanding. Additionally, the queries are usually platform–specific or do not allow for interrogation of trees if their underlying structure (schema) is not known beforehand)
- the scalability of existing approaches to very large models and thus complex scenarios of multi–paradigm modelling (such as system integration involving heterogeneous models) is uncertain.

To the best of our knowledge an approach addressing all of the limitations described above does not exist. A reason for this may be that existing model manipulation methods are tightly coupled with the metamodels for individual languages, making them hard to apply in a multi–paradigm modelling scenario.

Addressing this gap, we propose a solution that defines a basis for a practical model query–by–example approach. The approach is metamodel agnostic, and thus suitable for model querying of heterogeneous models in multi–paradigm modelling scenarios. We have tested the approach in a number of system integration scenarios involving large models in languages like UML, various DSLs (Domain Specific Languages) , XMI, Java, SQL and others.

The key components of our approach are: (i) a common interchange format, (ii) a tree query metamodel, (iii) a query results metamodel, and (iv) an algorithm. In this paper we present the first three components in full detail. We also provide an outline of the algorithm.

The rest of this paper is organized as follows. In section 2, a small example is shown to motivate the problem. In section 3, we review the related work. In section 4, we present our approach. In section 5, we outline a scenario to show how our approach can be used in multi–paradigm modelling. In section 6, we make conclusions and suggest future directions.

2 Example

In this section we present a small example to further motivate the problem. We use an example XML file (Figure 1) as the source of data, and XQuery snippets to demonstrate the extraction of items of interest from the source file. The source file represents a small library of books in the form of an XML database. The library has two sections (Classics and Children) each containing a single book. One book is first organized in parts, then in chapters, whilst the other has only chapters (no parts). Many of the elements are further decorated with attributes (e.g. each book

```
1  <?xml version ="1.0" encoding="UTF-8" ?>
2  <library>
3    <section name="Classics">
4      <book name="Ulysses" author="James Joyce">
5        <details publisher="... />
6        <contents>
7          <part name="The Telemachiad" number="I">
8            <chapter number="1" name="Telemachus"/>
9            <chapter number="2" name="Nestor"/>
10           <chapter number="3" name="Proteus"/>
11         </part>
12         <part name="The Odyssey" number="II">
13           <chapter number="4" name="Calypso"/>
14           ...
15         </part>
16         ...
17       </contents>
18     </book>
19   </section>
20   <section name="Children">
21     <book name="Alice's Adventures..." author="Lewis Carroll">
22       <details publisher="... />
23       <contents>
24         <chapter number="1" name="Down the Rabbit Hole"/>
25         <chapter number="2" name="The Pool of Tears"/>
26         <chapter number="3" name="A Caucus-Race and..."/>
27         ...
28       </contents>
29     </book>
30   </section>
31 <library>
```

Figure 1: An example XML file.

```
<library>
   <section name="Classics">
      <book name="Ulysses" author="James Joyce">
         <contents>
            <part name="The Telemachiad" number="I">
               <chapter number="1" name="Telemachus"/>

<library>
   <section name="Childrens">
      <book name="Alice's Adventures in Wonderland" author="Lewis Carroll">
         <contents>
            <chapter number="1" name="Down the Rabbit-Hole"/>
```

Figure 2: XQuery results using the ancestor–or–self axis.

element has a name and an author attribute). The database is organized as a tree (ignoring the standard XML declaration on line 1). The root of the tree is the *library* element. All other elements are contained inside some other element. The *leaf* nodes do not contain any other elements. We recognize two types of containment relationships: *direct* (or, parent–child) and *indirect* (ancestor–descendant). For example, *library–section* and *book–details* are examples of the former, while *library–book* and *book–chapter* are examples of the latter. In addition, each parent–child relationship is also a special case of the ancestor–descendant relationship.

In XPath, the special symbols '/' and '//' are used to signify the parent–child and ancestor–descendant relationships, respectively. XQuery is built on XPath expressions and we can use XQuery to search for specific items in the XML tree. For example, the XQuery path expression `library/section` returns the following result:

```
<section name="Classics">
<section name="Childrens">
```

Another query, that uses the '//' symbol, `book//chapter[number = '1']`, returns:

```
<chapter number="1" name="Telemachus"/>
<chapter number="1" name="Down the Rabbit--Hole"/>
```

We observe several things from these examples. Firstly, although two kinds of elements are specified in each of the queries (*library* and *section* in the first, *book* and *chapter* in the second), the query results contain only the elements of the rightmost type. Secondly, the results are flat and structural relationships are not immediately obvious. XPath does provide a mechanism to interrogate the returned items using the concept of *axes*. An axis defines a node set relative to the current node, and there are a number of pre–named axes. In particular, using the *ancestor–or–self* axis we can select all ancestors of the current node as well as the current node itself. However, these sets of ancestors may contain nodes not included in the original query. For instance, if we were to use axes in the second example (above), we would get the two sub–trees shown in

Figure 2. Finally, there is a substantial effort required to generate results shown in Figure 2 using the axes approach, and a further effort to reduce the sub–trees in the figure to contain only the element types specified in the query (i.e. *book* and *chapter*).

3 Related Work

A common approach to querying trees is XQuery [W3C10b]. XQuery is a W3C recommendation, is built on the XPath [W3C10a] expressions, and is often regarded in relation to XML as the equivalent of SQL [Bea09] to databases. It is the language for finding and extracting elements and attributes from XML documents. Its advantages include its ability to query XML data and the ease with which one can learn it. There have been attempts to enhance the XQuery usability (e.g. [BCC05], but the key limitation, other than those discussed earlier, is that its mechanisms for dynamic binding and query composition are cumbersome.

Pattern matching in trees has received much research attention. The main focus has been on traditional databases and performance aspects, such as the efficiency of matching algorithms [AJK+02][BKS02][CLT+06][CJLP03][GKP05][HO82][JAC+02][Kos89]. For example, Hoffman and O'Donnell [HO82] introduce new techniques for pattern matching, analyzed for time and space complexity. Chen et al. [CLT+06] work on minimizing intermediate results, while in [CJLP03] the authors demonstrate how their plans based on generalized tree patterns significantly outperform straightforward plans and plans based on navigation. Kosaraju [Kos89] reports improvement in pattern matching time by extending convolutions and suffix trees, and by partitioning trees into chains and anti-chains. Sarkar's work on tree pattern matching in source code [SSB01] aims to produce an optimized compiler that performs retargetable object code generation.

There is also an abundance of work specific to querying XML data. Amer–Yahia et al. [ACS02] suggest techniques for approximate XML query matching which reportedly outperform rewriting–based and post–pruning strategies. Al–Khalifa et al. [AJK+02] argue that complex query tree patterns can be decomposed into a set of primitive relationships between pairs of nodes. The query pattern can then be matched by (i) matching each of the binary structural relationships against the XML database, and (ii) 'stitching' together these basic matches. Bruno et al. [BKS02] propose a technique that uses a chain of linked stacks to compactly represent partial results to root–to–leaf query paths. These are then composed to obtain matches for the specified patterns.

From a mathematical perspective, there is research on conjunctive queries over trees, sometimes referring to incomplete XML trees (a.k.a. tree patterns), and at other times talking about data exchange and integration (i.e. transforming tree patterns). Gottlob et al. [GKS04] present a detailed study on the tractability of conjunctive queries over trees. They define a dichotomy based on a set of axes and demonstrate that these queries are tractable if they use a subset of: {Child, NextSibling, NextSibling*, NextSibling+}, {Child+, Child*} or {Following}. For each other set of axes, the combined complexity of the query evaluation problem is NP–complete. Bjorklund et al. [BMS07] investigate containment of conjunctive queries over trees considering tree patterns without data value comparisons (unlike our approach). They found that the containment $P \subseteq Q$ holds if there is a homomorphism from the canonical database of Q to the canonical

database of P, and that such a homomorphism is a sufficient, but not a necessary condition for containment (in relational databases it is a necessary condition). The results were encouraging, as the complexities (compared with acyclic queries) did not increase much. Barcelo et al. in [BLPS10] aim to provide a classification of problems associated with incompleteness for XML documents. They prove a strong generalization of polynomial–time algorithms for XML cases by addressing standard problems of consistency (does an incomplete database have a completion satisfying some conditions) or query answering (asking, for instance, whether a given tuple of values is a certain answer, i.e. an answer that is independent the way in which the missing parts of patterns are interpreted). Arenas and Libkin [AL05] use declarative schema formalisms to define formal semantics for data exchange between XML schemas, and providing the basis for extensions dealing with rewritability, query answering, or schema composition.

In the MDE space there has been substantial work on the QVT standard for model querying and transformation, and implementations of QVT and QVT–like languages have appeared. Constraint languages such as OCL have also been used for model querying, though these are targeted at models that comply with the OMGs metamodelling infrastructure.

Finally, there have been attempts to deal with recursion in pattern matching. Lindqvist et al. [LLP07] propose a star operator, similar to that in Kleene algebra, for a model query language. This is a pattern based approach that can be used to represent recursive or hierarchical structures in models. The approach is based on rule schemas with sub–graphs that can be cloned or copied before applying the rule. Varro et al. [VHV08] present core data structures and algorithms for matching graph patterns with general recursion. Although the approach is limited to recursive sub–patterns, a highly–sophisticated local–search based graph pattern matching allows the tool (VIATRA2) to scale up to very large models. Körtgen [KÖ8] develops a Loop–construct to express repetitive structures, and introduces a concept of a general region that could be used for further extensions like recursive transformation rules.

4 Approach

The key objective of our work was to provide a practical mechanism for querying models (represented as trees) by example. Specifically, a query that is in the form of a tree–like pattern should produce a result in the same or similar structure as the pattern itself. A logical conjecture which follows is that it is highly desirable that the query result: (a) includes the structural relationships from the source tree, (b) is in a form of a tree, and (c) provides this hierarchical association in an obvious (i.e. easily accessible) manner. This is so that structure can be directly exploited in down–stream activities, and so we do not have to manage flat query results (which could be extremely large for complex queries on large models). In other words, we desire queries that are compositional.

We can see that the second query pattern in the previous section is a tree: a node of type *book* at the root of the tree, and a descendant node of type *chapter*. It is beneficial that the result is in the same (tree) form. Consequently, instead of having the results in a list form as shown earlier, or in a cluttered tree form as in Figure 2, we would get them as a set (or a sequence) of trees shown in Figure 3 that closely resemble the query pattern.

This can be achieved in XQuery as follows:

```
<book name="Ulysses" author="James Joyce">
  <chapter number="1" name="Telemachus"/>
</book>

<book name="Alice's Adventures in Wonderland" author="Lewis Carroll">
  <chapter number="1" name="Down the Rabbit Hole"/>
</book>
```

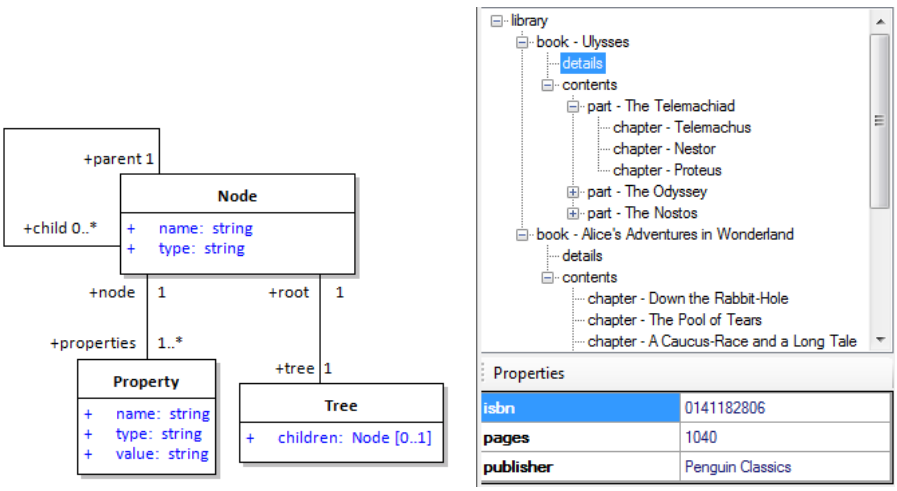
Figure 3: Query results that resemble the query pattern.

```
for $b in $input//book, $c in $b//chapter
where $c/@number = "1"
return element book { $b/@*, $c }
```

However, the limitations mentioned in Section 1 are still valid: besides the platform specificity, XQuery does not scale up well and for a more detailed search in large input trees, queries quickly become more elaborate whilst the processing time is substantially increased.

We now explain in more detail the four components of our method.

We define a *subject tree* simply as a tree–like representation of an input model. This representation is the well–known *rooted tree* from graph theory [Wik11]. The vertices (nodes) represent the structural model elements, whilst the edges represent the containment relationship. Nodes are of two types: *normal* and *reference* (to support cross–referencing within a model and across different models). In addition, each node may have a set of one or more properties attached to it. A property has three attributes: the type, the name and the value. Figure 4 depicts the subject tree meta–model in UML notation on the left, and a tree example on the right. This meta–model represents the interchange format which allows us to operate on many commonly used specifications (UML, Java, SQL, etc.) in a consistent fashion.



(a) **(b)**

Figure 4: Subject tree: meta–model (a), and an example (b).

A *tree query* consists of two parts: the mandatory *pattern* and an optional set of *matching*

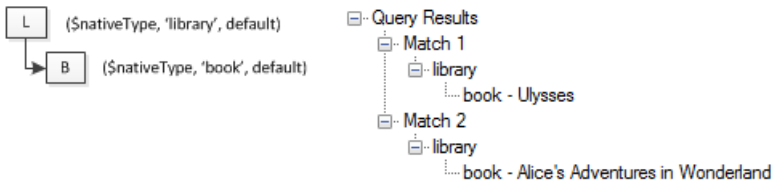


Figure 5: A simple tree query with a pattern that uses only normal nodes (left) and its result set (right).

conditions. The patterns are trees in a similar sense as the subject trees, with a few differences.

Firstly, *two* kinds of nodes can be present in a pattern: *normal* nodes and *wildcard* nodes (explained by example shortly). Here, 3 constraints related to the structure of the pattern must hold: (i) the root must always be a normal node, (ii) a wildcard must have children, and (iii) the children of a wildcard node must all be normal nodes. Secondly, pattern nodes must be named, i.e. they have a single mandatory attribute called *name* which is of the string type. And thirdly, pattern nodes do not have a set of properties associated with them. Instead, normal pattern nodes (only) typically have matching conditions attached.

A matching condition comprises three elements: property name, property value, and a compare method. During the execution of a tree query, the matching algorithm parses the set of properties of the current node in the subject tree, comparing the property names and their values with corresponding elements in the matching condition using the specified *compare method*. When a match is found, this method returns a Boolean *true* value.

We now describe how a pattern containing only the normal nodes is matched against a subject tree. A simple query is shown on the left of Figure 5. The pattern consists of two nodes (L and B). The L node has a single match condition which, in simple words, looks for nodes in the subject tree whose native type is *library*. Similarly, the B node looks for books. Running the query against the subject tree in Figure 4, returns the result shown on the right of Figure 5. If we wanted to query the tree from Figure 4 to return all the book chapters, but that each returned chapter is associated with the book it belongs to, it would be impossible to achieve it with a single query that has only the normal nodes. This is because the chapter elements are placed in different position within the tree. The Ulysses book is divided in parts, then in chapters. The second book has no parts – only chapters. We would have to use separate queries as shown in Figure 6: the query on the left for the Alice book, and the query on the right for the Ulysses book. Moreover, the query results would consist of sub–trees that include the nodes which lie in between the required ones (e.g. *contents* and *part* nodes shown in Figure 7. It is not difficult to see that the resulting sub–trees can be substantial in size and almost as equally hard to study as the original subject tree.

The introduction of the wildcard nodes in the pattern resolves this problem. A wildcard node can be regarded as a 'sink' or a 'collector' for the intermediate nodes that are typically not relevant (desired), and are typically (but do not have to be) excluded from the query results. For instance, the pattern in Figure 8 that searches for books and chapters is sufficient to parse the entire subject tree from Figure 4 in a consistent fashion (the wildcard node is shown as a star). This means that all the matched sub–trees (unlike those in Figure 7) have the same structure

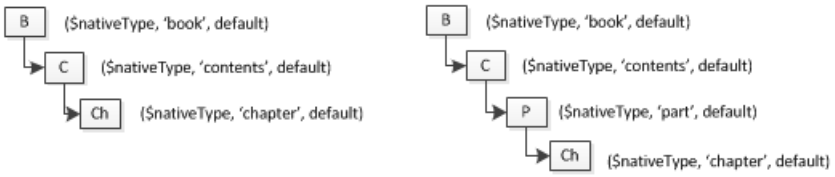


Figure 6: Queries needed to find chapters (normal nodes only).

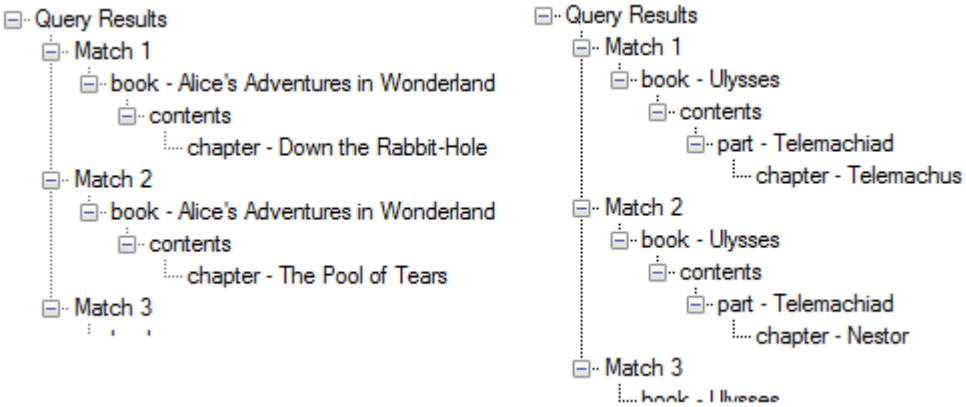


Figure 7: Query results for the chapter search shown on the left of Figure 6.

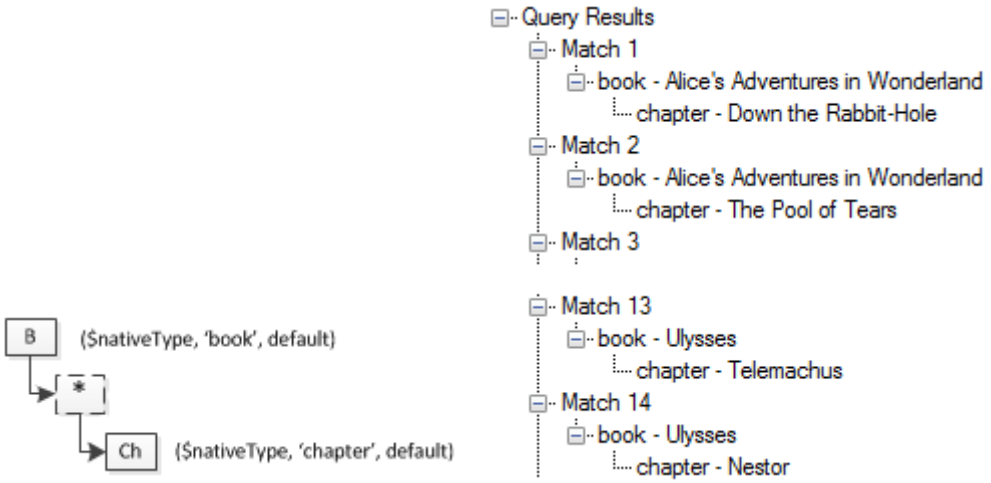
(Figure 8). Importantly, the intermediate nodes are not present in the result set.

There are two minor differences between the definitions of the normal and the wildcard nodes. Firstly, the wildcard nodes do not have to be named (the name attribute can be an empty string). And secondly, the wildcard nodes have an additional attribute (of Boolean type) called *hide* with default value equal *true*. This means that, by default, the intermediate nodes (like the *content* and *part* nodes from Figure 7) will not be part of the matched sub–trees in the result set. Graphically, this is shown by a dashed line that outlines a wildcard node (Figure 8). (if *hide* is *false*, the line is solid and the intermediate nodes are included in the result set).

Finally, each `PatternNodeInstance` has its own solution. This is best illustrated by example. In Figure 10, (a) represents a tree query pattern that we are trying to find in the subject tree (b), and the solution is represented in (c). For simplicity, we assume that all nodes in (b) are of the same type, and that the matching conditions associated with the pattern nodes in (a) require that A, B, C and D can match any node type in the subject tree. In other words, we are really trying to find the node structure specified by the pattern within the subject tree (b). It is pretty obvious that there are only two matches as shown in (c).

We can now relate the 3 types from Figure 9(b) (Solution, Image and PatternNodeInstance) to the nodes shown in the previous tree examples. For instance, in Figure 8, the 'Query Results' corresponds to the Solution, 'Match X' (X=1, 2, 3,) corresponds to the Image, and the children of the 'Match' nodes are of the PatternNodeInstance type.

A pattern is a tree structure with a single root node and zero or more other nodes. However, each pattern node with all of its descendants can also be described in the same way, i.e. a pattern



(a) **(b)**

Figure 8: A query with a wildcard (a) used against the tree in Figure 4, and the result set (b).

node with its descendants is just another pattern. Consequently, each has its own solution.

Next, we provide a high level overview of the algorithm used to obtain the query results.

The algorithm uses recursion, is designed to traverse the subject tree only once, and does not return non–injective matches. The search starts by calling Patterns Search method, passing a reference to the subject tree to be searched (Figure 11). The Contract() method (line 8) ensures that any wildcard nodes with their property hide set to true are invisible in the solution (the meaning of the word 'contract' here is that of the verbs 'shrink' or 'reduce'). The FindSolution() method is called next. The parameter required by this method is a set of nodes that need to be searched (along with all of their descendants). Although a subject tree only has one immediate child (the tree root), the InstanceTree class implements this as a set of InstanceTreeNode objects to make it consistent with the implementation of the Children property in InstanceTreeNode, and to enable a single implementation of the FindSolution() method. The FindSolution() method is recursive; its body consists of a single loop that iterates through the set of instance nodes (method's sole parameter). Here, the method performs two key steps. Firstly, it attempts to match the current instance node to the pattern's root. And secondly, it continues iterating through the current instance node's children (by recursively calling itself) and thus ensuring navigation of the entire subject tree. The code listed in Figure 11 traverses the entire subject tree, node by node, and attempts to find all matches for the pattern's root node. To use an example, this would be the equivalent of trying to match the root node B from Figure 6 (either pattern – left or right) with the nodes of the subject tree from Figure 4, i.e. finding all the books. If a match is found, then the code on lines 18 through to 26 (Figure 11) is executed. On line 19, a PatternInstanceNode object is created, pairing the pattern's root node with a subject tree node that matches it. The constructor for the PatternInstanceNode class immediately searches for a solution. If a solution is found for the PatternInstanceNode object (line 20 in Figure 11), it is then encapsulated inside

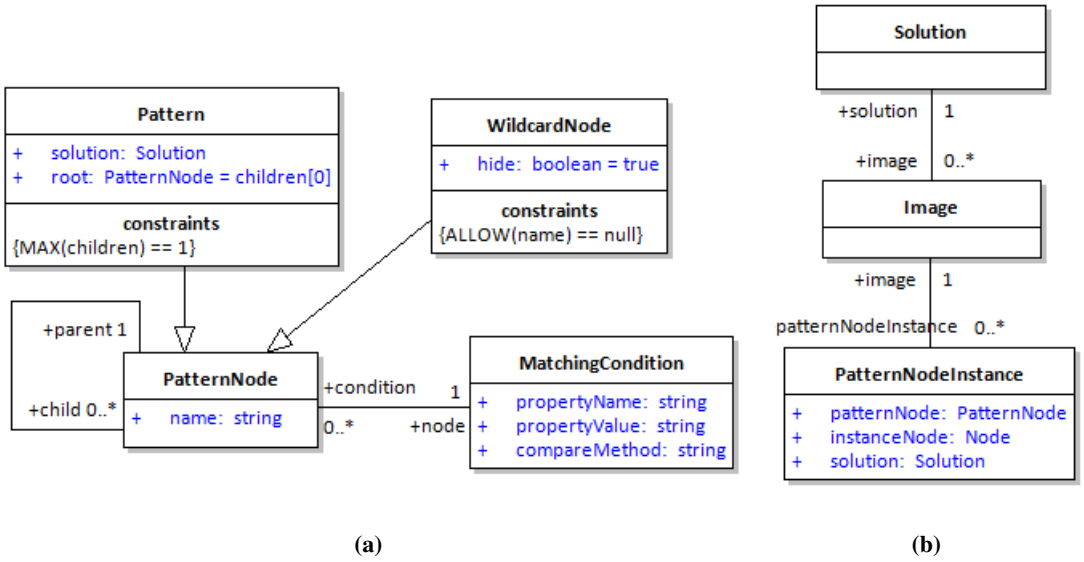


Figure 9: Tree query meta–model (a) and the query result data model (b).

an Image object and added to the overall solution (lines 22–24).

5 Prototypical Scenarios

Space limits prohibit a detailed example of the approach. We have applied the model querying approach extensively, particularly for system integration scenarios where we needed to first carry out matching (via querying) of heterogeneous system models expressed in multiple dialects of UML (e.g., legacy version 1.x models) supported by different tools (e.g., obsolete versions of Rational Rose). Another scenario involved using queries in a model management workflow, featuring *analysis of an input model* and then *transformation* to a different model. The input

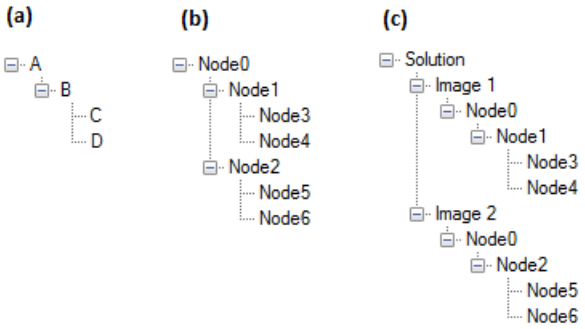


Figure 10: Example: (a) a pattern, (b) a subject tree, and (c) the solution.

```
CLASS Pattern (inherites from PatternNode)
BEGIN

  VAR root : PatternNode
  VAR solution: Solution

  FUNCTION Search(instanceTree)
    Contract()
    FindSolution(instanceTree.Children)

  FUNCTION FindSolution (instanceNodes)
    VAR pni : PatternNodeInstance
    VAR img : Image

    FOREACH instanceNode IN instanceNodes
    BEGIN
      IF (root.Match(instanceNode))
      BEGIN
        pni = NEW PatternNodeInstance(root, instanceNode)
        IF (pni.HasSolution()
        BEGIN
          img = NEW Image()
          img.Add(pni)
          solution.Add(img)
        END
      END
      FindSolution(instanceNode.Children)
    END
END
```

Figure 11: Pattern class: search algorithm (top level)

model was a UML model consisting of multiple class diagrams and state machines expressed in UML 1.x (legacy models created for a system engineering project). The output model was expressed in a DSL (annotated behavioural models for simulation). We first used the model querying to extract information from the UML class and state machine diagrams sufficient for simulation; the results of this (in the form of trees) was then passed to a model transformation that produced models in the DSL, optimised for simulation. The input models consisted of large numbers of model elements, and the tree patterns for querying consisted of tens of elements; the query results were produced effectively instantaneously.

6 Conclusions and Future Directions

Querying is a critical component of model management, as evidenced by standards and languages such as QVT, OCL and XQuery. A wealth of querying techniques that treat models as trees have been proposed, but the limitations of these approaches which make it difficult to extract structural information from query results, which are not based on example, and which are often platform–specific - need to be addressed for multi–paradigm modelling and model management in the large. Overall, we have argued that current model querying techniques generate results that are typically flat, and that a research gap lies in providing mechanisms that allow us to obtain structural information from the result set in a straightforward manner.

We have presented a solution that consists of four components: a common interchange format, a tree query meta–model, a query results meta–model, and an algorithm. We argued that various model management techniques and activities, such as model analysis, model transformation and

system integration, could benefit greatly from our approach.

At the moment, our approach is implemented in a prototype tool and we use graphical notation for specifying queries and patterns. Our near–term objectives are to provide external interfaces (in the form of an API) to the query engine, and provide a concrete textual notation akin to XPath, in order to support diverse users and diverse model management scenarios.

Bibliography

[ACS02] S. Amer-Yahia, S. Cho, D. Srivastava. Tree Pattern Relaxation. In *8th International Conference on Advances in Database Technology (EDBT 2002)*. Pp. 496–513. Springer Berlin / Heidelberg, Prague, Czech Republic, 2002.

[AJK+02] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, Y. Wu. Structural Joins : A Primitive for Efficient XML Query Pattern Matching. In *18th International Conference on Data Engineering*. P. 12. 2002.

[AL05] M. Arenas, L. Libkin. XML Data Exchange : Consistency and Query Answering. In *24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '05)*. Pp. 13–24. ACM New York, Baltimore, MD, 2005.

[BCC05] D. Braga, A. Campi, S. Ceri. XQBE (XQuery By Example): A Visual Interface to the Standard XML Query Language. *ACM Transactions on Database Systems (TODS)* 30(2):398–443, 2005.

[Bea09] A. Beaulieu. *Learning SQL. 2nd ed.* O'Reilly Media, 2009.

[BKS02] N. Bruno, N. Koudas, D. Srivastava. Holistic Twig Joins. In *2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. P. 12. ACM Press, New York, New York, USA, 2002.

[BLPS10] P. Barceló, L. Libkin, A. Poggi, C. Sirangelo. XML with incomplete information. *Journal of the ACM* 58(1):1–62, 2010.

[BMS07] H. Björklund, W. Martens, T. Schwentick. Conjunctive Query Containment over Trees. In *11th International Conference on Database Programming Languages (DBPL '07)*. Volume 77(3), pp. 66–80. Springer Berlin / Heidelberg, 2007.

[CJLP03] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, S. Paparizos. From tree patterns to generalized tree patterns: on efficient evaluation of XQuery. In *29th International Conference on Very Large Data Bases (VLDB'03)*. Pp. 237–248. VLDB Endowment, 2003.

[CLT+06] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, K. S. Candan. Twig 2 Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *32nd International Conference on Very Large Data Bases (VLDB'06)*. Pp. 283–294. VLDB Endowment, Seoul, Korea, 2006.

[GKP05] G. Gottlob, C. Koch, R. Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems* 30(2):444–491, 2005.

[GKS04] G. Gottlob, C. Koch, K. U. Schulz. Conjunctive queries over trees. In *23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. Volume 53(2), pp. 180–200. ACM New York, Paris, France, 2004.

[GL07] E. Guerra, J. de Lara. Adding Recursion to Graph Transformation. In *6th International Workshop on Graph Transformation and Visual Modeling Techniques*. Volume 6. Electronic Communications of the EASST, Braga, Portugal, 2007.

[HO82] C. M. Hoffmann, M. J. O'Donnell. Pattern Matching in Trees. *Journal of the ACM* 29(1):68–95, 1982.

[JAC+02] H. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, C. Yu. TIMBER: A native XML database. *The International Journal on Very Large Data Bases (VLDB)* 11(4):274–291, 2002.

[KÖ8] A.-T. Körtgen. Modeling Successively Connected Repetitive Subgraphs. In *3rd International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*. Pp. 426–441. Springer Berlin / Heidelberg, Kassel, Germany, 2008.

[Kos89] S. Kosaraju. Efficient tree pattern matching. In *30th Annual Symposium on Foundations of Computer Science (FOCS 1989)*. Pp. 178–183. IEEE, 1989.

[LLP07] J. Lindqvist, T. Lundkvist, I. Porres. A Query Language With the Star Operator. In Ehrig and Giese (eds.), *6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*. Pp. 69–80. Electronic Communications of the EASST, Braga, Portugal, 2007.

[SSB01] V. Sarkar, M. J. Serrano, B. Bluestein Simons. Retargeting Optimized Code by Matching Tree Patterns in Directed Acyclic Graphs. 2001.

[VHV08] G. Varró, A. Horváth, D. Varró. Recursive Graph Pattern Matching (With Magic Sets and Global Search Plans). In *3rd International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*. Volume 67651, pp. 456–470. Springer Berlin / Heidelberg, Kassel, Germany, 2008.

[W3C10a] W3C. XML Path Language (XPath) 2.0 (Second Edition). Technical report, 2010. http://www.w3.org/TR/xpath20/.

[W3C10b] W3C. XQuery 1.0: An XML Query Language (Second Edition). Technical report, 2010. http://www.w3.org/TR/xquery/.

[Wik11] Wikipedia. Tree (graph theory). 2011. http://en.wikipedia.org/wiki/Tree_(graph_theory).