Designing and Applying a Framework for Logic-Based Model Querying

Patrick Dohrmann and Sebastian Herold
Department of Informatics – Software Systems Engineering
Clausthal University of Technology,
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
Email: {patrick.dohrmann,sebastian.herold}@tu-clausthal.de

Abstract—Querying models is one of the most essential and most elementary tasks in model-based software development. More complex activities like, for instance finding source patterns of model transformations, measuring models, or checking consistency between models, include querying models for certain properties, elements, or substructures.

Logic formalisms like full first-predicate logic or description logics provide the well-understood foundation for implementing efficient model querying mechanisms. Regarding the specific purpose of querying models, a more efficient but less expressive logic formalism might be more useful than in other use cases.

In this paper, we will introduce a framework which enables us to easily realize meta model independent query tools based on different subsets of first-order logic. We show the application of the framework by checking a UML design model for architectural properties.

Keywords—Model-Based Software Development, Model Querying, Logic-Based Querying

I. INTRODUCTION

Models are the central items in model-based software development (MBSD). They describe software systems at different levels of abstraction during requirements elicitation, software design, implementation, and maintenance [1]. Beside the manipulation of models by adding, deleting, and editing model elements, querying models is one of the most essential and elementary tasks in MBSD.

Querying models to retrieve properties, elements, or substructures from them is required in several more complex activities during MBSD. For instance, searching in a model for places where a model transformation rule can be applied means querying the model for the substructures where the source pattern of the rule matches. Further important applications include checking compliance or consistency between models. For example, a software architect might want to check whether the structures specified in a design model are compliant with the intended and separately modelled software architecture.

Among other approaches, one way to implement model queries is to use approaches based on logic formalisms like first-order logic. Models are formalized as a set of facts. Querying the model corresponds to evaluating solutions for a formula representing the query in that set of facts. Logic knowledge representation or programming systems like Prolog provide inference algorithms to compute such solutions [2].

The advantages of logic-based approaches are diverse. First, they are meta model independent - the same mechanism of executing a query can be used for arbitrary modelling languages. Second, the theoretical foundations are well-understood which, on the one hand, lead to very efficient implementations of inference algorithms over the years. On the other hand, the need for efficient solutions has lead to subsets of logical systems that are less expressive but better to handle w.r.t complexity and decidability, like for instance description logics [3]. Thus, logic-based approaches are flexible w.r.t. the requirements of the specific purpose of model querying. Last but not least, the well elaborated foundations of logics have lead to a broad range of sophisticated tool support.

However, most logic-based model query approaches are implemented as tools for a fixed logical programming system. To benefit from a more efficient or a more suitable programming system by changing the underlying logic is not easy, since large parts of the implementation of the tool under consideration have to be adopted.

In this paper, we introduce a framework which allows us to realize logic-based query tools more efficiently. It uses an intermediate representation of first-order logic to generate predicates for arbitrary meta models and facts for according models. Language-specific back-ends, which implement a common interface defined by the framework, generate code for the particular logic programming system. Thus, changing the logical programming system for a query tool just means to change the selected back-end. To support a new logical programming system just means to implement the back-end interface for it. The framework allows creating meta-model independent query tools with less effort.

The remaining paper is structured as follows. The following section will introduce the framework for logic-based model querying. Section III will demonstrate the use of tools based upon the framework to check the architectural compliance of a UML design model. Section IV takes a look upon related work, section V concludes the paper.

Author’s version
II. CONCEPTS OF THE FRAMEWORK FOR LOGIC-BASED QUERYING

This section describes foundations, functionalities and concepts of the framework. The next section will shortly describe some general foundations of logic knowledge representation and programming systems. Section II-B will give a quick overview of the developed framework. This overview will be detailed in the subsequent sections that deal with single components and concepts of the framework.

A. Foundations of Logic Knowledge Representation and Programming Systems

A logic program consists of a knowledge base and queries that retrieve information from it. A knowledge base contains facts that describe properties of entities of a universe of discourse or relations between them by expressing that certain predicates hold for them. Consider the knowledge base depicted in Fig. 1. It describes several entities to be cars or persons by the included facts using the predicate car(<entity>) and person(<entity>) and additional attributes and relations between them. Beside explicit facts, the knowledge base also contains rules that describe how new knowledge can be concluded from existing facts. The rule contained in the knowledge base from Fig. 1 (line 4) states for example, that we can conclude from the fact that an entity is a person that owns a car, expressed by the rule body, the right part of the rule, that this person has to have a driver’s license. This is defined by the left part of the rule hasDriversLicense, the rule head. A knowledge base like this can be queried, for example, to find all persons owning a powerful car (as depicted in Fig. 1).

It is obvious that a model can be represented as a logic knowledge base, too. The entities of a model and their attributes and relations have to be represented as logical facts using a defined set of predicates and rules. This set may differ from meta model to meta model, thus, it cannot be defined in advance. The framework described in the following sections is able to transform a given meta model into a set of defined predicates and rules, and to transform a specific model according to the predicates derived from its meta model. This is done independently from the specific logic programming language system.

B. The Framework at a Glance

The framework is based on the Eclipse Modeling Framework (EMF)[6]. Its inputs are models and meta models represented in XMI [7]. The framework consists of the following components as depicted in Fig. 2:

- **Meta Model Transformer**: This component accepts a meta model and transforms its content into a set of predicates, rules, and types. Types are supported, since some logic programming systems support typing of entities.
- **Intermediate Logic Model**: Predicates, rules, and types returned from the Meta model transformer are represented as intermediate logic model that is independent from the specific logic programming system.
- **Model Transformer**: This component realises the transformation of a specific model. It uses the intermediate logic model created from the model’s meta model and transforms it into a set of facts according to the intermediate logic model representing the meta model.
- **Backend Interface**: Its task is to provide a common interface for logic programming language specific code generation. Implementations of the interface use the result of the model transformer and the intermediate logic model to generate code of the target language, for example Prolog.

The transformation steps for meta model and model are depicted in Fig. 3 for the example of a model written in the Unified Modeling Language (UML) [5]. The meta model is transformed into a set of predicates and rules
by transforming every relevant meta model element into a predicate. This transformation is described in the following section. Every element of a UML model is transformed into a fact according to the predicate of its meta class. This transformation step is described in section II-E.

C. Meta Model Transformation

As mentioned before, a transformation of the meta model to adequate predicate definitions is necessary to generate facts out of the model data. To support arbitrary meta models, this has to be done systematically by taking the meta meta model into account that defines the elements of an arbitrary meta model may consist of. The framework described here uses the Essential Meta Object Facility (EMOF) [8] as the corresponding meta meta model. Figure 4 depicts the elements of the EMOF relevant for the first transformation step. According to that meta meta model, meta models consist of meta classes that own properties. Properties can be typed with meta classes or primitive types. This allows references between meta classes. Furthermore, meta classes can inherit from each other. Consider the UML meta model cut-out from Fig 3. “Class”, “Property”, and “NamedElement” are instances of EMOF-Class, the first two inheriting from the latter, while “ownedAttribute”, for example, is an instance of EMOF-Property.

The main idea to generate predicates from arbitrary meta models is to traverse a meta model according to the following algorithm:

for instance X of EMOF-Class do

1. To avoid confusion, the elements are prefixed to distinguish them from the identically named elements of the UML meta model.

Using this algorithm, each meta class is represented as a unary predicate. For example, in the case of UML (see Fig. 3), a “Property” is represented as a predicate Property(?X). Properties of meta classes, i.e. meta attributes or references to other meta classes, are represented as binary predicates. The first parameter of a corresponding fact indicates the instance of the meta class, the second one contains the value of the property. For example, “ownedAttribute” (see upper part of Fig. 3), is represented by a binary predicate. Furthermore, rules are created to reflect the inheritance relationships inside the meta model. In the case of UML, they express for example that every class is also a classifier, a namespace, and a named element.

D. Intermediate Logic Model

Instead of directly translating the generated predicates into source code of a fixed logical programming language, a language-independent intermediate model is used. It contains predicates, types and rule definitions for each meta model element and its properties as shown in figure 5. Every predicate object contains information about the predicate’s name, its arity, and its parameter types. Furthermore, it contains some source information that refers to the meta model element that the predicate is generated from. This will
be helpful for following model transformation step, since the sources of meta model elements generation remain traceable.

Rules in a logic programming language consist of two parts. The first part is a rule head consisting of a single predicate, and a rule body, a logical expression. If the body is satisfiable with a certain variable binding, the rule head is satisfiable as well with the same binding. The syntactical structure of rules is reflected in the intermediate model but not depicted in Fig. 5 for simplicity. Rules are used in the intermediate logic model for the representation of inheritance relationships between meta model elements within the meta model.

The elements Predicate, Rule, Type and the intermediate logic model itself implement the interface VisitableElement. This interface is a part of the Visitor-Pattern [9] which will be used to generate predicates, rules and types for the favoured logic programming language by the language-specific implementations of the backend interface. If a visitable element gets visited, its accept-method will be invoked by a visitor producing syntactically correct code of the logic programming language. Therefore, if this logic model structure represents a meta model with all its information, this logic model will be represented in a logic programming language without loss of information.

E. Model Transformation

The second transformation step generates facts from model elements according to the predicates of the meta model’s intermediate logic model. Each instance of a meta model class – i.e., each model element – and its properties have to be traversed and has to be mapped to the corresponding predicate. Consider again the example from Fig. 3. Assuming, that each model element has a unique identifier (here: an integer), we can easily transform the class “Car” and its attribute “power” in corresponding facts Class(1) and Property(2) by getting their meta type and traversing to the predicate in the intermediate logic model. Furthermore, the according facts for the (meta) properties “class” and “ownedAttribute” can be assigned the same way, i.e. the facts class_ownedAttribute(1,2) and property_Class(2,1) can be added.

More difficult is the mapping of meta properties that are not defined by a meta class itself but by one of their super classes. Thus, also the generalisations of model element’s meta class have to be traversed to generate the according facts (see [10] for further details). In the example of 3, this is the case for the fact namedElement_name(1, ’Car’). The according predicate is not defined for “Class” but for the (indirect) super class “NamedElement”.

The result of this transformation is a set of facts that can be written to code of a specific logic programming system by using a corresponding implementation of the backend interface.

III. USING THE FRAMEWORK TO CHECK ARCHITECTURAL COMPLIANCE

This section describes the case study in which the introduced framework was used to realise a prototypical UML model query tool. The tool was designed to check whether a UML design model for a given system is compliant to its intended software architecture. First, the setting of the case study and the prototype will be shortly introduced. Section III-B will describe the queries developed for checking architectural compliance regarding logical layers[11]. The results of the checks are presented in section III-C.

A. Setting and Developing Tool Support

The experiences applying the introduced framework are taken from an industrial cooperation project. The task was to check the compliance between the intended software architecture of a system and its design and implementation as a part of quality assurance. The industrial project partner planned to introduce a new medium-sized information system and charged an external service provider with the development. Adequate tool support had to be developed to be able to check whether the commonly agreed architecture was followed during design and implementation. Due to the size and the complexity of this system and most of real-life systems, checking compliance manually is impossible. Existing tools like for instance SonarJ [12] or Architecture Rules [13] were not considered to be sufficiently flexible for the different architectural aspects to be checked.

One important aspect of the specific architecture is the adherence to the constraints the logical layering implies regarding allowed dependencies between subsystems/components. Figure 6 depicts the logical layers (boxes) of the system and allowed dependency relationships (arrows) with the dependent layer at the end of the arrow. The system consists of a part containing the application logic (layer application core), a part handling the data access (layer persistence) and a
part providing functionality for user interaction (presentation and client) or interaction with external applications (web services). A tool for checking architectural compliance has to be able to check whether this layering is followed in design and implementation. For example, it has to detect an error if some dialogue class (from layer presentation) directly accesses functionality from the layer persistence.

To realise an architectural compliance checking tool for the implementation level, the existing logic-based query tool JQuery [14] for Java source code was used. With slight adaptations, it was able to detect architectural violations. The constraints that have to hold are formalized as queries in TyRuBa [4], a prolog-like logic programming system. Realised as Eclipse plug-in, JQuery returns results of queries as typed elements that reflect nodes of the syntax tree of Java source code.

However, JQuery is only able to execute queries on the abstract syntax tree of Java source code. The design of the system under consideration was modelled by static UML diagrams. The introduced framework was used to realise a prototypical interactive UML query tool for Eclipse[10]. It uses a TyRuBa implementation of the framework’s back-end interface. An extension of this interactive tool includes the generation of predicates for a meta model for simple layer diagrams as depicted in Fig. 6.

Using this tool, a software architect can model the layers of a system and create mappings to packages of a UML design model. The tool generates facts according to the predicates layer/1, allowedDependency/2, and assignPkgToLayer/2 for a layer diagram. The facts for the layers of Fig. 6 are:

```
layer(client).
layer(presentation).
layer(applicationcore).
...
allowedDependency(client,presentation).
allowedDependency(presentation,
applicationcore).
...
assignPkgToLayer(applicationcore,
"org::anonymous::admin").
...
```

The last fact expresses that the mentioned package is mapped to the application core layer.

After that, the user can check automatically whether the actual design model is compliant to the intended layers. For this purpose, the compliance checking tool internally executes the following query for every single layer bound to the variable ?l1:

```
:- illegalLayerDep(?l1,?l2,?el1,?el2).
```

Results of the query contain layers assigned to ?l1 that ?l1 illegally depends on due to a dependency between the model elements ?el1 and ?el2. The following section will refine the definition of this query.

B. Exemplary Queries

At the top-level, the rule defining the illegalLayerDep/4 predicate simply states that a dependency is illegal if there is no allowedDependency/2 fact which states the contrary circumstance:

```
illegalLayerDep(?l1,?l2,?el1,?el2) :-
layerDependency(?l1,?l2,?el1,?el2),
NOT(allowedDependency(?l1,?l2)).
```

The body of the rule defining layerDependency/4 refines the dependencies between layers to dependencies between single model elements:

```
layerDependency(?l1,?l2,?e1,?e2) :-
layer(?l1), layer(?l2),
inLayer(?e1,?l1), inLayer(?e2,?l2),
dependsOn(?e1,?e2).
```

The predicate inLayer(?e,?l) basically checks whether ?e is directly or indirectly contained in a package that is mapped to the layer ?l (see sec. III-A). A detailed definition is omitted here for the sake of brevity and can be found in [16].

dependsOn/2 defines in which cases a dependency between two model elements exists. For static UML diagrams there are mainly four cases:

1) Directed relationships. Relations such as generalisations, interface implementations, package imports, are directed relationships in UML and have to be considered (see ([5])). For example, if a class A implements an interface B, A depends on B.
2) Associations. Associations are often specified with a direction of navigation between its assigned classes or interfaces. If class A is connected with class C by an association navigable towards C, it depends on C.
3) Attributes. Attributes of classes can be typed by other elements of the model, like classes or interfaces. Thus, attributes depend on their type.

4) Operation parameters. Operations may have call or return parameters whose types refer to other elements in the model. As well as attributes, parameters depend on their type.

For each of these cases, a definition of `dependsOn/2` exists. Elements depending on each other by directed relationships are detected by the following rule:

```
dependsOn(?e1, ?e2) :-
    Element(?e1), Element(?e2),
    (EXISTS ?d : DirectedRelationship(?d),
     directedRelationship_source(?d, ?e1),
     directedRelationship_target(?d, ?e2)).
```

This rule holds for two elements `?e1` and `?e2` if and only if there is a directed relationship with `?e1` as its source and `?e2` as its target. The rule for attributes that depend on their type is defined for the classifiers that contain the attributes. Attributes are specified as properties in the UML meta model and belong to a classifier. Thus, the rule

```
dependsOn(?e1, ?e2) :-
    Classifier(?e1), Classifier(?e2),
    (EXISTS ?toEnd : Property(?toEnd),
     property_class(?toEnd, ?e1),
     typedElement_type(?toEnd, ?e2)).
```

is true for classifiers `?e1` and `?e2` whereas `?e2` is the type of a property of `?e1`. Since, association ends are also properties, this rule matches also for some cases where a dependency between elements is manifested by an association, and the opposite association end (`?toEnd`) is a property of the dependent classifier (`?e1`). However, the association ends do not have to be properties of the associated classifiers but can also be properties of the association itself. Nevertheless, the association manifests a dependency in this case, too, and can be detected by the following rule:

```
dependsOn(?e1, ?e2) :-
    Classifier(?e1), Classifier(?e2),
    (EXISTS ?dep, ?end1, ?end2 :
     Association(?a),
     Property(?end2), Property(?end1),
     association_navigableOwnedEnd(?a, ?end2),
     property_association(?end1, ?a),
     typedElement_type(?end2, ?e2),
     typedElement_type(?end1, ?e1)).
```

There is a dependency between classifiers denoted by `?e1` and `?e2`, if an association exists that owns a navigable end towards the classifier `?e2`, i.e. that association end (`?end2`) is typed with `?e2`.

One more rule is required for `dependsOn/2` to detect operation parameters. Operations of classifiers are so-called `features` in UML. If a classifier (`?e1`) contains an operation `?op`, represented by `feature_featuringClassifier/2`, that has a parameter typed with the classifier `?e2`, then there is a dependency between these two classifiers.

```
dependsOn(?e1, ?e2) :-
    Classifier(?e1), Classifier(?e2),
    (EXISTS ?op : Operation(?op),
     feature_featuringClassifier(?op, ?e1),
     (EXISTS ?p : Parameter(?p),
      parameter_operation(?p, ?op),
      typedElement_type(?p, ?e2))).
```

These alternative rules for `dependsOn/2` match in different cases where dependencies in the UML design model exist. Their nested usage in `illegalLayerDep/4` leads to results that describe the identified violations at the level of classifiers and helps to analyse the actual reasons for the violations by looking at the affected classifiers.

C. Results

As mentioned above, the tool prototype based on the introduced framework and the rules described in section III-B were used to check the architectural compliance of the design model of the system depicted in Fig. 6. Several violations of the layers structure were found as depicted by darker arrows in Fig. 7.

The most significant violations in numbers of participating elements (between 10 and 100) were dependencies from the presentation layer to the client layer and from the application core layer to the web services layers which let us draw the conclusion that there is one or more principal design error. The model elements causing violations were determined by the queries and further analysed, thus the following reasons for layer violations were identified:

- Definition and usage of data transfer objects [17]: Most violations between presentation layer and client layer result from using transfer objects for the exchange of data between these layers. The use of transfer objects is discouraged because they can introduce complex dependencies and make the code harder to understand and maintain. Instead, it is recommended to use data access objects or value objects for data exchange.

![Figure 7. Identified violations of the logical layers structure.](image_url)
of data (similar between persistence and web services layer). The classes defining the shape of those transfer objects, however, are defined in the client layer causing violations by the necessary access of the presentation layer. A better solution would probably be to define the transfer object classes in the presentation layer.

- The violations between the application core layer and client layer result from using a special kind of search functionality defined in the client layer. It should be refactored and moved to the cross-cutting utility layer.
- The utility layer accesses the persistence layer to realize some security functionality, mainly for authorization and authentication. It has to be further investigated whether and how the system has to be adapted to avoid this violation.

Between the application core layer and the web services layer the observer pattern [9] is used. Observable objects call callback methods to inform observers in the web services layer about modifications. This leads to the import of corresponding observer interfaces that need to be used. However, as described in [18], the call of a callback method does not actually constitute a dependency, since the correct functionality of the caller does not depend on the correct implementation of the callee. Thus, these matches of the query were not considered architectural violations.

The query tool helped significantly to detect architectural violations. For the exemplary system and its relevant 1,600 classes and interfaces, the checks for architectural layer violations took about 1–2 minutes on a common desktop PC.

IV. RELATED WORK

The proposed approach is one among many other approaches dealing with model queries. The most trivial query mechanism to query the content of a model is to use APIs of modelling tools. Of course, this solution is not very flexible, since it is restricted to the modelling language a tool supports. For model-driven development, more flexible approaches are required.

Another logic-based approach is proposed in [19]. It introduces a transformer called Model Manipulation Tool (MoMaT). It accepts different kinds of modelling languages like Event Process Chains, Use Case Maps or the Unified Modeling Language. Each model is transformed into an internal representation that is independent from the input model language. MoMaT defines an own meta model that cover the most model elements occurring in different languages. This meta model and the internal model representation are mapped to Prolog facts and can be queried. The MoMaT approach is similar to the one proposed here. The main difference, anyway, is that MoMaT is restricted to Prolog.

In [20] the logic programming language LOOM is used check inconsistencies within UML models. Models have to be transformed into the LOOM language format via an XSL [21] transformation. The source format of the models is XMI-based and restricted to it. After the transformation process LOOM is able to check the models for inconsistencies and to fix them to a certain degree. Although, it is possible to query UML models this way, the approach has a slightly different focus on consistency detection. Goal of our work is a general querying framework that can be used to realize more complex tasks like inconsistency detection with more flexibility.

Database-based approaches store models in databases and use well-known languages like SQL to query them. SQL is either used directly as query language or indirect to represent constraints in higher-level languages like OCL (e.g. [22]).

OCL itself could be used as query language directly, since there exist some sophisticated implementations [23]. However, expressions constrain instances of the classes and concepts that are mentioned in the expression [24]. Thus, to query a model for certain properties, e.g. for all classes with a certain attribute, a constraint would have to be defined in the meta model (retrieve all instance of the meta class “Class that...”). This is not easily possible or desirable for non-language but purpose-specific querying tasks.

Furthermore, there exist also graph-based approaches that represent meta model as graph schemes and specific models as graphs. Queries are expressed as graph patterns that are searched for in a host graph which corresponds to detecting subgraph isomorphisms between the pattern and the host graph, the model. Graph-based approaches or engines that support querying can be found in [25].

V. LESSONS LEARNED AND FUTURE WORK

The experiences from the case study are very positive. The development effort of the prototypical tool of a UML query tool was very low and was mainly reduced to the integration as Eclipse plug-in and of the logic programming system in the back-end. At the moment, the framework is used to realise a more-general architectural compliance checking approach with support for a larger set of meta models to check architectural compliance for more model types [26]. It support to easily realise meta model independent tools but also to easily change the underlying logic programming system to test different systems for their adequacy for this purpose.

From the case study we have learned that logic-based query tools can perform very well. The performance of the query tool was good for our intended use. Of course, more detailed performance analysis, especially for larger models and systems have to be made. This may eventually lead to the usage of more efficient but less expressive formalisms like description logics.
The user acceptance was high due to the relatively intuitive way to formulate queries although most of them were not familiar with prolog-like languages. One drawback applies especially or complex meta models like the UML specification. To efficiently use the predicates generated by the framework, one has to have a certain knowledge of the meta model and the abstract syntax of the modelling language it defines. But usually, e.g., users of UML case tools are more familiar with the concrete syntax they come into contact with by diagrams and tools. Additionally generated predicates that are more similar to the concrete syntax would be necessary.

However, the main drawback remains the definition of queries in tools that use the framework. At the moment, they have to be defined using the language of the intended back-end implementation. Changing the implementation means that queries have to be rewritten. Thus, the intermediate model of the framework will be improved and extended in a way that first-order logic queries are reflected there. The back-end interface will be adapted and support code generation for queries defined in the intermediate model.

ACKNOWLEDGEMENT

We would like to thank Andreas Rausch and Constanze Deiters for the cooperation during the exemplary case study.

REFERENCES


