An implementation of VMQL

Radu-Vlad Acretoaie

Kongens Lyngby 2012
IMM-MSc-2012-81

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-MSc-2012-81

Summary (English)

As domain models may reach considerable sizes, retrieving the knowledge contained in them is often a nontrivial task. The fact that this task, known as *model querying*, must sometimes be performed by non-technical domain experts only makes matters more difficult. Existing model querying techniques suffer from limitations that make them unsuitable for many practical purposes.

A proposed solution attempting to overcome the drawbacks of current ad-hoc model querying facilities is the visual model query language (VMQL). It aims to simplify the process of retrieving information from models by allowing modelers to express queries using the host modeling language, thus eliminating the need for them to learn a new query language. Additional expressive power is added to queries through model annotations. VMQL is a highly portable solution which may be applied to a wide array of host modeling languages.

This thesis presents a tool implementing VMQL for UML. Since it is the second prototype to attempt this, the tool is called MQ-2. In order to provide seamless integration with existing modeling facilities, it is integrated with the MagicDraw UML modeling tool. MQ-2 is based on a query execution algorithm written in the Prolog logic programming language.

Summary (Danish)

Eftersom domæne modeller kan nå store størrelser, er det at finde den viden der er indeholdt i dem, ofte ikke en triviel opgave. Det faktum, at denne opgave, der er kendt som model forespørge, skal undertiden udføres af ikke-tekniske domæne eksperter gør kun opgaven vanskeligere. Eksisterende model søgnings teknikker lider af begrænsninger, som gør dem uegnede til mange praktiske formål.

En foreslået løsning at forsøge at overvinde ulemperne ved den nuværende ad hoc-model søgninger faciliteter er visual model query language (VMQL). Formålet er at forenkle processen med at hente information fra modeller ved at tillade brugere at udtrykke forespørgsler med værten modellering sprog, hvilket eliminerer behovet for dem at lære et ny forespørgsel sprog. Yderligere udtrykskraft føjes til forespørgsler gennem model anmærkninger. VMQL er en transportabel løsning, som kan anvendes til en lang række værtsceller modellering sprog.

Denne afhandling præsenterer et værktøj gennemførelse VMQL for UML. Da det er den anden prototype at forsøge dette, er det værktøj kaldet MQ-2. For at give en problemfri integration med eksisterende modellering faciliteter, er det integreret med MagicDraw UML modellering værktøj. MQ-2 er baseret på en forespørgsel udførelse algoritme skrevet i Prolog logik programmeringssprog.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfillment of the requirements for acquiring an M.Sc. in Informatics. Work on this thesis was conducted under the supervision of Associate Professor Harald Störrle.

The thesis deals with describing MQ-2: an implementation of the visual model query language (VMQL) as a plug-in to the MagicDraw CASE tool. VMQL is a graphical model query language that allows expressing queries as annotated model fragments in the host modeling language. This approach enables modelers to express complex queries without requiring them to master a dedicated query language. In addition to implementing the original specification of VMQL, the presented tool brings several modifications to the language. The motivations behind these modifications are also discussed in this thesis.

The presented implementation is based on a query execution engine created using the Prolog logic programming language. It relies on representing existing models as Prolog fact databases. As such, the implementation also allows users to query models directly through a fully featured integrated Prolog console.

While VMQL is in theory applicable to a wide range of modeling language, this thesis focuses on its application to UML.

The thesis consists of an introduction presenting its domain, objectives, and related work (Chapter 1); an overview of the visual model query language (Chapter 2); an analysis of the requirements for the presented tool (Chapter 3); a description of the tool's implementation (Chapter 4); a guided tour of the tool (Chapter 5); an evaluation of the tool's performance and coverage of the VMQL

specification (Chapter 6); conclusions and possible future developments (Chapter 7).

Lyngby, 10-August-2012

Radu-Vlad Acretoaie

Acknowledgements

I would like to thank my supervisor, Associate Professor Harald Störrle, for his invaluable feedback and thoughtful suggestions regarding both the overall direction of this thesis and day-to-day implementation challenges.

Contents

Chapter 1

Introduction

This chapter presents the context and objectives of the thesis (Section 1.1), followed by a review of related work (Section 1.2).

1.1 Context and objectives

Models play a prominent role in several software development approaches. In model based software development, domain models are an important focal point of the development process, as part of an effort ensure the suitability of the created application to its intended usage context.

Industrial expertise has shown that the large sizes reached by models in general, and domain models in particular, can quickly become a limiting factor in their usage [Stö10]. Activities that appear trivial for small models, such as tracking the differences between model versions or searching for certain structures within a model, become time consuming tasks for large scale models.

This thesis focuses on the problem of model querying: effectively specifying and executing searches on a model with the purpose of retrieving certain model elements or model structures. Model querying is arguably one of the most basic

day-to-day activities performed in the process of working with models. It is also a necessary first step in more complex activities such as enforcing model constraints or performing model transformations. When working with large models, query languages capable of ensuring high levels of precision and recall are a necessity. Apart from the challenges brought by model size, querying domain models raises particular requirements (high usability and learnability) motivated by the fact that they are often created and maintained by domain experts that do not necessarily have a technical or programming background.

Several approaches to model querying have been proposed, ranging from basic full text search to full blown textual and graphical query languages (see Section 1.2). One recent such approach is the visual model query language (VMQL) [Stö11b]. In short, VMQL is a graphical query-by-example approach that allows modelers to formulate queries as annotated model fragments in the modeling language that they are already using to create their models. The goal of VMQL is to leverage modeler's existing skills and provide them with a powerful query facility that incurs a minimal learning effort. A full description of VMQL is provided in Section 2.

The main objective of this thesis is to provide a working implementation of VMQL. The implementation is to be created as a plug-in to the popular MagicDraw CASE tool[1], so that it can be available to a wide audience of existing modelers. Furthermore, the implementation should be based on the Prolog logic programming language in order to further explore the use of Prolog for model manipulation presented in [Stö07] and [Stö09]. This consideration leads to a secondary objective for this thesis: providing modelers with a fully featured Prolog console integrated in MagicDraw. This feature will allow advanced processing extending beyond the immediate scope of model querying to be performed on MagicDraw models. Finally, the implementation's performance should be evaluated on a set of models of various sizes. In what follows, the implemented tool will be referred to as MQ-2 (an implementation of an earlier version of VMQL has been developed under the name of MQ [Win09]).

Although VMQL is not bound to any specific modeling language, the MQ-2 tool is targeted at the Unified Modeling Language [UML11]. This choice is motivated by the fact that UML has become a de-facto standard in the software engineering community. Furthermore, a successful implementation for UML is portable to other Meta-Object Facility (MOF) [MOF06] modeling languages with minimal effort. In fact, VMQL's design is explicitly aimed at portability, making the application of MQ-2 to other modeling languages such as Business Process Model and Notation (BPMN) [BPM10] possible with relatively minor adjustments. However, such adjustments are outside the scope of this thesis.

[1] http://www.nomagic.com/products/magicdraw.html

1.2 Related work

The model query approaches available at present can be grouped into four general categories: tool specific solutions (Section 1.2.1), domain specific solutions (Section 1.2.2), textual query languages (Section 1.2.3) and visual query languages (Section 1.2.4). As suggested in [Stö11b], the following aspects should be taken into account when evaluating model query approaches:

1. *Expressiveness*: The variety of queries that may be expressed.
2. *Genericity*: The variety of supported modeling languages.
3. *Usability*: The ability of users to take advantage of the tool's features regardless of the depth of their technical background.
4. *Practicality*: The extent to which a tool is actually implemented and available to users.
5. *Performance*: The extent to which a tool's response times accommodate interactive work.

Table 1.1 presents an overview of the advantages and drawbacks of the different model query approaches that fall under each of the four categories, while the following sections present a more comprehensive analysis of each approach.

1.2.1 Tool specific solutions

Given that executing queries is a fundamental necessity in the modeling process, most modeling tools provide some form of model query facility. Possibly the most common among such facilities is the ability to perform a *full text search of element names* in the model. While it is trivial to use, a full text search is only expressive enough for the most trivial of queries. Furthermore, many domain-relevant search terms are likely to appear in a large number of locations in the model. Finally, the results produced by a full text search are dependent on the search algorithm implemented in the tool, which may or may not provide accurate results. While some tools (e.g. MagicDraw) support regular expressions as search terms, the severe limitations of this query method remain present.

Recognizing the need for expressing more complex queries, some tools provide *predefined queries* considered of interest by a particular tool's creators. MagicDraw, for instance, provides the ability to search for model elements by stereo-

Table 1.1: Overview of existing model query approaches

Approach	Strengths	Weaknesses
Full text search of element names	usability, practicality	expressiveness, false positive results
Predefined queries	expressiveness, usability, practicality	no support for ad-hoc queries
Model visualization tools	usability	genericity, practicality (depending on implementation)
Tool specific APIs	expressiveness	usability
Low level query facilities	expressiveness	usability
Domain specific solutions	expressiveness, usability	genericity, practicality
OCL [AB01]	expressiveness, practicality	genericity, usability
Visual OCL [BKPPT01]	expressiveness	practicality
Constraint Diagrams [Ken97]	usability	genericity, practicality (never implemented)
BP-QL [BEKM08]	usability, practicality	genericity (limited to WS-BPEL models)
BPMN-Q [AWW11]	usability, practicality	genericity (limited to business process modeling languages)
JPDDs [SHU04]	usability	genericity, practicality

type, meta-class or meta-attributes. Other tools, such as ADONIS[2], allow parameters and logical connectors between the predefined queries. However, this approach does not allow defining ad-hoc queries.

Model visualization tools are also a commonly provided facility in many modeling environments. They include various tree views of the model, such as containment or inheritance trees, automatically generated overview diagrams, and textual model reports. All of these tools can be of use in the process of inferring certain properties of a model or locating elements in the overall model structure. But, strictly speaking, they are not query facilities, and their implementation may vary considerably between modeling tools.

Several established modeling tools such as Rational Rose[3], Sparx Enterprise Architect[4], and MagicDraw provide model access through *tool specific APIs* [Stö11b]. Each tool vendor may provide such an API in its programming language of choice. Although these APIs provide virtually unlimited access to the model, they require modelers to possess programming skills in a particular programming language. This requirement is not always met, especially when modelers come

[2] http://www.boc-group.com/products/adonis/
[3] http://www-01.ibm.com/software/awdtools/developer/rose/
[4] http://www.sparxsystems.com.au/

from a background as domain experts. Furthermore, writing a new application for every new type of query is exceedingly time consuming.

Finally, considering that all tools must provide persistent model storage, it may be possible to query the persistent representation of a model using *low level query facilities*. Several tools support model storage in files adhering to the XML Metadata Interchange (XMI) standard [XMI11]. It is thus possible to extract information from such files using generic XML querying facilities such as XPath [XPa10]. Alternatively, in case the model is stored in a relational database, its internal representation may be queried using SQL. Still, these approaches are at least as technically challenging as tool specific APIs, and for this reason are also unlikely candidates for specifying ad-hoc queries.

1.2.2 Domain specific solutions

After considering the available tool specific solutions in Section 1.2.1, it is possible to observe that they are plagued by one of two problems: they are either not flexible enough to allow complex ad-hoc queries or require a significant investment in terms of time and programming expertise. Both of these issues are successfully addressed by domain specific query solutions, which are able to obtain expressiveness and usability at the expense of genericity. Such solutions address the needs of various domains of human activity that produce large amounts of data that must be analyzed. Domain specific query languages significantly improve the efficiency of the analysis process.

An example domain specific query approach is the PHEASANT visual query language [AHM05], which addresses the domain of high energy physics. Its authors state that prior to the development of PHEASANT, physicists were required to program individual queries in various programming languages, a problem very similar to the one identified in Section 1.2.1 for tool specific solutions. PHEASANT features a strictly defined semantics, but relies on a highly domain specific syntax, making it inapplicable for tasks outside the area of high energy physics. The query language is implemented as a stand-alone tool.

A second relevant example is the HyperFlow visual query and dataflow language, targeted primarily at querying scientific workflows in areas such as bioinformatics [DP05]. The main challenge identified by the authors in this domain is related to data integration, with various research groups making experimental results available through a plethora of interfaces, ranging from databases and ontologies to Web forms and Web services. In addition to its role as a visual query language, HyperFlow also serves the purpose of a visual dataflow language, allowing simple queries on different types of data sources to be combined into

more complex workflows. HyperFlow offers a rich variety of features, including sub-queries and SQL-like operations such as JOIN and GROUP BY. While not as limited to a single domain as PHEASANT, the syntax of HyperFlow is still nowhere near as general as that of a generic modeling language such as UML.

Although they are well suited for their intended usage, HyperFlow and PHEASANT serve the purpose of highlighting the main weakness of domain specific query solutions: they can only be used in a very limited context. In order to overcome this drawback and maintain the advantages of expressiveness and usability, it is necessary to introduce either a textual (see Section 1.2.3) or a visual (see Section 1.2.4) generic model query language.

1.2.3 Textual model query languages

Creating tool independent, highly expressive model queries requires the availability of dedicated query languages. In the case of UML, this role is most commonly fulfilled by the Object Constraint Language (OCL). Although OCL is most often mentioned in the context of expressing model constraints, the Object Management Group (OMG)[5] defines OCL as both a query and constraint language for the Meta-Object Facility (MOF) family of modeling languages, which includes UML [OCL11].

The usage of OCL as a query language for UML data models has been proposed in [AB01], where the authors also propose a number of extensions to OCL, with the purpose of facilitating its usage as a query language. A parallel is drawn between relational databases and Object-Oriented data models, based on the observation that both require a means of querying data. Relational databases possess such a means in the form of the Structured Query Language (SQL). In the field of relational databases a query language is required to have, at a minimum, the expressive power of a relational algebra [Dat99]. A relational algebra is defined as "a set of operators that take relations as their operands and return a relation as their result" [AB01]. Consequently, a query language that claims to have the expressive power of a relational algebra must support a minimum set of primitive operators (Union, Difference, Product, Project, and Select) [Dat99]. For Object-Oriented data models, the combination of OCL and UML does indeed have the expressive power of a relational algebra [AB01] (note, however, that OCL *in isolation* does not support the specification of some queries that can be specified in relational algebra [MC99]). While OCL independently supports the Union, Difference, and Select relational operators, it requires the UML concepts of *AssociationClass* and *n-ary Association* in order

[5] http://www.omg.org/

to support the Product and Project operations. The authors of [AB01] propose the addition of a *Tuple* type to OCL in order to streamline the usage of OCL as a query language without making any changes to the queried UML model.

An OMG initiative aimed at basing a standard set of model query and transformation languages on OCL has been proposed in the form of Query / View / Transformation (QVT) [QVT11]. Early QVT requests for proposals [QVT03] have garnered a significant level of interest, leading to the creation of OCL-based model transformation languages such as ATL [JABK08] or MOLA [KBC04]. Note that a model transformation can be defined as the process of transforming a source model conforming to a certain meta-model into a target model conforming to a different meta-model [JABK08]. While it only partially complies with the QVT requirements, ATL benefits from relatively extensive and mature tool support based on the Eclipse Rich Client Platform (RCP)[6]. Furthermore, ATL is not limited to MOF meta-models, allowing source and target models represented as XML documents or SQL databases. MOLA combines a textual language with a graphical syntax in an effort to improve usability.

Although they are the most prevalent text-based model query language, OCL and its extensions are not the only such languages available. In [Stö07], an approach based on using Prolog as a query language is proposed. The Model Manipulation Toolkit (MoMaT) is in fact a predecessor to VMQL and introduces the concept of representing models as collections of Prolog facts which can then be queried from a Prolog console. MoMaT also includes a library of predefined Prolog predicates that provide a more accessible way of interacting with the model's Prolog representation. However, early usability evaluations have pointed to the fact that the low level query interface provided by MoMaT is not adequate for most modelers [Stö09]. As a consequence, a more comprehensive library of Prolog predicates functioning on top of the MoMaT infrastructure has been put together under the umbrella of the Logical Query Facility (LQF). LQF "captures the properties and relationships of model elements in the terms modelers are accustomed to rather than in terms of the underlying meta model" [Stö09].

A performance comparison between MoMaT and an Eclipse Modeling Framework (EMF)[7] based OCL interpreter is presented in [COL08]. The authors consider large UML models consisting of up to 10000 classes. The results of this comparison highlight the fact MoMaT outperforms OCL for relatively simple queries based on model element properties. However, OCL is faster when evaluating queries based on relationships between model elements. A second aspect tested is the time required to load a model into memory in the two query

[6] http://wiki.eclipse.org/index.php/Rich_Client_Platform
[7] http://www.eclipse.org/modeling/emf/

frameworks. For this scenario, MoMaT proves to be faster by a factor of two.

Nevertheless, query execution performance is not the main setback of OCL. As with all textual query languages, OCL suffers from the existence of a medium gap between the graphical notations used to express models and the textual notations used to express queries. Empirical studies have shown that using OCL to express all but the most trivial queries can be challenging even for experienced modelers [Stö11b]. It is reasonable to expect that such usability deficiencies on the part of OCL become more prominent when its users are not professional modelers but domain experts. One proposed solution to bridging the medium gap between OCL and the models it is meant to be executed on is Visual OCL [BKPPT01]. The main feature of Visual OCL is the ability to represent object property navigation expressions as collaboration diagrams. The implicit assumption of the authors is that object navigation can be made easier by offering a visual representation for the navigation paths. Visual OCL strives to mostly use existing UML notation elements (though some minimal additions prove to be necessary), thus avoiding the need for modelers to learn a new graphical language for expressing queries and constraints. However, tool support for Visual OCL is not presently available.

Although it represents a step in this direction, Visual OCL cannot fully eliminate the medium gap between models and queries. This is because a query expressed using Visual OCL does not visually reflect the structure of the expected result, but rather the structure of the underlying OCL expression. Thus, the complexity of OCL remains an aspect that modelers must handle. A real improvement in usability in the area of model query languages must place no requirement on modelers other than that of being familiar with the modeling language itself. This improvement can be achieved by adopting a query-by-example approach, in which the query is a "blueprint" for the expected results. Such an approach has led to the development of several visual model query languages.

1.2.4 Visual model query languages

Expressing model constraints using a visual notation has been proposed shortly after the introduction of UML, in the form of Constraint Diagrams [Ken97]. The described notation has a relatively narrow scope: it aims to enhance the expressiveness of UML object diagrams by introducing constraints on the relations between object states. The author claims that Constraint Diagrams offer an improved usability compared to textual notations available for similar purposes at the time, which notably do not include OCL. It should be noted that model constraints and queries are complementary concepts: expressing a constraint may be interpreted as expressing a query that must have an empty result set

in order for the constraint to hold [Stö11a]. Thus, Constraint Diagrams can be viewed as an early example of a graphical model query language. They are visually represented as a combination between class diagrams and Venn diagrams, and as such cannot be regarded as a true query-by-example approach with respect to UML class diagrams.

BP-QL [BEKM08] is a visual language for querying business process models. It provides usability advantages similar to the ones offered by VMQL, but restricted to the area of business process modeling, namely an abstraction of the Business Process Execution Language (BPEL), currently standardized and known as Web Services Business Process Execution Language (WS-BPEL) [WSB07]. With this restriction in mind, BP-QL is tailored to the particular challenges of querying business processes. These include, for instance, the existence of infinitely large result sets obtained when a business process contains recursive activities. While very likely appropriate for its intended use, BP-QL was not designed with the ability to query other modeling notations in mind, and some of the challenges it attempts to solve are not relevant for a more general modeling language such as UML. An interesting aspect of BP-QL is the fact that simple constraints on the query model are specified by custom graphical notations, rather than a textual constraint language. As an example, paths of unlimited length are identified by a double headed arrow. The implementation of BP-QL presented in [BEKM08] is based on processing the XML representation of BPEL models and translating them into Active XML [GT08] documents for increased ease of query processing. While the authors provide a motivation for their implementation choices, they do not compare them with other possible alternatives. At the same time, no alternative query languages for BPEL models are mentioned, making the usability arguments invoked by BP-QL's creators hard to substantiate.

Another proposed solution for graphically querying business process models comes in the form of the Business Process Model Notation Query language (BPMN-Q) [AWW11]. Its authors envision BPMN-Q as a means of specifying domain-dependent compliance constraints on business process models. Thus, modelers are provided with an alternative to model checking for the task of verifying constraint compliance. However, the underlying task remains fundamentally that of model checking. For this reason, BPMN-Q models are translated to Computational Tree Logic (CTL) [Hal85] - a widely used logic in the area of model checking - expressions at execution time. It is argued that BPMN-Q provides an accessible means for modelers to harness the power of CTL without being required to possess model checking expertise. BPMN-Q can consequently be seen as a graphical front-end for CTL. An implementation of BPMN-Q has been created and presents a number of interesting facilities. Among them is the ability to show constraint violations as BPMN-Q "anti-patterns" which, when used as queries on the original model, retrieve the constraint violating model

fragments. It thus becomes apparent that anti-patterns are essentially model queries. Just as in the case of BP-QL, a potential drawback of BPMN-Q is the fact that it proposes the addition of new notation elements to the host modeling language (in this case BPMN [BPM10]). Unlike BP-QL, BPMN-Q is designed as an abstraction over common business process modeling notations, making it easily portable to other host modeling languages.

For the task of querying UML models, the approach with the highest conceptual similarity to VMQL is that featured in Query Models [SHU04]. The authors also propose a Query-By-Example approach to formulating queries on UML models, while motivating their work using a series of Model-Driven Architecture and Aspect-Oriented Software Development scenarios in which it proves relevant. Join Point Designation Diagrams (JPDDs) are proposed as a notation for specifying model queries. JPDDs are based on elements of the UML notation, but introduce several query-specific symbols. Further work by the same authors [SHU05] identifies the need to specify binary relationships between JPDDs with the main purpose of increasing the expressiveness of queries. For instance, two JPDDs representing queries on a UML class diagram and, respectively, a UML sequence diagram may be combined to create a query specified from both a structural and behavioral perspective. This approach also permits the creation of abstractions covering recurring selection patterns. Such abstractions may be stored and re-used whenever appropriate.

As opposed to VMQL, which maps to Prolog executable code, Join Point Description Diagrams are meant to be translated into and executed as OCL constraints. The process of obtaining OCL constraints from JPDDs is, however, not fully defined. Only some examples are presented. A second limitation of JPDDs is that they are only exemplified for class diagrams and sequence diagrams, without any mention of their applicability to other UML diagram types. Finally, the notation introduced for creating JPDDs contains some additions to the standard visual representations of UML model elements, such as crossed lines depicting indirect associations. Considering these aspects, it may be concluded that JPDDs share a common vision with VMQL but stop short of carrying this vision through into an actual implementation, which is precisely the objective of this thesis with respect to VMQL.

Chapter 2

The visual model query language

The visual model query language (VMQL) is a novel model querying approach that proposes using the host modeling language as a basis for expressing queries. This section provides an overview of VMQL based on its specification in [Stö11b]. Some additions to this specification have been made as part of this thesis. They are discussed in Section 6.1, while this section limits itself to VMQL's original specification. A recent paper [Stö11a] proposes extending VMQL to function as a model constraints language. However, these extensions are outside the scope of this work. VMQL is considered here exclusively as a model query language.

The target usage scenario of VMQL is within the context of domain modeling, where modelers with a business background do not have the necessary skills and motivation to master a complex new query language. In order to appeal to this category of users, VMQL adds no additional syntax elements to the host modeling language: any model fragment can be executed as a query. Additional expressiveness is added to queries by attaching comments containing textual VMQL constraints to certain query model elements. The syntax of the various VMQL constraints is described in the following paragraphs. The feature of VMQL that sets it apart from other query-by-example languages ([SHU04], [AWW11], [BEKM08]) is the fact that it is not bound to a certain modeling language. This is made possible by the fact that VMQL operates pri-

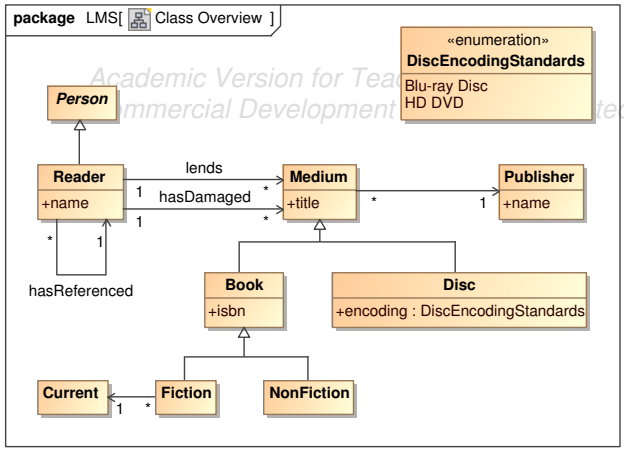


Figure 2.1: Library Management System source model: class overview

marily at a syntactical level: it matches model elements from a query model to model elements from a source model based on their type and attributes, without taking into account their semantics. The only constraint placed on the host modeling language is its required support for some form of comments or annotations. While the MQ-2 tool is designed to support querying UML models, and the features of VMQL are exemplified in this section based on UML diagrams, there is no conceptual limitation to applying the same techniques on EPC or BPMN models, for instance.

In what follows, VMQL is described through a series of examples based on a sample Library Management System (LMS) scenario. A summary of the constraints proposed by VMQL is provided at the end of this section in Table 2.1. The sample model that will be queried in what follows consists of two diagrams: a Class Diagram presented in Figure 2.1 and an Activity Diagram presented in Figure 2.2. The Class Overview diagram showcases the relationships between some of the entities involved in a typical LMS scenario, where readers may lend various media types (in this case books or discs), which are published by certain publishers. A record is kept for readers that have damaged items, as well as for readers that have referenced other readers and pursued them to join the library. The Lend Medium diagram illustrates the business process of a reader lending a medium. The medium is identified and checked for availability, the reader is identified and checked for eligibility, and a new lending is created. After these three steps are performed in parallel, a decision is made as to whether the lending can proceed. If this is the case, a record of the transaction is created.

Otherwise, an error notification is displayed.

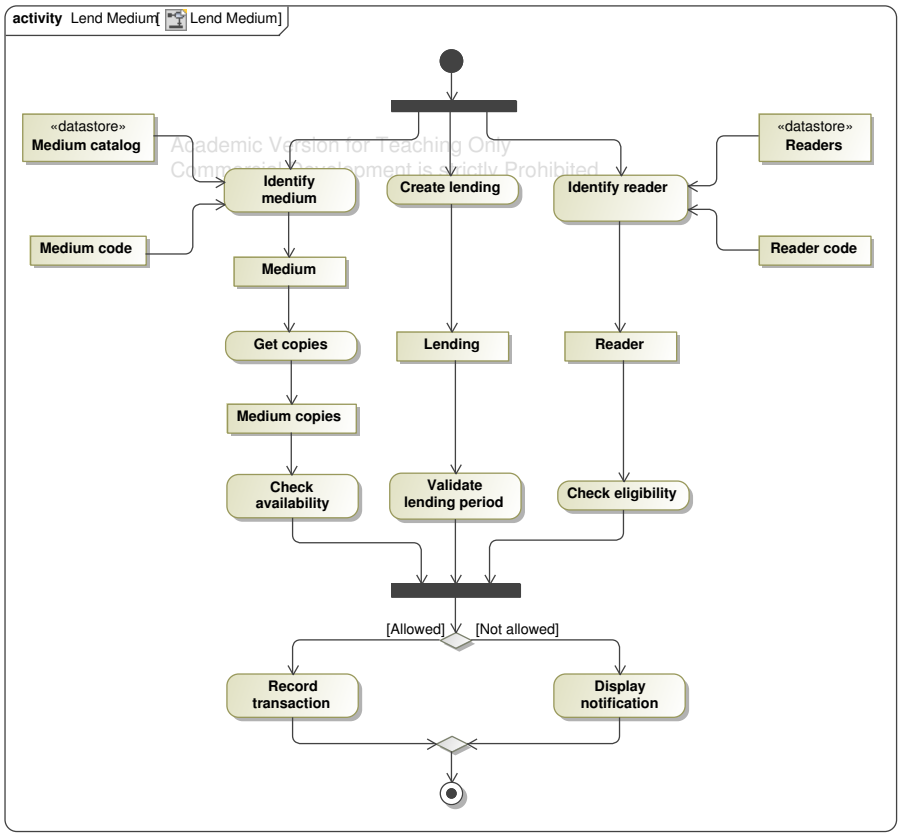


Figure 2.2: Library Management System source model: lending a medium

This model is referred to as *source model* in what follows. VMQL queries executed on this model are referred to as *query models*, given that they are in fact models themselves. A query model normally consists of a single diagram.

2.1 Base queries

In this context, a modeler may be interested for instance in determining the media types provided by the library. Query 1 presented in Figure 2.3 is a VMQL query performing this function. Since it does not contain any annotations rep-

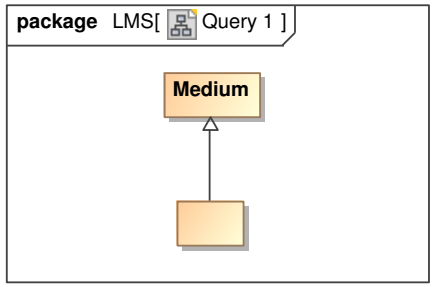


Figure 2.3: VMQL query finding all subclasses of the `Medium` class

resenting VMQL constraints, this is a *base query*. It should be interpreted as instructing the VMQL execution engine to match all classes of the source model that are named Medium and have one subclass, regardless of the name of this subclass. It is important to notice that a VMQL query specifies a *minimum set of constraints* that must be satisfied by any matching source model fragment: any other properties of the matched model elements are irrelevant. For this reason, executing Query 1 will match the `Medium` class of the source model even if it has an extra attribute (`title`) and a number of associations to other classes. It will also match the unnamed subclass of `Medium` from the query model to two classes of the source model, `Book` and `Disc`, despite their extra attributes and subclasses. Query 1 thus succeeds in identifying all media types supported by the library, without incurring any additional query-specific notations: it is an ordinary Class Diagram.

2.2 The `mattr`, `mclass`, `name`, and `match` constraints

Consider now a scenario in which a modeler is interested in finding all generalizations involving an abstract super-class. Such a query may be written by adjusting Query 1: the name of the `Medium` class must be removed and the class must be made abstract. Alternatively, Query 2 presented in Figure 2.4 may be used. It uses the `mattr` VMQL constraint to set the value of the `isAbstract` meta-attribute of the super-class to false. It also stores the name of the identified super-class in the `$Superclass` variable, using the `name` VMQL constraint, which is a short-hand notation for a `mattr` constraint involving the `name` meta-attribute. The `name = $Superclass` constraint in Query 2 could be re-written as `mattr name = $Superclass`.

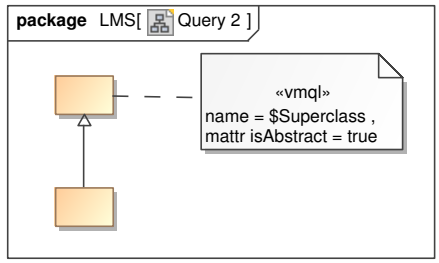


Figure 2.4: VMQL query finding all abstract super-classes

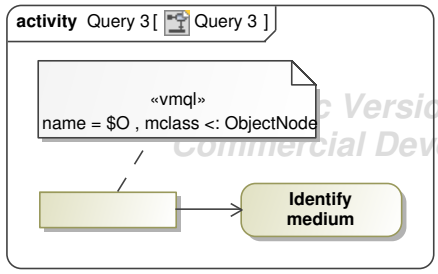


Figure 2.5: VMQL query finding all object nodes preceding the `Identify medium` action

Query 3 in Figure 2.5 illustrates the `mclass` constraint. Suppose a modeler is interested in finding all object nodes directly preceding the `Identify medium` activity in the Lend Medium Activity Diagram. The expected results would be the `Medium catalog` and `Medium code` object nodes. However, because `Medium catalog` is actually a data store (a specialized type of object node), it will not be returned in case the `mclass` constraint is omitted. By adding the `mclass <: ObjectNode` constraint, the modeler specifies that he is interested in all model elements of types that are specializations of `ObjectNode` in the UML meta-model. Thus, when taking into account the `name` constraint, the variable `$O` will be bound to the values "Medium catalog" and "Medium code".

A similar usage of the `mclass` constraint is featured in Query 4 presented in Figure 2.6. This query also introduces the `match` constraint, which is used here to set a condition on the `name` meta-attribute of the model elements to which it is anchored. In this case, it states via a regular expression that the names of the returned elements must start with the prefix "Disc". The query returns both

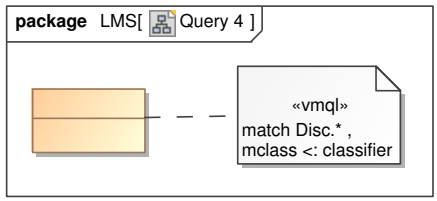


Figure 2.6: VMQL query finding all classifiers having the prefix "Disc" in their names

the `Disc` class and the `DiscEncodingStandards` enumeration from the Class Overview diagram, keeping in mind that in the UML meta-model both classes and enumerations are specializations of the `Classifier` meta-class. Note that the `mclass` and `mattr` constraints are the only VMQL constraints that explicitly access the meta-model of the host modeling language.

2.3 The `distinct` and `once` constraints

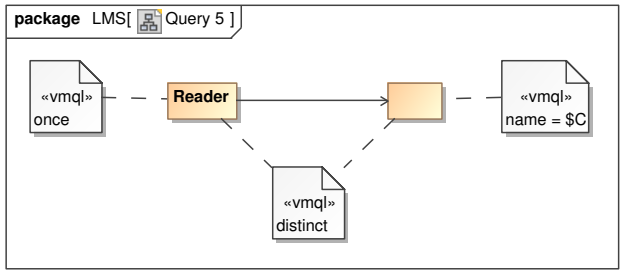
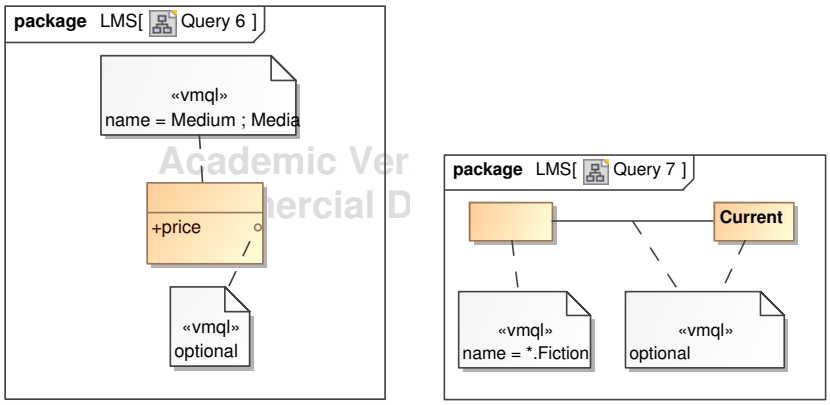


Figure 2.7: VMQL query finding all classes linked by a directed association to the `Reader` class

In the Class Overview diagram, the `Reader` class has outgoing directed associations towards itself and the `Medium` class. Consider a situation in which a modeler wished to locate only the classes connected to the `Reader` class by such directed associations that are *different* from the `Reader` class itself. Query 5 in Figure 2.7 accomplishes this by using the `distinct` VMQL constraint. This constraint states that the query model elements to which it is anchored must have distinct bindings. Notice, however, that the source model contains two

associations of the specified type between the Reader and Medium classes. This will cause the query to produce a duplicate solution. To avoid this, the once VMQL constraint is introduced. This constraint specifies that a solution "may occur only once in the set of all solutions" [Stö11b].

2.4 The optional, either, and not constraints

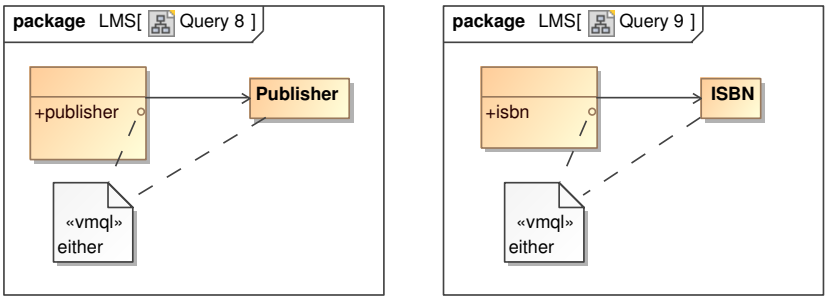


(a) (b)

Figure 2.8: VMQL queries for finding: (a) a class with an optional attribute, (b) a class with an optional association

To exemplify VMQL's optional constraint, consider Query 6 in Figure 2.8(a). In this scenario, a modeler is searching for a class that he knows is named either Medium or Media and expresses this using the name constraint (notice the use of the *or* logical connector denoted by the ; symbol). He also knows that such a class may have a price attribute, but would also like to retrieve it in case it does not. This is achieved using the optional constraint, which states that the query model element to which it is anchored may or may not be bound to a source model element. Consider now the query depicted in Figure 2.8(b), where the modeler is searching for a class whose name ends in the "Fiction" suffix and which has an optional directed association to a class named Current. This query returns the Fiction and NonFiction classes. Applying the optional constraint to the Current class is not sufficient, as the association in the query model is still required in any binding. The optional constraint must also be applied to this association.

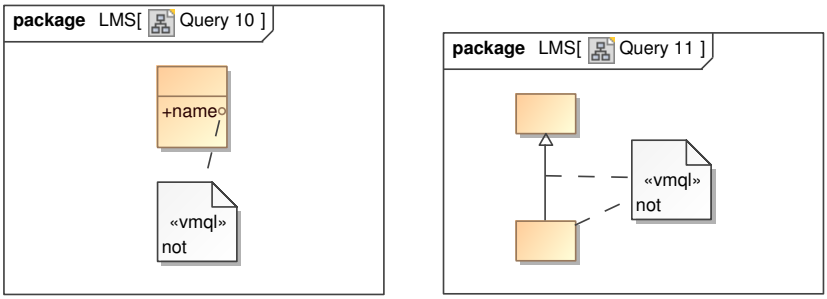
Query 8 in Figure 2.9(a) illustrates the either VMQL constraint. Suppose a



(a) (b)

Figure 2.9: VMQL queries for finding a class having a property expressed either as an attribute ((a)) or an association ((b))

modeler is looking for a class that has a "publisher" property, regardless if this property is expressed as an attribute or an association. The `either` constraint in Query 8 states that the leftmost class must either have a `publisher` attribute or be connected via a directed association to a `Publisher` class, *but not both*. As a result, this class will be bound to the `Medium` class in the source model, since this class is linked via a directed association to a class named Publisher. Figure 2.9(b) illustrates a similar scenario, retrieving the `Book` source model element. However, in this case the `isbn` property is present in the solution, not the `ISBN` class to which it is associated.



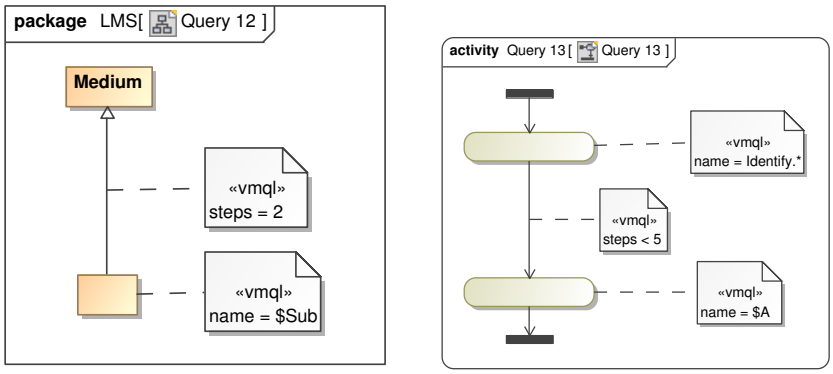
(a) (b)

Figure 2.10: VMQL queries for finding: (a) all classes which do not have a `name` public attribute, (b) all classes which do not have subclasses

The `not` constraint is used to prevent the query model elements to which it is anchored from being bound in a solution. It is exemplified by Query 10

in Figure 2.10(a), which retrieves all classes that do not have a `name` public attribute (not to be confused with the `name` meta-attribute, which all UML classes possess). That is, the query matches all source model classes with the exception of `Reader` and `Publisher`. Figure 2.12 illustrates the usage of the `not` constraint to identify all classes that do not have a subclass. Similarly to the `either` constraint discussed above, the `not` constraint must be applied to the subclass in the query model, as well as to the generalization relationship. Were the constraint not applied to the generalization relationship, the query would produce no solutions, since it would be interpreted as searching for all super classes connected to a generalization relationship that has no subclass attached to the other end.

2.5 Path-related constraints: `steps` and `indirect`



(a) (b)

Figure 2.11: VMQL queries exemplifying the `steps` constraint

VMQL's path-related constraints, `steps` and `indirect`, contribute significantly to increasing the language's expressiveness. Consider for instance Query 12 presented in Figure 2.11(a), which uses the `steps = 2` constraint to identify the classes inheriting from the `Medium` class through a path of exactly two generalizations - that is, the `Fiction` and `NonFiction` classes. This is a very concise manner of expressing what would otherwise be a complex query in a language such as OCL. VMQL also supports specifying an upper or lower limit to the number of steps in a relationship path, as exemplified by Query 13 in Figure 2.11(b) (specifying a lower limit is not currently implemented in MQ-2). This query retrieves all flow paths of length at most four that start with an `Action` node whose name contains the "Identify" prefix. Furthermore, the path must be enclosed

between a `Fork-Join` pair of model elements. Query 13 matches two paths from the `Lend Medium` Activity Diagram in the source model: The path starting with the `Identify medium` action and ending with the `Check availability` action, and the path starting with the `Identify reader` action and ending with the `Check eligibility` action. Notice that the types of the nodes included in the path are not relevant: both action nodes and object nodes can be included. The only constraint is that the link type between the nodes must be consistent with the type of the link to which the `steps` constraint is anchored in the query model. In Activity Diagrams, the `Object Flow` and `Control Flow` link types are considered equivalent.

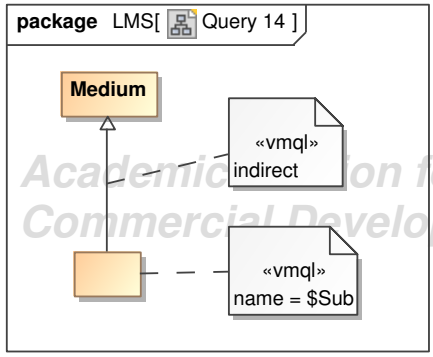


Figure 2.12: VMQL query exemplifying the `indirect` constraint

The `indirect` constraint showcased in Query 14 is a short-hand notation for specifying a path of undetermined length. It is equivalent to the `steps = *` constraint, where the * symbol is a wild card replacing any value. Consequently, Query 14 finds all transitive subclasses of the `Medium` class and binds their names to the `$Sub` variable. The `Book`, `Disc`, `Fiction`, and `NonFiction` classes are all found. This example highlights VMQL's ability to handle transitive closures, which is essential to the expressiveness of any model query language.

The VMQL constraints supported by MQ-2 are defined in Table 2.1, based on Table 1 in [Stö11b]. The `precision` and `strict` constraints have been considered as having low priority and have not been implemented in MQ-2, with the implementation effort being re-directed to the VMQL extensions presented in Section 6.1.

Table 2.1: VMQL constraints implemented by MQ-2

Constraint	Informal definition	Examples

`mattr`	Constrains the value of a meta-attribute. If given a value expression, the meta-attribute's value must conform to it. If given a variable (i.e.any expression starting with $), the value of the meta attribute is bound to that variable if possible.	`mattr isRoot = true`
		`mattr aggregationKind = composition; none`
		`mattr isAbstract = *`
		`mattr name = $N`

`name`	Alias for `mattr name = $N`	`name = $N`
`match`	Restricts the name of the constrained model element by a wild card expression or regular expression.	`match pa?tern.*`

`mclass`	Modifies the constrained element's meta-class.	`mclass = Class`
		`mclass = Class; Component`
		`mclass = *`
		`mclass <: Feature`

`once`	Enforces that a solution occurs only once in the set of all solutions.	`once`
`distinct`	Enforces that a set of constrained model elements are bound to distinct source model elements.	`distinct`
`optional`	Specifies that a constrained query model element may or may not have a binding in the result.	`optional`
`either`	Allows only one of the set of constrained model elements to appear in a result.	`either`
`not`	Prevents a result from containing a binding for the constrained model element.	`not`

`steps`	Defines the length of a path between two connected model elements. Only one type of relationship may occur on the path. Applicable values are integers >0 or * for arbitrary length >0. Applicable only to elements that are subclasses of `Relationship`.	`steps = 3`
		`steps < 3`
		`steps = *`

`indirect`	Alias for `steps = *`	`indirect`
`precision`	Reduces the matching precision level to values below 1.	`precision = 0.8`
`indirect`	Enforces that a query model element must match exactly one result model element.	`strict`

Chapter 3

System analysis

The overall objective of this thesis is to create a fully functional implementation of VMQL that appeals to the needs of modelers coming from a business background while also providing more advanced features suitable for expert modelers. The implementation must be created as an extension to an existing modeling tool and must preferably be portable across several operating systems.

At the highest level, users of MQ-2 must be able to perform two use cases: querying a model through a Prolog console, and querying a model using VMQL. The first requirement is targeted at experienced modelers that are also familiar with Prolog, while the second must ensure a minimal learning curve for novice modelers. This pair of fundamental use cases helps divide MQ-2 into two subsystems, as shown in Figure 3.1.

The use cases envisioned for MQ-2's integrated Prolog console are:

- **Formulate Prolog query:** Modelers must be able to formulate Prolog queries via MQ-2's Prolog console user interface.
- **Execute Prolog query on model:** Queries must be executable on the Prolog representation of a model. The model could be the one currently open in MagicDraw or another locally stored model.

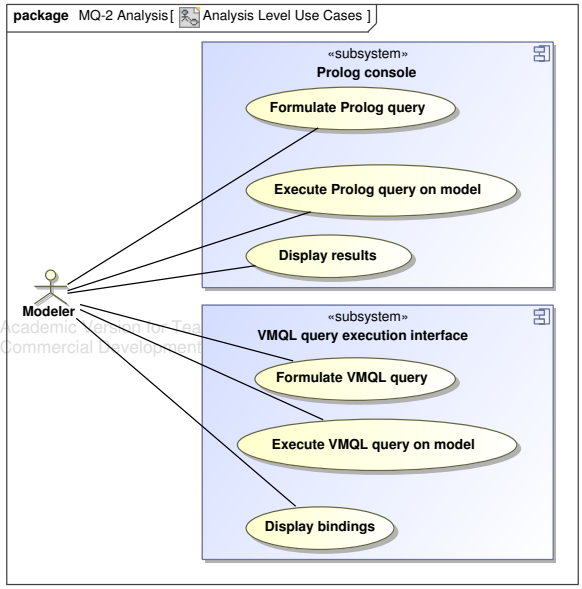


Figure 3.1: MQ-2 analysis level use cases

- **Display results:** Query results must be displayed both textually in the console and graphically by highlighting them on the model's diagrams. Integration with the host modeling tool's query results display facilities is also desirable.

The use cases envisioned for MQ-2's VMQL query execution interface are:

- **Formulate VMQL query:** Modelers must be able to formulate VMQL queries by creating query models annotated with VMQL constraints.
- **Execute VMQL query on model:** Modelers must be able to execute VMQL queries by selecting a query model to be matched against a selected source model.
- **Display bindings:** The bindings resulting from the execution of a VMQL query must be displayed by highlighting them on the source model. If a query has produced several bindings, modelers must be able to select one binding to display.

Chapter 4

Implementation

This chapter describes the implementation of the requirements identified in Chapter 3, from the overall architecture down to more detailed descriptions of its components. It also presents the difficulties encountered and the testing methodology employed in order to verify the implementation's conformance to the VMQL specification.

4.1 System architecture

MQ-2 is implemented as a plug-in to the popular MagicDraw CASE tool. It is integrated with an underlying SWI-Prolog[1] instance responsible for both direct and VMQL-based query execution. Given that MQ-2 is implemented in Java, communication between MQ-2 and SWI-Prolog is carried out through the JPL Java-Prolog bridge[2]. The high level architectural of MQ-2 is presented in Figure 4.1.

MagicDraw performs a two-fold role in this architecture. First, it acts as a model repository for source models and VMQL query models. Second, it provides an

[1] http://www.swi-prolog.org/
[2] http://www.swi-prolog.org/packages/jpl/

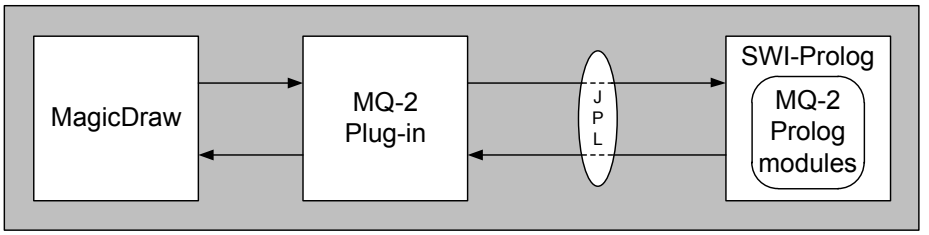


Figure 4.1: MQ-2 architecture. Arrows indicate data flow direction.

interface for creating and manipulating these models. MagicDraw has been selected as a basis for MQ-2's implementation with several considerations in mind. It is a mature modeling tool, offering full support for UML 2.x. It is also a cross-platform tool currently supported on Windows, Linux, and Mac OS X, thus satisfying MQ-2's portability requirement identified in Section 3. But more importantly, MagicDraw supports the creation of plug-ins through the MagicDraw Open API. Since MagicDraw is itself a Java application, the Open API provides a means of utilizing a large number of internal MagicDraw Java classes, both for extending the tool's user interface and manipulating the internal representation of models. Further details regarding the implementation of the MQ-2 plug-in based on the MagicDraw Open API are provided in Section 4.2.

While MagicDraw provides the infrastructure on which MQ-2 is built, actual query execution is carried out by SWI-Prolog. After a model has been transformed into a Prolog facts database (see Section 4.3.1 for details on this process), it is available for querying through the integrated Prolog console. SWI-Prolog has been selected from several available Prolog implementations in view of the fact that previous work on transforming models to Prolog fact databases has been carried out using this Prolog implementation. Furthermore, SWI-Prolog is an open-source cross-platform Prolog implementation that meets the requirement for portability placed on MQ-2. VMQL queries are also executed by SWI-Prolog by passing the Prolog representations of an annotated query model and source model to the matching algorithm described in Section 4.3.2.

4.2 The MQ-2 plug-in

The MQ-2 plug-in is implemented according to the guidelines laid out in the MagicDraw Open API user guide [Ope10]. It consists of:

- A JAR archive containing the MQ-2 plug-in implementation
- An XML file describing the plug-in's properties and dependencies
- Several folders containing resources such as image files, Prolog modules and the JPL library

This section focuses on the Java-based part of the plug-in's implementation, which is split into the five packages shown in Figure 4.2. The implementation consists of 43 Java classes, totaling 2151 lines of code.

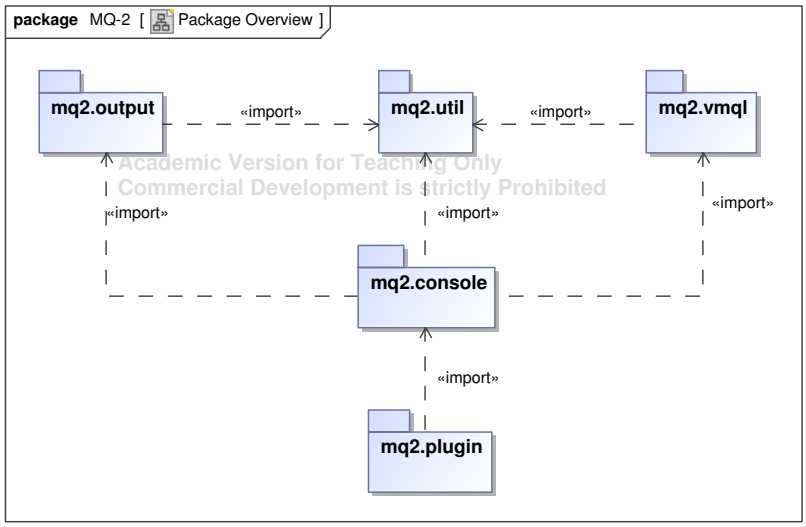


Figure 4.2: MQ-2 packages

The `mq2.plugin` package contains code responsible for exposing the MQ-2 plug-in to MagicDraw at run-time and adding an MQ-2 entry to the standard MagicDraw Tools menu. The `mq2.console` package (detailed in Section 4.2.1) handles the tasks of extending the MagicDraw UI to include a Prolog console and sending queries typed at the console to an underlying Prolog instance for execution. The `mq2.vmql` package (detailed in Section 4.2.2) provides functionality for invoking the VMQL matching algorithm implemented in Prolog and processing the results returned by this algorithm. Since the VMQL-related parts of the user interface are integrated into the Prolog console, they reside in the `mq2.console` package with the rest of the MQ-2 UI components. This aspect accounts for the import relationship between the `mq2.console` and `mq2.vmql` packages. The

`mq2.output` package (detailed in Section 4.2.3) offers query result display facilities for both console queries and VMQL queries. Finally, the `mq2.util` package (detailed in Section 4.2.4) encapsulates classes providing common utility methods used in other parts of the application for interacting with the JPL library and the MagicDraw Open API. For this reason, it is imported by all packages involved in query execution.

4.2.1 Integrated Prolog console

The implementation of the Integrated Prolog console is based on the analysis level use cases identified in Chapter 3, which are mapped to the implementation level use cases shown in Figure 4.3 as follows:

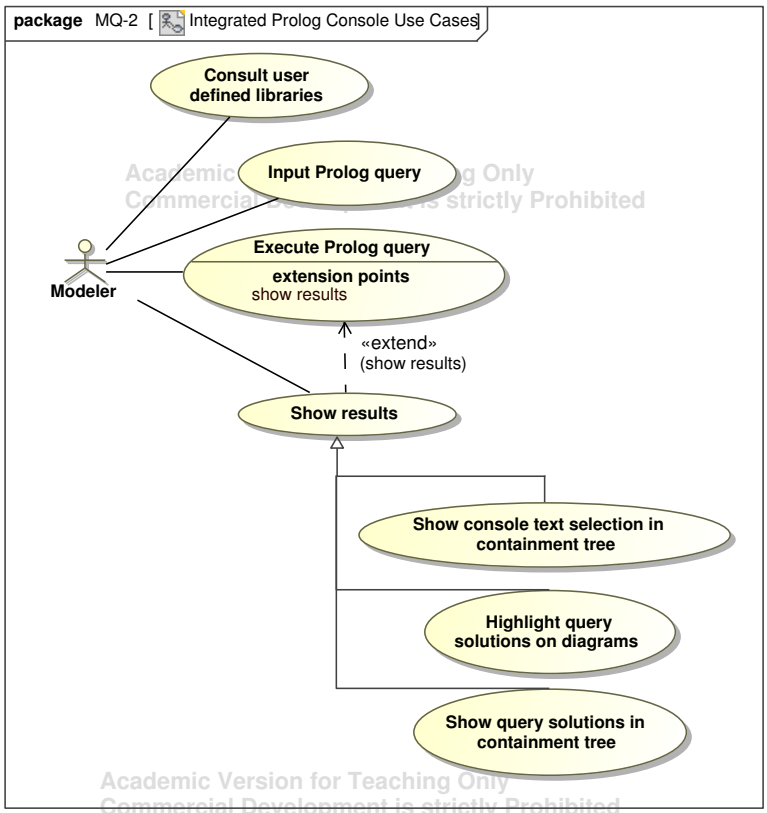


Figure 4.3: Integrated Prolog console implementation level use cases

- The *Formulate Prolog query* analysis level use case corresponds to the *Consult user defined libraries* and *Input Prolog query* implementation level use cases, implemented by the `mq2.console` package. The process of query formulation is facilitated by the existence of a query history maintained by the console and accessible through the *up* and *down* arrow keys. Users can consult Prolog libraries placed in a pre-defined directory at run-time without re-starting MagicDraw.
- The *Execute Prolog query on model* analysis level use case corresponds directly to the *Execute Prolog query* implementation level use case, implemented by the `mq2.console` package.
- The *Display query results* analysis level use case corresponds to the *Show results* use case and its sub use cases, implemented by the `mq2.console` package in conjunction with the the `mq2.output` package.

The structure of the `mq2.console` package is presented in Figure 4.4. The central component of this package is the `PrologConsole` class, which leverages methods provided by the JPL library to send queries for execution to an underlying SWI-Prolog connection. An instance of the `ConsolePanel` class, which extends `javax.swing.JPanel`, manages the GUI components of the console, including the Console Tool Bar (implemented by the `ConsoleToolBar` class, which extends `javax.swing.JToolBar`) and the console text area (represented by the `ConsoleTextArea` class, which extends `javax.swing.JTextArea`). The various actions available to users are accessible through buttons placed on the Console Tool Bar, each associated to an action listener in standard Java fashion. The `PrologConsole` class makes use of utility methods from the `MQ2Util` and `MagicDrawUtil` classes, which are not part of the `mq2.console` package but are included in the diagram for completeness. The same observation applies to the `QuerySolutionDisplayer` class.

At console start-up time, several MQ-2 specific Prolog modules are consulted through the constructor of the `PrologConsole` class. These include modules responsible for transforming models to Prolog facts databases, executing VMQL queries and providing library predicates such as the `highlight/2` predicate used to highlight model elements in diagrams. In addition, all user defined Prolog modules placed in the plug-in's *user* directory are consulted.

Before a model can be queried through the Prolog console, it is transformed to a Prolog facts database which must also be consulted in the underlying SWI-Prolog engine. These tasks are carried out by the `useMDProject()` method of the `PrologConsole` class. Re-consulting the current model is carried out by the `refresh()` method, while a full re-start of the console is performed through the `reset()` method.

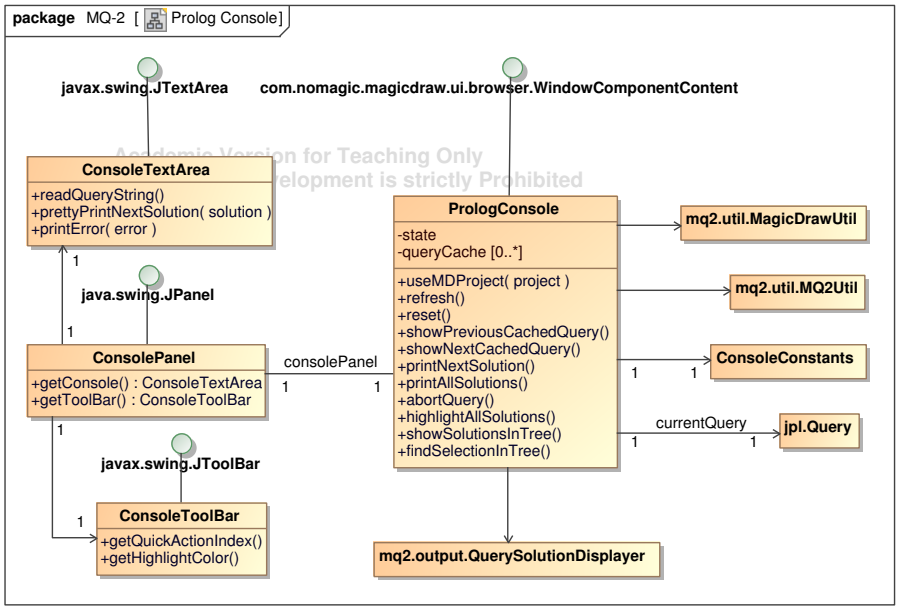


Figure 4.4: Classes implementing the Prolog console

Queries typed into the console are executed via the `printNextSolution()` method of the `PrologConsole` class. A notable aspect is that one user action (pressing the *return* key) may have two distinct meanings: triggering the execution of a new query and printing the next result of the current query. For this reason, the console must maintain an internal state taking either the *Waiting for query* or *Result displayed* value. This internal state is maintained in conjunction by the `printNextSolution()` and `abortQuery()` methods, which together implement the finite state machine presented in Figure 4.5.

To further emulate the expected behavior of a Prolog console, all executed queries are stored in a query history internal to the `PrologConsole` class, in the `queryCache` property of this class.

Query strings typed into the console are forwarded to SWI-Prolog via the JPL library, which returns each result in the form of a hash table in which every key is a query variable. The value associated to a key is the term to which the respective variable is bound. The JPL library provides a dedicated data structure for Prolog terms, consisting of the abstract base class `Term` extended by concrete classes for every Prolog data type, as shown in Figure 4.6. Given

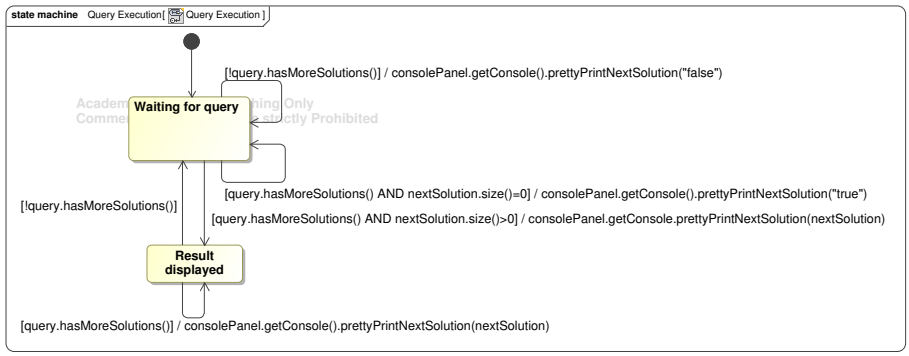


Figure 4.5: State Machine Diagram describing the internal state transitions of the Prolog console

the presence of this data structure in the JPL library, implementing an MQ-2 specific representation for Prolog terms is no longer necessary.

The consulted model can be queried either through the MQ-2 Prolog console using the `me/2` predicate, the `get_me/3` MQ-2 library predicate, or a user defined library predicate. Highlighting the model elements included in the results of such queries is facilitated by the `highlight/2` MQ-2 library predicate. However, the implementation of this library predicate is somewhat unusual, in that only its signature is defined in the MQ-2 Prolog library. The predicate is actually detected using a regular expression on the Java side of the MQ-2 plug-in and removed before queries containing it are forwarded to SWI-Prolog. Its arguments are extracted and provided as parameters to the solution highlighting methods in the `vmql.output` package. This approach has been adopted due to the fact that model element highlighting must be performed through the MagicDraw Open API, and implicitly in Java. The signature of the predicate is only defined in the MQ-2 library to prevent SWI-Prolog from producing an error when encountering it (otherwise, SWI-Prolog would correctly identify the `highlight/2` predicate as undefined.)

In addition to the standard key-based interaction style, queries may be executed by using one of the buttons on the Console Tool Bar. These buttons expose MQ-2 specific query execution options. *Print all solutions* is one of these options, introduced in order to facilitate query result display given that a query of the form `me(Type-Id,Properties)` (returning all model elements) will likely produce a large number of results, presumably tedious to display one at a time. The repeated execution of a query to the point where all of its solutions have been exhausted is implemented by the `printAllSolutions()`

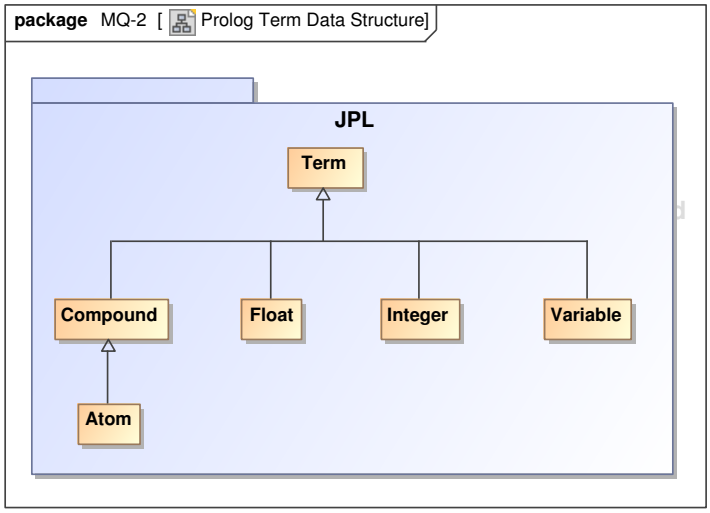


Figure 4.6: The Prolog term data structure provided by JPL

method of the `PrologConsole` class, which in turns makes repeated calls to the `printNextSolution()` method. The `highlightAllSolutions()` and `showSolutionsInTree()` methods extend the behavior of the `printAllSolutions()` method by visually highlighting query solutions in the current project. Model element highlighting is carried out by the `QuerySolutionDisplayer` class, detailed in Section 4.2.3.

Once the solutions of a query have been displayed in the Prolog console, the requirements of MQ-2 state that users must be able to select a section of text from the console and view the model elements which it identifies. This is possible if the selected text represents an original XMI ID of a model element, a generated ID created for a model element at Prolog transformation time, or the name of a model element. This feature is implemented in the `findSelectionInTree()` method of the `PrologConsole` class, which wraps around the method of the same name of the `QuerySolutionDisplayer` class.

In case a query is particularly slow to execute, it may be aborted through the *Abort* button of the Console Tool Bar. At the implementation level, the `abortQuery()` method of the `PrologConsole` class wraps around the `close()` method provided by the JPL `Query` class, which effectively terminates the query's execution in the SWI-Prolog engine.

Testing the MQ-2 plug-in has revealed the fact that using the Console Tool Bar buttons to perform query execution and result highlighting tasks is sometimes not appropriate for a console style of interaction. To address this issue, the option to modify the behavior associated to pressing the *return* key has been introduced. The Console Tool Bar contains a drop-down list containing several highlighting and query result display options (informally dubbed *Quick Actions*) which mimic the functionality of each tool bar button. The selected quick action is performed each time the *return* key is pressed, augmenting the standard console behavior. At the implementation level, the `printNextSolution()` method of the `PrologConsole` class - which is called at each press of the *return* key - makes a call to the `getQuickActionIndex()` method of the `ConsoleToolBar` class to determine the action to perform.

4.2.2 VMQL query execution

The implementation of VMQL query execution support is based on the analysis level use cases identified in Chapter 3, which are mapped to the implementation level use cases shown in Figure 4.7 as follows:

- The *Formulate VMQL query* analysis level use case corresponds to the *Select source and query models* implementation level use case, implemented by the `mq2.vmql` package. VMQL queries are formulated as regular MagicDraw models, with optional additional VMQL constraints expressed as comments.
- The *Execute VMQL query* analysis level use case corresponds to the implementation level use case of the same name. This use case is implemented by the `mq2.vmql` package in conjunction with the matching algorithm described in Section 4.3.2.
- The *Display bindings* analysis level use case corresponds to the *Highlight bindings* implementation level use case and its sub use cases, implemented by the `mq2.vmql` package in conjunction with the `mq2.output` package.

Since VMQL queries are executed in Prolog, the Java side of the VMQL implementation merely provides a user interface through which the matching algorithm described in Section 4.3.2 is invoked and its results are displayed in a user friendly manner. Figure 4.8 presents the classes of the `mq2.vmql` package involved in query execution. Note that the `VMQLPanel` class is included in the `mq2.console` package due to the fact that the VMQL user interface is a part of

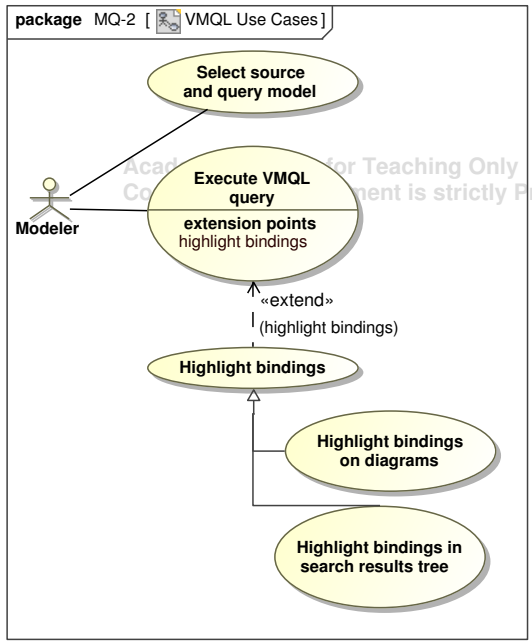


Figure 4.7: VMQL implementation level use cases

the Prolog console user interface - the VMQL interface elements can be shown or hidden at the click of a button.

Query execution is triggered through the `executeQuery()` method of the `VMQLPanel` class, which in turns invokes the method of the same name of the `QueryEngine` class. The `match/4` predicate is then invoked through JPL. As a prerequisite to invoking this predicate, the source and query models must be specified. While the source model is implicitly the model currently consulted in the Prolog console, the query model must be explicitly specified through the `setQueryFile()` method of the `VMQLPanel` class.

The result of invoking the `match/4` predicate is a Prolog list of query solutions, where each solution is a list of source model XMI element IDs. Displaying this list directly to users is not feasible from a usability standpoint. Thus, the query results must be processed further. Results are displayed in an instance of the `javax.swing.JTable` class through a custom table model implemented by the `ResultsTableModel` class. Based on the results obtained from the JPL invocation of the `match/4` predicate, this class provides implementations for

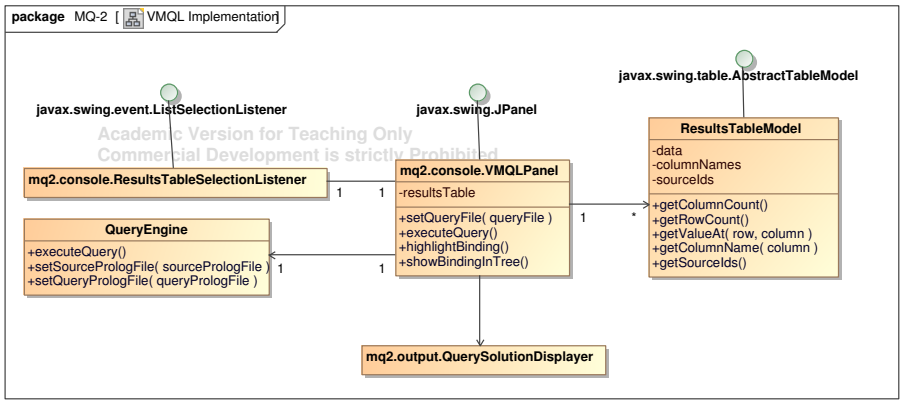


Figure 4.8: Classes implementing the VMQL front-end

the methods required by the `javax.swing.table.AbstractTableModel` class, which it extends.

For visualizing VMQL query results, a `ListSelectionListener` implemented by the `ResultsTableSelectionListener` class is associated to the results display table. When a user selects a solution by clicking on a row of the table, this listener invokes the `displayAsHighlightedElements()` method of the `QuerySolutionDisplayer` class in order to show the model elements included in the solution in the MagicDraw Search Results Tree. A button on the VMQL Panel allows users to highlight the selected solution by invoking the `displayAsHighlightedElements()` method of the `QuerySolutionDisplayer` class.

4.2.3 Query solution display methods

The methods employed to graphically display query solutions are shared by the Prolog console and the VMQL front-end. They have consequently been encapsulated in the `mq2.output` package, detailed in Figure 4.9.

All classes that require query solution display functionality must obtain an instance of the singleton `QuerySolutionDisplay` class via its `getInstance()` method. The `displayElementsInSearchResultsTree()` method of this class displays the model elements whose XMI IDs are provided as arguments in the MagicDraw Search Results Tree. This method of displaying search results is

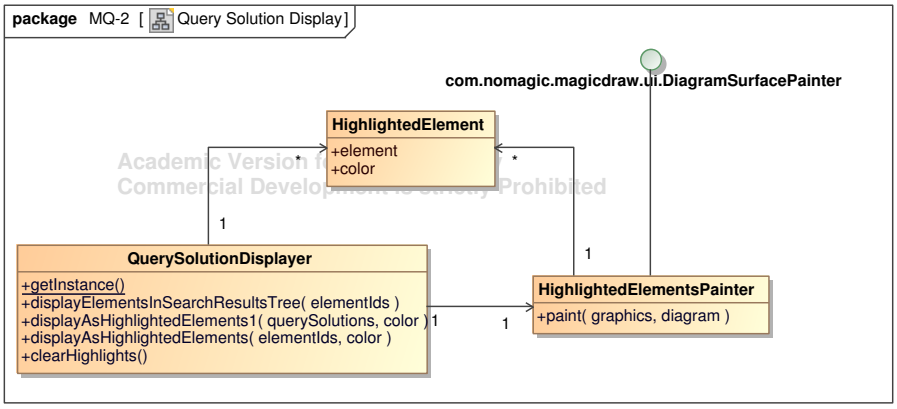


Figure 4.9: Classes implementing the query solution display methods

also adopted by MagicDraw's built-in search feature, making it familiar for existing users. However, displaying MQ-2 query results in this manner has raised some unexpected challenges. Adding elements to the Search Results Tree requires an undocumented use of the MagicDraw Open API, suggested by the MagicDraw support team. Namely, an instance of the undocumented `com.nomagic.magicdraw.ui.SearchResult` class must be created.

A more unique query solution display method supported by MQ-2 consists of highlighting solutions directly in relevant diagrams. Model elements are highlighted using a color selected by the user through a color picker included in the MQ-2 Prolog Console Tool Bar. This functionality is implemented by the `displayAsHighlightedElements()` method of the `QuerySolutionDisplayer` class. The method is overloaded to support highlighting a list of elements identified either by their XMI IDs or by a Prolog query result presented in JPL-specific format. Highlighting is carried out by adding a custom painter class (`HighlightedElementsPainter`) to the relevant diagrams. This class maintains a reference to the list of model elements to highlight and, through its `paint()` method, renders a rectangle of the appropriate color around these elements whenever they appear in a diagram. This highlighting method can also be applied to model elements which have a purely textual syntax, such as classifier attributes. Each highlighted model element is encapsulated in an instance of the `HighlightedElement` class, which also stores the highlight color.

Finally, the `QuerySolutionDisplayer` class exposes the `clearHighlights()` method, which removes all existing highlights. This is achieved by re-painting each diagram without making use of the custom painter class described above.

4.2.4 Internal MQ-2 utilities

The `mq2.util` package detailed in Figure 4.10 contains static utility methods required by several other classes or likely to be re-used by an extension to MQ-2. The methods are encapsulated in two classes: the `MQ2Util` class, which contains methods for processing the Prolog representation of a model, and the `MagicDrawUtil` class, which contains methods for processing the MagicDraw representation of a model.

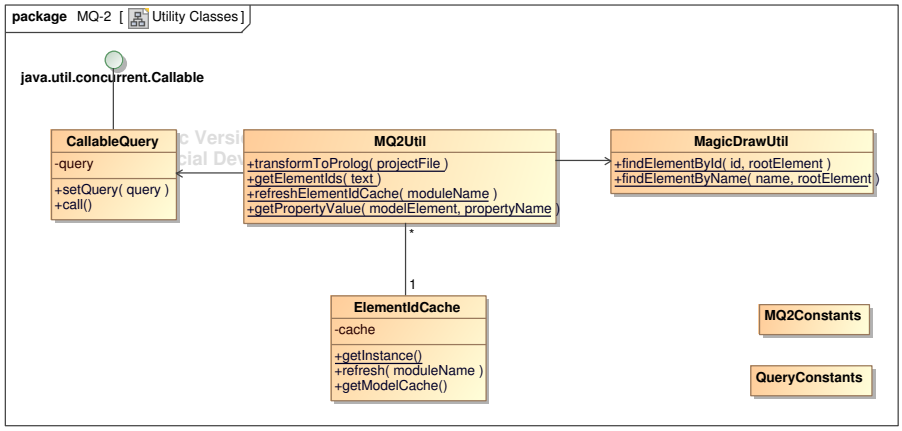


Figure 4.10: Classes implementing internal MQ-2 utilities

The `transformToProlog()` method of the `MQ2Util` class is called whenever a MagicDraw model must be transformed to a Prolog facts database. The first step in this process is to create an un-compressed copy of the model in the MDXML format (if the model is not already stored in this format), as the Prolog transformation algorithm does not support MagicDraw's compressed MDZIP format. The MDXML model storage format is a MagicDraw specific extension to the standard XMI format. Once the existence of an MDXML file describing the model is ensured, the name of this file is passed as a parameter to the Prolog transformation algorithm described in Section 4.3.1. Tests have shown that supplying this algorithm with a corrupt MDXML file may prevent it from terminating, a situation in which the MQ-2 plug-in becomes unresponsive. Situations of this type are avoided by enforcing a time-out on the JPL query triggering the transformation algorithm's execution. This is achieved by wrapping the query inside a class implementing the `java.util.concurrent.Callable` interface, namely the `CallableQuery` class, whose `call()` method triggers the query's execution. Thus, in the `transformToProlog()` method of the `MQ2Util` class, the query can be executed through the `get()` method of an instance of

the `java.util.concurrent.Future` class, which supports a time-out on the asynchronous task it is executing. This time-out is set to a default value of 10 seconds through a constant defined in the `MQ2Constants` class.

Perhaps the most interesting in terms of implementation is the `getElementIds()` method of the `MQ2Util` class. It takes a string as a parameter and returns the XMI IDs of the model elements identified by the string either by name or by their XMI or generated ID. The method is used, among others, for finding the model elements identified by a section of text selected in the Prolog console. When this method is called, it first identifies candidate substrings of the provided string, i.e. substrings that could represent model elements. For instance, substrings enclosed between single quotes are selected since they may represent model element names or XMI IDs in a Prolog query result. Each candidate substring is then subjected to a series of at most three tests, with the aim of determining the model element which it identifies (if one of the tests succeeds, the remaining tests are no longer performed). First, a check is performed to determine if the candidate substring represents the XMI ID of a model element from the current model. Second, a check is performed to determine if the candidate substring represents a model element name. These checks are performed by the `findElementById()` and `findElementByName()` methods of the `MagicDrawUtil` class, respectively. They do not involve executing any Prolog queries on the model. The final test applied determines if the candidate substring represents a generated model element ID created at Prolog transformation time. This would normally involve an extra Prolog query for retrieving all model elements in the Prolog facts database. However, this requirement is circumvented by maintaining a hash table that maps the correspondence between XMI and generated model element IDs. The hash table is encapsulated by the `ElementIdCache` class and refreshed each time a MagicDraw model is transformed to a Prolog facts database.

The `getPropertyValue()` method of the `MQ2Util` class retrieves the value of a given property for a model element represented as a Prolog fact of the form generated by the Prolog transformation algorithm. The method is called, for instance, when refreshing the model element ID hash table described in the previous paragraph.

4.3 MQ-2 Prolog modules

The MQ-2 plug-in relies on Prolog modules consulted at start-up to perform two operations: transforming MagicDraw models from their original XMI representation to Prolog facts databases, and executing VMQL queries based on the

Prolog representations of a source and a query model. The following subsections detail the implementation of these Prolog modules.

4.3.1 XMI to Prolog transformation

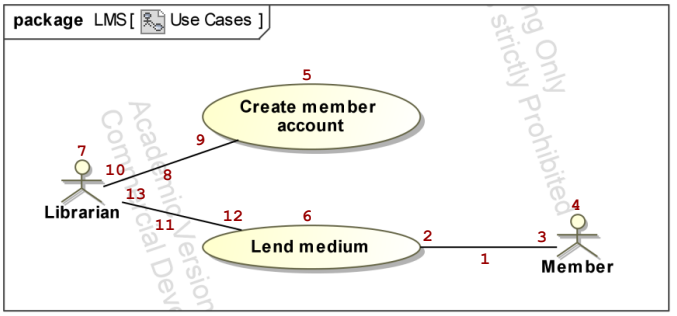
Models are stored by MagicDraw as files conforming to the proprietary MDXML format, which is based on the standard XMI format. XMI is in turns an XML-based file format specified by OMG with the goal of promoting interoperability between UML modeling tools by ensuring that models are stored in a tool-independent format. The existence of the XMI standard [XMI11] implies that MQ-2 may process UML models created using a wide range of tools with little or no extra implementation effort. However, any XML model representation is not appropriate for model manipulation using Prolog. In order to be queried from a Prolog console, a model must be represented as a set of Prolog facts conforming to a pre-determined format. Querying the model then becomes a simple matter of executing a query conforming to the same format. Transforming a model from its XMI representation to a Prolog representation is performed in MQ-2 by the `xmi2pl/1` predicate, implemented in a stand-alone Prolog module. It receives the name of an XMI file as input and produces a file representing the same model as a Prolog module.

The resulting Prolog module consists of instances of the `me/2` predicate introduced in [Stö07], one for each model element. The `me/2` predicate has the form:

```
me(type-id, [tag-value, ...]).
```

where `type` is the model element's meta-class, `id` is a unique generated identifier for the model element, `tag` is one of the model element's meta-attributes, and `value` is the corresponding meta-attribute value. A model element may have any number of `tag-value` pairs, stored as elements of a Prolog list. An example of how a model is represented in Prolog following this convention is presented in Figure 4.11. The considered model consists solely of a Use Case Diagram illustrating a considerably simplified version of the use cases and actors involved in a Library Management System (LMS) scenario. Below the diagram, the Prolog representation corresponding to this model is shown. To highlight the correspondence between model elements and Prolog facts, each element's ID is shown next to it on the diagram in red.

After consulting the generated 'Use Cases' module in a Prolog console, it may be queried by making use of the `me/2` predicate in conjunction with any standard




```
:-module('Use Cases',[me/2]).
   me(model-0,[ownedMember-ids([1,4,5,6,7,8,11]),name-'LMS',visibility-public).
   me(association-1,[ownedMember-ids([2,3]),visibility-public,
                     navigableOwnedEnd-ids([3,2]),memberEnd-ids([3,2])]).
   me(property-2,[visibility-private,type-id(6),association-id(1)]).
   me(property-3,[visibility-private,type-id(4),association-id(1)]).
   me(actor-4,[name-'Member',visibility-public]).
   me(useCase-5,[name-'Create member account',visibility-public]).
   me(useCase-6,[name-'Lend medium',visibility-public]).
   me(actor-7,[name-'Librarian',visibility-public]).
   me(association-8,[ownedMember-ids([9,10]),visibility-public,
                     navigableOwnedEnd-ids([10,9]),memberEnd-ids([10,9])]).
   me(property-9,[visibility-private,type-id(5),association-id(8)]).
   me(property-10,[visibility-private,type-id(7),association-id(8)]).
   me(association-11,[ownedMember-ids([12,13]),visibility-public,
                      navigableOwnedEnd-ids([13,12]),memberEnd-ids([13,12])]).
   me(property-12,[visibility-private,type-id(6),association-id(11)]).
   me(property-13,[visibility-private,type-id(7),association-id(11)]).
```

Figure 4.11: A simple model and its Prolog representation

Prolog library predicates supported by the console. For example, the IDs and meta-attributes of all actors included in the model can be retrieved through the following query:

```
me(actor-ID, MetaAttrs).
```

Model elements having a certain value for one of their meta-attributes can be retrieved by making use of the `memberchk/2` standard library predicate. As an example, the query below retrieves the ID and meta-attributes of the actor named 'Librarian':

```
me(actor-ID, MetaAttrs), memberchk(name-'Librarian', MetaAttrs).
```

Alternatively, the same query can be formulated in the MQ-2 Prolog console in a more concise manner by utilizing the `get_me/3` MQ-2 library predicate proposed in [Stö07]:

```
get_me(name-'Librarian', actor-Id, MetaAttrs).
```

Further details on how the Prolog representation of a model can be queried from the MQ-2 Prolog console are presented in Section 5.1. The `xmi2pl` module has not been developed entirely as part of the MQ-2 project, as it was considered to be outside of the project's scope. Instead, an existing implementation has been extended to meet the requirements of MQ-2. The two notable modifications required are described in what follows.

- Originally, the `xmi2pl/1` predicate only required an MDXML file name as a parameter. The predicate has been modified to accept an optional second parameter. If this parameter has the value `[explicit]`, the `me/2` predicate is included in the generated module's header. Otherwise, it is omitted from this header. This option is relevant for situations in which two models must be consulted in the same Prolog console, for instance for the purpose of version control. In such situations, the `me/2` predicate must be omitted from the two module's headers. Otherwise, an error will be produced by the Prolog engine upon consulting the second module.
- Since the element ID included in each `me/2` clause is generated at transformation time for readability purposes (XMI element IDs are considerably long strings of characters), the element's original XMI ID must be included in its list of `tag-value` pairs. Thus, the `xmi_id` tag having the model element's XMI ID as a value is now added at transformation time to each `me/2` clause.

4.3.2 VMQL matching algorithm

VMQL queries are executed by computing a set $B = \{b_1, b_2, ..., b_n\}$ of bindings between the elements of a query model and those of a source model. Each binding represents a query solution passed to the Java side of the `MQ-2` plug-in for display. A query may have no solutions, in which case $B = \emptyset$. A binding $b_i = \{(q\_id_1, s\_id_1), (q\_id_2, s\_id_2), ..., (q\_id_m, s\_id_m)\}$ is a set of m pairs consisting of the ID of a query model element and the ID of the matching source model element, where m is the number of elements of the query model. If a query model does not have a match in the source model, it is represented in

the binding as $(q\_id, NULL)$. Given the Prolog representations of a query and a source model obtained as described in Section 4.3.1, the bindings between these two models are computed by Algorithm 4.1. This algorithm, as well as all the subsequent algorithms presented in this section, are imperative representations of the logic behind the actual Prolog implementation. The corresponding Prolog source code is split into three modules, implementing the matching algorithm, VMQL constraint parsing, and a representation of relevant aspects of the UML meta-model, respectively. The modules consist of a total of 900 lines of code.

Algorithm 4.1 MATCH

```
1: Inputs:
2:     q_model - a VMQL annotated query model
3:     s_model - a source model
4:
5: Outputs:
6:     bindings - a list of bindings between q_model and s_model
7:     var_bindings - a list of bindings for the VMQL variables in q_model
8:
9: begin
10: c ← COLLECT_CONSTRAINTS(q_model)
11: ADD_CONSTRAINTS(q_model, c)
12: bindings ← GET_BINDINGS(q_model, s_model)
13: bindings ← FP_REFINE(bindings, q_model, s_model)
14: bindings ← GET_PAIRWISE_BINDINGS(bindings)
15: bindings ← VERIFY_CONSTRAINTS(bindings, c, q_model, s_model)
16: var_bindings ← GET_VAR_BINDINGS(bindings, c, s_model)
17: end
```

The *MATCH* algorithm takes as parameters a query model (*q_model*) and a source model (*s_model*), both represented as lists of `me/2` clauses. It produces a list of bindings (*bindings*), each representing a query solution, and a list of variable bindings (*var_bindings*), each representing the values of the VMQL variables in the query model for the corresponding query solution. Note that *bindings* and *var_bindings* are ordered lists of the same length, and a correspondence exists between elements located on equal positions in these lists. A variable binding is a list of pairs of the form (var_name, var_value), associating a variable to its value.

The first step of the matching process is to collect the constraints included as comments in the query model and store them in a separate list, a task performed by the COLLECT_CONSTRAINTS algorithm. Some of the collected constraints require modifications to the query model prior to the computation of bindings. These modifications are performed by the ADD_CONSTRAINTS

algorithm. The initial bindings can then be computed by the GET_BINDINGS algorithm (see Algorithm 4.2). The initial bindings only take into account model element types and attribute values, while ignoring references (links) between model elements. The FP_REFINE algorithm (see Algorithm 4.3) applies a fixed point refinement process to the initial bindings by considering the links between model elements: if two model elements are linked in the query model, their bindings must also be linked in the source model. Up to this point in the matching process (Line 14), the bindings are stored as a single list of pairs associating each query model element ID to a *set* of source model element IDs. The GET_PAIRWISE_BINDINGS algorithm is invoked in order to transform this representation into the list B of lists b_i described above, where each binding b_i consists of pairs of the form (q_id, s_id) - that is, a one-to-one correspondence between query model elements and source model elements. The last step in the binding computation process is to eliminate those bindings that do not satisfy the VMQL constraints included in the query model. This step is carried out by the VERIFY_CONSTRAINTS algorithm. Once the model element bindings are computed, the query model variables can be assigned corresponding values for each binding by the GET_VAR_BINDINGS algorithm.

The COLLECT_CONSTRAINTS algorithm takes the Prolog representation of a model as input and produces a list of *constraints* as output. The constraints are extracted from the comments included in the model - note that a comment may be anchored to several model elements. The text of the comment is parsed according to a grammar describing the syntax of VMQL constraints. This parsing process is carried out by a separate Prolog module that utilizes Prolog's built in facilities for parsing context free grammars. Namely, the $-->$ predicate is used to specify definite clauses in a manner closely resembling a BNF (Backus-Naur Form) specification of the VMQL grammar. In case parsing succeeds, the comment represents a VMQL constraint and must be added to the list of constraints. The returned list of constraints also contains implicit constraints, such as the *optional* constraint added to elements referenced by a model element annotated with a *steps* constraint.

The GET_BINDINGS algorithm (Algorithm 4.2) takes a query and a source model as inputs, and produces a list representing the initial bindings between these models as output. The resulting bindings are represented as a list with elements of the form (q_id, s_ids), where q_id is a query model element ID and s_ids is a list containing the IDs of the source model elements that match the type and meta-attribute values of s_id. The algorithm starts by initializing an empty bindings list (Line 9) and iterates through each element of the query model in order to discover its bindings. Some element types, such as comments, must not be bound (Line 11). For each query model element, an inner loop iterates through all source model elements and checks if their type and meta-attribute values match those of the current query model element (Lines 14-19).

Algorithm 4.2 GET_BINDINGS

```
1:  Inputs:
2:      q_model - a query model
3:      s_model - a source model
4:
5:  Outputs:
6:      bindings - an initial list of bindings
7:
8:  begin
9:  bindings ← ∅
10: for all q_me ∈ q_model do
11:   if !SKIP_TYPE(q_me) then
12:     b ← ∅
13:     q_id ← GET_ID(q_me)
14:     for all s_me ∈ s_model do
15:       if (MATCH_TYPE(q_me, s_me) ∧
              MATCH_ATTRS(q_me, s_me)) then
16:         s_id ← GET_ID(s_me)
17:         b ← b ∪ {s_id}
18:       end if
19:     end for
20:     bindings ← bindings ∪ {(q_id, b)}
21:   end if
22: end for
23: return bindings
24: end
```

If this is the case, the ID of the considered source model element is appended to the list of source element IDs bound to the current query model element (Line 17). When all source model elements have been considered, the new binding is stored (Line 20). In case no source model elements match the query model element, the new binding pair will contain an empty list as its second element.

Although not explicitly presented here, the MATCH_ATTRS function called by the GET_BINDINGS algorithm requires some clarifications. First, it only considers meta-attributes with fixed values, ignoring meta-attributes representing references to neighbor model elements. Second, it ignores meta-attributes that are not relevant for matching, such as the *xmi_id* attribute introduced by the Prolog transformation algorithm. Finally, it supports matching regular expressions through SWI-Prolog's PCE library.

The FP_REFINE algorithm (Algorithm 4.3) takes as inputs a list of bindings

Algorithm 4.3 FP_REFINE

```
 1: Inputs:
 2:     bindings - a list of bindings
 3:     q_model - a query model
 4:     s_model - a source model
 5:
 6: Outputs:
 7:     r_bindings - a refined list of bindings
 8:
 9: begin
10: old_bindings ← bindings
11: new_bindings ← bindings
12: repeat
13:   old_bindings ← new_bindings
14:   new_bindings ← ∅
15:   for all (q_id, s_ids) ∈ old_bindings do
16:     new_s_ids ← ∅
17:     for all s_id ∈ s_ids do
18:       if VERIFY_LINKS(old_bindings, q_id, s_id, q_model, s_model)
          then
19:         new_s_ids ← new_s_ids ∪ {s_id}
20:       end if
21:     end for
22:     new_bindings ← new_bindings ∪ {(q_id, new_s_ids)}
23:   end for
24: until old_bindings = new_bindings
25: return new_bindings
26: end
```

along with a source and a query model and produces a refined list of bindings computed by executing a fixed point refinement algorithm on the original bindings. The algorithm takes into account the references between query model elements, which are ignored by the GET_BINDINGS algorithm. The fixed point refinement process is performed by a `repeat` loop (Lines 12-24) that terminates upon the convergence of two binding lists: *old_bindings* and *new_bindings*. At each step of the loop, a new version of the bindings is computed by considering each (q_id, s_ids) binding of the old version and verifying its consistency with the rest of the bindings through the VERIFY_LINKS function. Namely, for each $s\_id \in s\_ids$, the VERIFY_LINKS function checks that if the query model element identified by q_id references a query model element with ID ref_q_id, then there exists a binding (ref_q_id, ref_s_ids) such that $s\_id \in ref\_s\_ids$. In other words, model element references in the query

model must be reflected by the source model. s_id is only maintained as a member of s_ids if it respects this condition (Lines 18-20). If after an execution of the `repeat` loop no changes have been made to the list of bindings, a fixed point has been reached and the bindings can be returned (Line 25).

To show the necessity of a fixed point algorithm in this situation, consider two bindings (q_id1, s_ids1) and (q_id2, s_ids2), where a meta-attribute of the query model element identified by q_id1 holds a reference to q_id2. This implies that for every $s\_id1 \in s\_ids1$ there exists $s\_id2 \in s\_ids2$ such that the same meta-attribute of the source model element identified by s_id1 holds a reference to s_id2. In case s_id2 is removed from s_ids2, this condition no longer holds and the validity of the (q_id1, s_ids1) binding must be checked again. Hence, an iterative algorithm with an appropriate stop criterion is required. A fixed point algorithm meets this requirement.

After bindings are computed and each binding of the form (q_id, s_ids) is split into a list of pairwise bindings containing one pair of the form (q_id, s_id) for each $s\_id \in s\_ids$, the VERIFY_CONSTRAINTS algorithm eliminates the bindings that do not meet the VMQL constraints included in the query model. This is achieved by iterating through the list of bindings and verifying each type of constraint. All constraint types except *mattr* and *mclass* (which are handled by the ADD_CONSTRAINTS function) are considered here. The implementations of the *distinct* and *steps* constraints are detailed in Algorithm 4.4 and Algorithm 4.5, respectively. The implementations of the *once*, *either*, and *not* constraints are not discussed, since they are highly similar to the implementation of the *distinct* constraint.

The VERIFY_DISTINCT function (Algorithm 4.4) receives a pairwise binding and a list of constraints as inputs, and produces a boolean result signaling whether the binding respects all *distinct* constraints in the provided constraints list. It loops through all the constraints in the list and finds those matching the expected format for *distinct* constraints (Line 10). It then loops through all (q_id, s_id) pairs (where q_id is a query model element ID and s_id is a source model element ID) in the binding and uses the $distinct\_ids$ list to collect all values of q_id that belong to the list of IDs of elements to which the current *distinct* constraint is anchored. If at the end of this loop the $distinct\_ids$ list contains duplicate entries, it follows that the *distinct* constraint is not satisfied and the function must return `FALSE` (Lines 17-19). If all *distinct* constraints are satisfied, the function returns `TRUE`.

Similarly to the VERIFY_DISTINCT function, the VERIFY_STEPS function (Algorithm 4.5) receives a pairwise binding and a list of constraints as inputs. In addition, it receives a source model and a query model. It produces a boolean result signaling whether the binding respects all *steps* constraints in the pro-

Algorithm 4.4 VERIFY_DISTINCT

```
 1: Inputs:
 2:     binding - a pairwise binding
 3:     constraints - a list of constraints
 4:
 5: Outputs:
 6:     distinct - whether or not binding respects all distinct constraints
 7:
 8: begin
 9: for all c ∈ constraints do
10:   if c = (ids,'distinct') then
11:     distinct_ids ← ∅
12:     for all (q_id,s_id) ∈ binding do
13:       if q_id ∈ ids then
14:         distinct_ids ← distinct_ids ∪ {q_id}
15:       end if
16:     end for
17:     if !IS_SET(distinct_ids) then
18:       return FALSE
19:     end if
20:   end if
21: end for
22: return TRUE
23: end
```

vided constraints list. Upon identifying a *steps* constraint in this list (Line 12), it determines the type of relationship to which the constraint is anchored, as well as the IDs of the start and end points of the relationship (Lines 13-15). Note that *steps* constraints may only be anchored to so-called model elements representing relationships. The currently supported relationships are associations, generalizations, inclusions, control flows and extensions. As an example, if in a UML Class Diagram a *steps* constraint is anchored to a generalization relationship showing that class A is a superclass of class B, the VERIFY_STEPS algorithm will identify class B as a starting point and class A as an end point. After the extremities of the relationship are identified in the query model, a check is performed to determine whether the source model contains a path of the length specified in the constraint and consisting exclusively of relationships of the same type between the determined extremities (Lines 16-18). This check is performed by the EXISTS_PATH function, which utilizes Prolog's built-in backtracking to perform the search. If this is not the case, the *steps* constraint fails and the VERIFY_STEPS function must return `FALSE`. Note that the EXISTS_PATH function also takes a comparison operator specified by the *steps*

Algorithm 4.5 VERIFY_STEPS

```
1:  Inputs:
2:      binding - a pairwise binding
3:      constraints - a list of constraints
4:      q_model - a query model
5:      s_model - a source model
6:
7:  Outputs:
8:      distinct - whether or not binding respects all steps constraints
9:
10: begin
11: for all c ∈ constraints do
12:   if c = (id,'steps',comp,limit) then
13:     link_type ← GET_LINK_TYPE(id, q_model)
14:     start_id ← GET_START(id, q_model)
15:     end_id ← GET_END(id, q_model)
16:     if !EXISTS_PATH(start_id,end_id,link_type,comp,limit) then
17:       return FALSE
18:     end if
19:   end if
20: end for
21: return TRUE
22: end
```

constraint as a parameter. This comparison operator determines how the length of the identified path is verified. Currently, the supported comparators are '=' (for specifying paths of a fixed length) and '<' (for specifying the maximum allowed path length). If all *steps* constraints are satisfied, the function returns `TRUE`.

4.4 Encountered implementation difficulties

The MQ-2 plug-in relies on the JPL Java library to enable the execution of Prolog predicates from Java. This ability is vital to the plug-in's functioning, being used both to forward queries typed in the integrated Prolog console to the underlying SWI-Prolog engine and to invoke the the `xmi2pl/1` model transformation predicate, as well as the `match/1` query execution predicate. However, several unexpected problems have been encountered in the usage of JPL. Some of the problems have been caused by the JPL library itself, and some by unexpected behavior on the part of SWI-Prolog. This section briefly documents the

encountered setbacks, since a considerable amount of effort has been spent in handling them.

The first encountered issues concern cross-platform portability. Though MQ-2 is implemented purely in Java and Prolog, it requires separate distributions for 32-bit and 64-bit Windows operating systems. This is due to the fact that the JPL library, included with the MQ-2 distribution, relies on low level system calls that appear to be incompatible between the two versions. Furthermore, the version of JPL used with the MQ-2 plug-in must correspond to the installed version of SWI-Prolog. This implies that users must replace the JPL library included with the MQ-2 plug-in with the version of the JPL library included with their respective SWI-Prolog installations. While this is not a major setback for Windows users, it has a bigger impact on Linux users who are required to compile JPL for their particular distribution, since it appears that the compiled JPL library included with SWI-Prolog's Linux distribution is not functional on most systems.

A second problem has been caused by SWI-Prolog's unusual handling of errors occurring during predicate and module consulting via the `consult/1` and `use_module/1` built-in predicates, which both take a file name as an argument. This file name presumably identifies a file containing Prolog predicate definitions. However, in case it does not or in case the identified file cannot be found, the two mentioned predicates still succeed, only printing a warning message. Because JPL only passes Prolog errors to Java encapsulated inside Java exceptions and ignores Prolog warnings, it is in fact impossible to determine from the Java side of the implementation if consulting a file has succeeded. This aspect may affect modelers that wish to consult their own Prolog modules at run-time through the MQ-2 Prolog console. However, an inquiry to the official SWI-Prolog mailing list[3] has revealed the fact that this problem has been resolved in the development version of SWI-Prolog, which correctly produces errors upon failing to consult a file.

Finally, what is perhaps the most significant problem encountered can lead to unexpected MagicDraw crashes. It can be exemplified by considering the following scenario. Assume that two MagicDraw models, *Model_A* and *Model_B*, have been transformed to Prolog using the `xmi2pl/1` predicate, and both expose the `me/2` predicate. Assume now that they are both consulted from MQ-2's built-in Prolog console using the following two queries:

```
?- use_module('Module_A').
?- use_module('Module_B').
```

[3] https://lists.iai.uni-bonn.de/mailman/listinfo.cgi/swi-prolog

The second query fails, since the `me/2` predicate has already been imported as a result of executing the first query. This behavior is entirely expected. What is unexpected is the fact that SWI-Prolog enters debug mode after displaying an error message, which causes a crash of the Java Virtual Machine (JVM) if the two queries are issued through JPL. Naturally, this leads to a MagicDraw crash, since MagicDraw is itself a Java application. However, the behavior of SWI-Prolog that leads to this crash has also been resolved in its development version and will likely no longer occur in future versions of SWI-Prolog.

4.5 Unit testing

The development of the matching algorithm has followed the test-driven approach shown in Figure 4.12. After implementing support for base queries, the constraints proposed in the VMQL specification were prioritized according to their influence on the language's expressiveness: constraints adding a considerable expressiveness improvement to the implementation received priority. Once the next constraint to implement was determined, a series of test cases illustrating the constraint's usage were developed. The constraint would then enter a cycle of iterative development and testing, until all of its test cases were satisfied. At this point, regression tests were performed in order to mitigate the possibility that the most recent constraint's development has interfered with the evaluation of other constraints. In some cases, the testing phase of a constraint's implementation would lead to the identification of shortcomings in the constraint's specification, and possibly to the creation of a new VMQL constraint. A total of 85 Prolog unit tests were used, and their execution was facilitated by the PlUnit[4] unit testing framework supported by SWI-Prolog.

[4] http://www.swi-prolog.org/pldoc/package/plunit.html

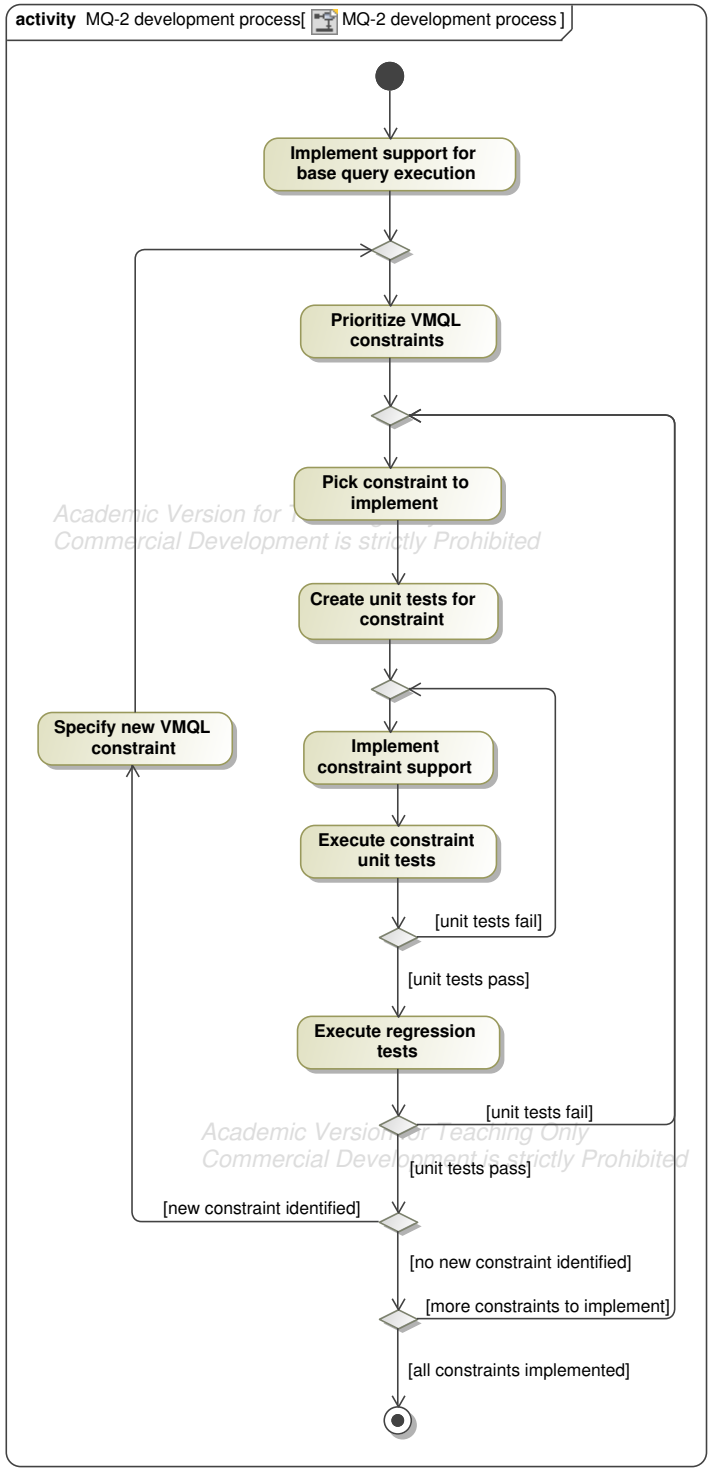


Figure 4.12: Activity Diagram illustrating the test-driven development process of the VMQL query execution algorithm

Chapter 5

User guide

5.1 Using the MQ-2 Prolog console

The Prolog console can be opened by selecting the *MQ-2* entry from the *Tools* menu in the MagicDraw menu bar. This action will cause the MQ-2 Prolog Console window to appear at the bottom of the MagicDraw application window (see Figure 5.1). The console may only be opened if a model is already open in MagicDraw. The console's initial behavior is that of a regular Prolog console: it allows executing queries supported by SWI-Prolog, and prints query results and error messages. The following keys must be used to execute queries in the MQ-2 Prolog console:

- `Return:` Prints the next query result. The behavior of the `Return` key may be modified using the Quick Actions drop-down on the Console Tool Bar (see Section 5.1.2).
- `Up:` Displays the previous query.
- `Down` Displays the next query.

The MQ-2 Prolog console offers a number of features aimed specifically at the task of model querying. These features are accessible through the Console Tool

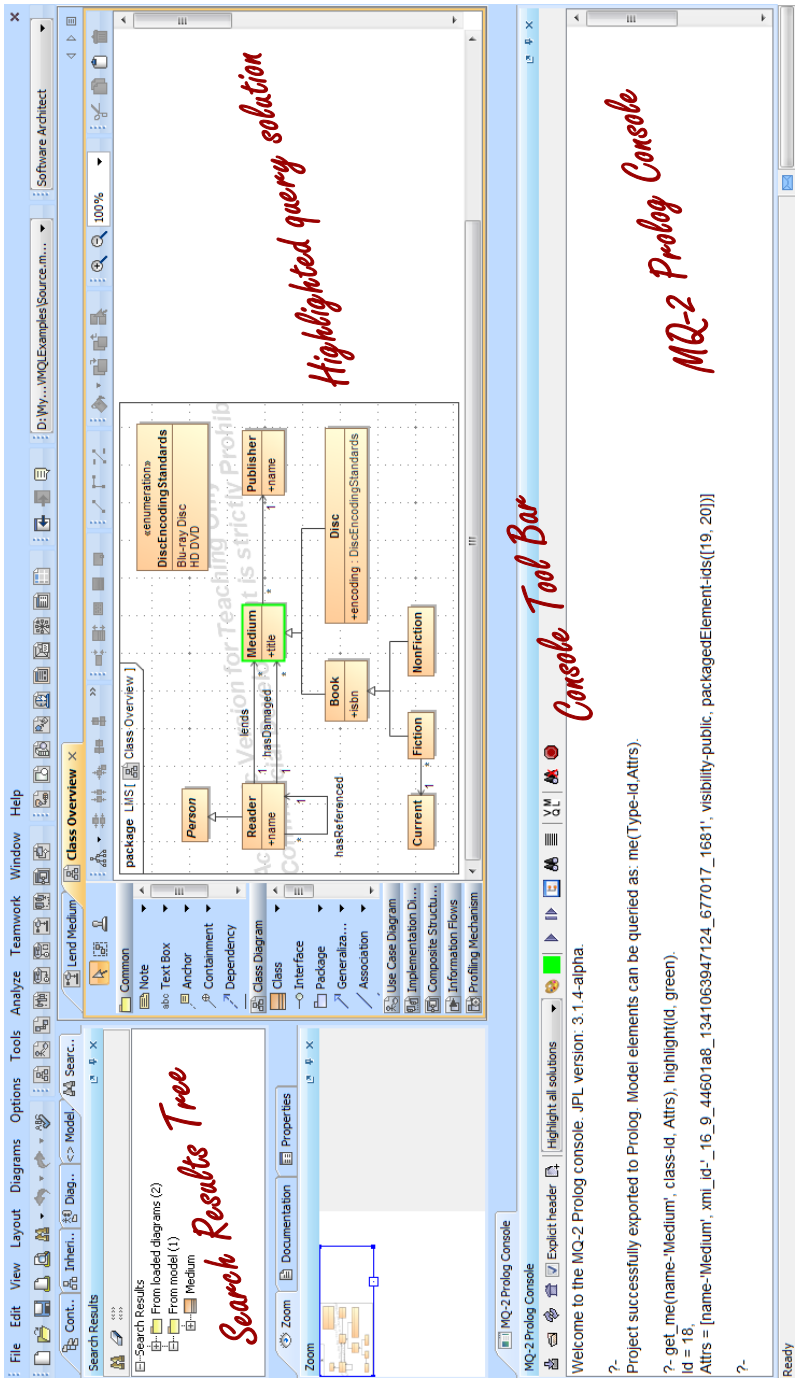


Figure 5.1: MagicDraw window featuring the MQ-2 Prolog console. Model elements returned by the last console query are highlighted in green.

Bar, placed above the console text area, and are detailed in what follows.

5.1.1 Consulting models

Before the model query facilities provided by the MQ-2 Prolog console can be used, a model must be consulted. The Console Tool Bar features the following buttons enabling model consulting:

- `Consult the currently active model:` Consults the model currently open in MagicDraw so that it is available for querying.
- `Select a MagicDraw model to consult`: Opens a file browser allowing users to select a MagicDraw project file to consult so that it is available for querying. Note that only projects stored in the MDXML file format may be selected.
- `Re-consult the MagicDraw model`: Re-consults the last consulted model, regardless if it is open in MagicDraw or not.
- `Re-start the Prolog console`: Clears the console contents and unconsults all previously consulted modules.

The `Explicit header` check-box allows users to select whether or not the `me/2` predicate should be included in the header of the Prolog module generated when a model is consulted through one of the methods above. In case it is checked, the `me/2` predicate will be included in the generated module header, and model elements may be accessed from the console using predicates of the form `me(Type,Id-Attributes)`. In case it is left un-checked, the `me/2` predicate will be omitted from the generated module header, and model elements may be accessed from the console using predicates of the form `'ModelName':me(Type,Id-Attributes)`, where 'ModelName' is the name of the consulted MagicDraw model. By default, the `Explicit header` check-box is un-checked.

Warning: *When using MQ-2 with versions of SWI-Prolog older than 6.1.9, consulting two Prolog modules that expose the same predicate in their headers causes MagicDraw to crash. For this reason, it is recommended to leave the* `Explicit header` *check-box un-checked.*

5.1.2 Querying models

Once a model has been consulted, it is possible to execute queries on it. The most direct way to query a consulted model is through the `me(Type-Id,Attrs)` predicate, where `Type` is the meta-type of a model element, `Id` is its unique generated identifier, and `Attrs` is a list of the element's meta-attributes and their values. For instance, a query retrieving a class named Reader has the following form:

```
me(Type-Id,Attrs), member(name-'Reader',Attrs).
```

Additional MQ-2 library predicates that can be used to query model elements are presented in Section 5.1.3. While all queries can be executed by simply typing them into the console and pressing the `Return` key, the Console Tool Bar features the Quick Actions drop-down that can be used to add custom behavior to the `Return` key. The Quick Actions drop-down contains the following entries:

- `Print next solution:` When this option is selected, pressing the `Return` key causes the next query solution to be printed on the console. If no other solutions exist, a new prompt is printed. This is the standard Prolog console behavior.
- `Print all solutions:` When this option is selected, pressing the `Return` key causes all query solutions to be printed, followed by a new prompt.
- `Show all solutions in tree:` When this option is selected, pressing the `Return` key causes all query solutions to be printed and all model elements included in the query solution to be shown in MagicDraw's Search Results Tree. A new prompt is printed on the console.
- `Highlight all solutions:` When this option is selected, pressing the `Return` key causes all query solutions to be printed and all model elements included in the query solution to be highlighted in the diagrams in which they appear. The highlight color may be selected using the `Select highlight color` button on the Console Tool Bar. A new prompt is printed on the console.
- `Show selection in tree:` When this option is selected, pressing the `Return` key causes all model elements which can be identified by the selected console text to be displayed in MagicDraw's Search Results Tree.

Alternatively, all actions included in the Quick Actions drop-down can be executed through corresponding buttons on the Console Tool Bar. The buttons do

not alter the behavior of the `Return` key, but rather replace its role. The Console Tool Bar contains two additional buttons addressing Prolog query execution: the `Clear highlights` button, which clears all highlights from all diagrams (including highlights generated by VMQL queries, as discussed in Section 5.2.2), and the `Abort query` button, which stops the execution of the current query.

5.1.3 Library predicates

Querying models using the integrated Prolog console is facilitated by the preconsulted MQ-2 library predicates:

- `get_me(Attr-Val,Type-Id,Attrs)`: Returns the attribute values of all model elements of type `Type` having the value `Val` for the attribute `Attr`. Example usage (finding the attributes of the class named 'Reader'):

  get_me(name-'Reader', class-Id, Attrs).

- `part_of(Kind,SuperId,SubId)`: Returns the ID of a model element representing a part of type `Kind` of the model element with ID `SuperID` in the variable `SubId`. Example usage (finding all owned ends of the model element with ID 1):

  part_of(ownedEnd, 1, SubId).

- `highlight(Elements,Color)`: Highlights the model elements identified by the `Elements` parameter in the specified `Color`. The `Elements` variable can either contain a list of `me/2` predicates, a list of model element IDs, or a list of model element names. Example usage (highlighting the class named 'Reader' in green):

  get_me(name-'Reader', class-Id, Attrs), highlight(Id, green).

In addition to the MQ-2 library predicates, users can consult their own custom defined library predicates at run time by pressing the `Consult user defined Prolog modules` button on the console tool bar. Files containing user-defined predicates must be placed in the `<MagicDraw home>/plugins/mq2/user/` directory prior to being consulted. Files containing helper predicates required by the user defined library predicates must be placed separately in the `<MagicDraw home>/plugins/mq2/user/helpers/` directory, so that they are not directly consulted in the MQ-2 Prolog console.

5.1.4 Limitations

The MQ-2 Prolog console does not support executing queries in debug mode. Calling the `debug/0` predicate must be avoided, as it will cause MagicDraw to crash. All errors that cause Prolog to enter debug mode will also cause MagicDraw to crash. This behavior occurs due to the fact that MagicDraw interprets the Prolog debug mode as a Java Virtual Machine crash.

5.2 Executing VMQL queries

Besides providing support for executing Prolog queries on models, the MQ-2 Prolog console also supports executing VMQL queries. The VMQL query execution interface, shown in Figure 5.2 on the bottom right corner of the MagicDraw main window, can be activated or de-activated from the `VMQL` toggle button on the Console Tool Bar.

5.2.1 Query execution

Executing a VMQL query requires a source model and a query model to be selected. The MQ-2 VMQL query execution interface assumes that the source model is the currently open MagicDraw model. Therefore, the first step in executing a VMQL query is consulting this model in the MQ-2 Prolog console, as described in Section 5.1.1. Selecting a query model and executing it as a VMQL query against the source model is facilitated by the following buttons on the VMQL query execution interface:

- `Select a MagicDraw project to be used as query model:` Opens a file browser allowing users to select a MagicDraw project file to be used as a VMQL query model. Note that only projects stored in the MDXML file format may be selected.
- `Re-consult the current query model:` Re-consults the last selected VMQL query model.
- `Execute the selected VMQL query:` Triggers the execution of the VMQL matching algorithm between the selected source and query models.

Query execution results are displayed in a tabular format bellow these buttons in the Bindings Table. Each binding is displayed as a row in the Bindings Table,

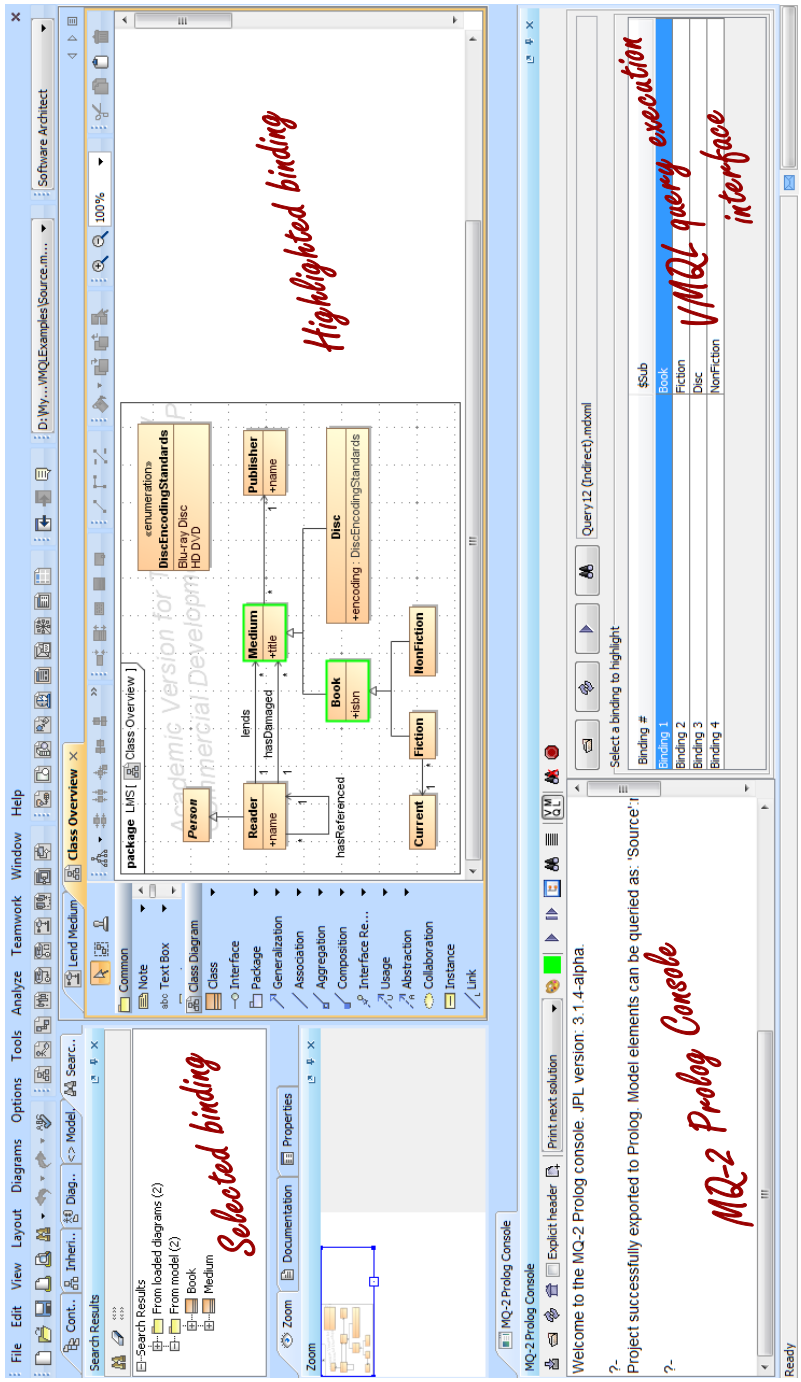


Figure 5.2: MagicDraw window featuring the MQ-2 Prolog console and VMQL query execution interface. The selected VMQL binding is highlighted on the model in green.

while the first column of each row identifies the index of the binding. In case the query model includes VMQL variables, subsequent columns correspond to the values taken by these variables in each binding.

5.2.2 Result highlighting

Selecting a binding from the Bindings Table by clicking on it leads to the source model elements included in this binding being displayed in the MagicDraw Search Results Tree. The selected binding can also be highlighted on the source model's diagrams through the `Highlight the selected binding` button. Just as in the case of highlighting Prolog query results, the highlight color can be selected via the `Select highlight color` button on the Console Tool Bar, and highlights can be cleared via the `Clear highlights` button on the Console Tool Bar.

Chapter 6

Evaluation

This chapter presents MQ-2's coverage of the VMQL specification, including extensions to this specification (Section 6.1), and offers an evaluation of the query execution algorithm's performance (Section 6.2).

6.1 VMQL implementation coverage

The set of VMQL constraints supported by MQ-2 is not identical to the original set of constraints specified in [Stö11b]. Some constraints' definitions have been extended, some constraints have been considered out of scope, and some have been added. Table 6.1 summarizes the implementation status of each constraint, while the following subsections detail the modifications and additions brought by MQ-2 to the VMQL specification.

6.1.1 The `once` constraint

The first modification to the original VMQL specification regards the ambiguous definition provided in [Stö11b] for the `once` constraint, which is defined as "enforcing that a solution occurs only once in the set of all solutions". This

Table 6.1: Implementation status of VMQL constraints

Constraint	Status
`mattr`	covered
`name`	covered
`match`	covered
`mclass`	covered
`once`	extended (see Section 6.1.1)
`distinct`	covered
`optional`	covered
`either`	covered
`not`	covered
`steps`	covered
`indirect`	covered
`precision`	out of scope
`strict`	out of scope
`optional+`	added (see Section 6.1.2)
`either+`	added (see Section 6.1.3)
`not+`	added (see Section 6.1.4)

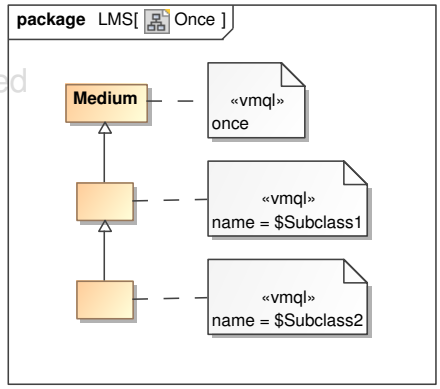
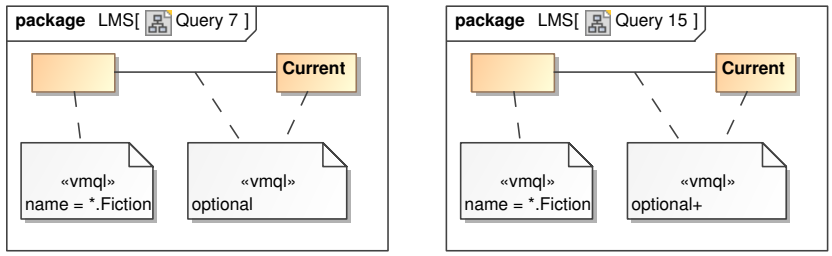


Figure 6.1: A query that may produce no bindings when applied to the LMS source model in Section 2 due to the non-determinism of the `once` constraint

definition does not specify which solution should be retained in case the query model element to which the constraint is applied is bound to the same source model element in several solutions. The compromise adopted for the MQ-2 implementation is to retain the first solution generated by the Prolog matching algorithm. Users should thus be aware that using the `once` constraint can lead to otherwise valuable query solutions being omitted. Such a situation can be exemplified by considering the LMS source model introduced in Section 2, together with the query model in Figure 6.1. Due to the presence of the `once` constraint, the query may actually produce no solutions in case the matching algorithm binds the top-most subclass in the query model (the one annotated with the `name = $Subclass1` constraint) to the `Disc` subclass first. This would become the only considered solution, and would later also be discarded due to the lack of a binding for the second subclass in the query model (the `Disc` source class has no subclasses). The second and third bindings produced by the matching algorithm, which would correctly match the query subclass pair with the `Book` and `Fiction` classes and with the `Book` and `NonFiction` classes, respectively, would be discarded due to the `once` constraint, leaving the query to produce no results.

6.1.2 The `optional+` constraint



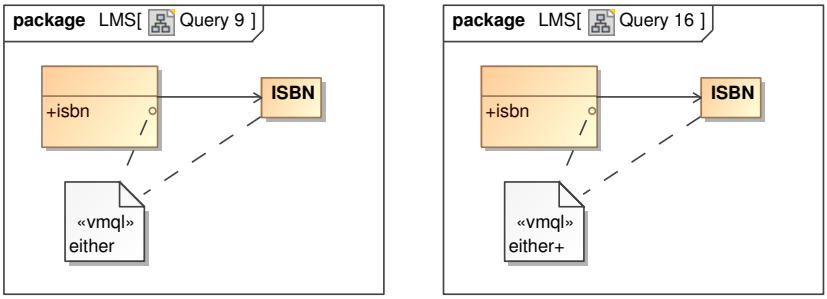
(a) (b)

Figure 6.2: Queries illustrating the `optional` constraint ((a)) and the `optional+` constraint ((b))

A second limitation has been discovered in what concerns the `optional` constraint. Consider Query 15 presented in Figure 6.2(b), which is similar to Query 7 presented in Figure 6.2(a), with the only difference being that it replaces the `optional` constraint with the new `optional+` constraint. The `optional+` constraint is not part of VMQL's original specification and has been introduced as a result of observations made during MQ-2's development. Query 7 should return

the `Fiction` and `NonFiction` classes. Applying the `optional` constraint to the `Current` class and to the association in the query model should intuitively have the desired effect of making their presence in a solution optional, so that the `NonFiction` class can also be found. However, due to the fact that in UML an association references a property of a class rather than the class itself, Query 7 also introduces a new property to the leftmost query class (the one annotated with the `name = *.Fiction` constraint). Consequently, the leftmost class in the query model cannot be matched with the `NonFiction` class of the source model. The solution to this issue is to also make the properties introduced by an association optional in case an `optional` constraint is anchored to the association. The same solution can be applied to any other relationship type. The `optional+` constraint does precisely this and enables Query 15 to find the `NonFiction` class in the source model.

6.1.3 The `either+` constraint



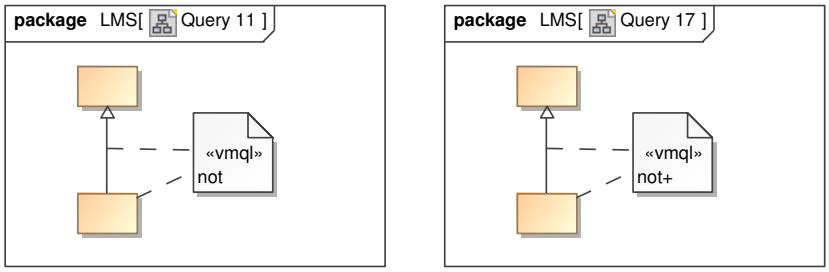
(a) (b)

Figure 6.3: Queries illustrating the `either` constraint ((a)) and the `either+` constraint ((b))

A similar limitation affects the `either` constraint. Consider the case illustrated by Query 16 in Figure 6.3(b), which is similar to Query 9 presented in Figure 6.3(a). This query aims to find classes that either have an `isbn` property or are associated to a class named `ISBN` in the LMS source model. If the `either` constraint is applied, as shown in Query 9, the matching algorithm fails to retrieve the `Book` class from the source model, since the `Book` class does not have any properties connected to an association - and, according to Query 9, the presence of such a property is mandatory. Though justified, this observed behavior contradicts the one expected according to the VMQL specification. Following a logic similar to that behind the introduction of the `optional+` con-

straint, the `either+` constraint is used in Query 16: this constraint implicitly makes the properties connected to the association in Query 16 optional. The query will thus successfully retrieve the `Book` class from the source model. The `either+` constraint was also introduced during the development of MQ-2 and is not featured in the VMQL specification.

6.1.4 The `not+` constraint



(a) (b)

Figure 6.4: Queries illustrating the `not` constraint ((a)) and the `not+` constraint ((b))

Finally, a limitation of this kind also affects the `not` constraint. This situation is exemplified by Query 17 in Figure 6.4(b), which is based on Query 11 in Figure 6.4(a). Query 11 searches for all classes that do not have subclasses, but actually fails due to the fact that the generalization relationship adds a `packagedElement` property to the superclass in the query model. No source model class that does not have subclasses will exhibit a property of this type, leading Query 9 to produce no bindings. This is once more an implementation level detail that would have been difficult to foresee at the time of creating the original VMQL specification. The new `not+` constraint is required instead, as shown in Query 17. It implicitly makes optional the `packagedElement` attribute introduced to the superclass in the query model by the generalization relationship. Query 17 thus succeeds in finding the `Reader`, `Publisher`, `Disc`, `Current`, `Fiction`, and `NonFiction` classes of the source model. Just as the `optional+` and `either+` constraints, the `not+` constraints has the additional effect of making all model elements referenced by the constrained relationship optional, regardless of the type of this relationship.

6.2 Performance evaluation

In order to evaluate the performance of the VMQL matching algorithm, a series of 60 queries covering all VMQL features are executed against several source models. They are executed on the following source models:

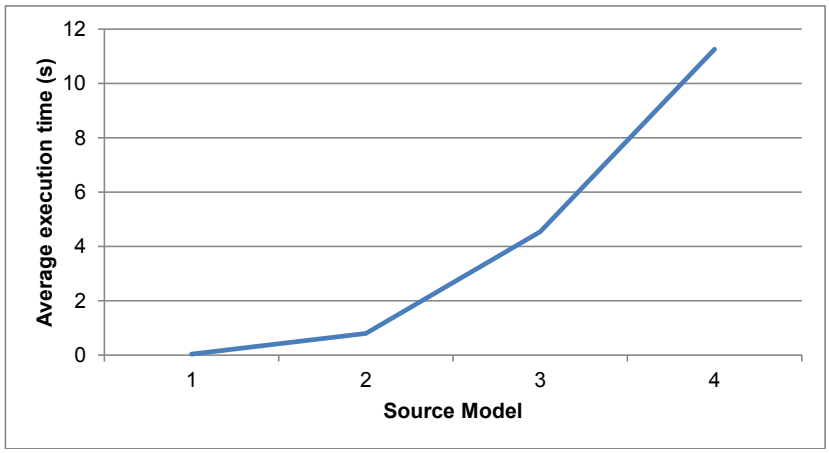


Figure 6.5: Average query execution time for the considered source models. Based on a set of 60 queries covering all VMQL features.

- **Source model 1:** The source model used in Section 2 to illustrate the constraints supported by VMQL. This model consists of 94 model elements.
- **Source model 2:** A subset of a Library Management System analysis model developed by students of a Requirements Engineering course at DTU. This model consists of 316 model elements.
- **Source models 3 and 4:** Two full Library Management System analysis models developed by students of a Requirements Engineering course at DTU. These models consists of 827 and 1774 model elements, respectively.

The average execution time for one query observed in these four scenarios is presented in Figure 6.5. While the increase in query execution time relative to source model size is considerable, it appears to follow a polynomial curve rather than an exponential one. Nevertheless, the query execution times of the order of seconds observed for the large models are sufficient to become a nuisance for

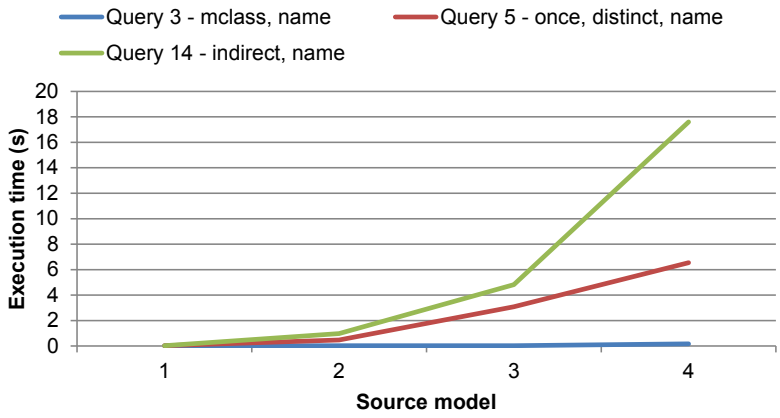


Figure 6.6: Query execution times for three queries of different complexity on the considered source models.

modelers. This suggests that the matching algorithm would benefit from further performance optimizations.

Another aspect of interest is the effect that query models have on query execution time. To illustrate this, three of the queries presented in Section 2 were executed on the same four source models. The following queries were considered:

- **Query 3 (Figure 2.5):** Illustrates the `mclass` and `name` constraints, both applied to modify the query model before match computation in Algorithm 4.1.
- **Query 5 (Figure 2.7):** Illustrates the `once` and `distinct` constraints, both applied to filter the results of match computation in Algorithm 4.1. The query also contains a `name` constraint.
- **Query 14 (Figure 2.12):** Illustrates the `indirect` constraint, allowing transitive closure computation. The query also contains a `name` constraint.

The execution times of these queries on the four source models are presented in Figure 6.6. Query 3 is executed almost instantly for all model sizes, suggesting that match candidates are filtered early on in the match process based on the query model element attributes. Query 5's execution time appears to increase on a polynomial scale with source model size. This is likely motivated by the

fact that the `once` and `distinct` constraints are taken into account towards the end of the matching process, leaving many bindings to be processed up to that point. Finally, Query 14's execution time increases at an exponential rate with respect to source model size. This is also not entirely surprising: path constraints are the most computationally demanding of all VMQL constraints.

Chapter 7

Conclusions

The objectives set for this project and stated in Section 1.1 have been met. The visual model query language has been successfully implemented as part of the MQ-2 plug-in for the MagicDraw CASE tool. The plug-in relies on an underlying Prolog algorithm for query execution. The results produced by this algorithm have been verified using an extensive suite of test cases, and the algorithm's performance has been evaluated against a set of models of different sizes. The plug-in also features a fully functional Prolog console, equipped with the means to query the Prolog representations of MagicDraw models.

During the development of MQ-2, a number of shortcomings in the original VMQL specification have been identified. Solutions to these shortcomings have been proposed and implemented. A second category of difficulties encountered during this implementation have been related to the reliability of the third party library employed for the task of connecting the Java-based front-end of MQ-2 with the Prolog-based query execution back-end. A significant effort has been devoted to ensuring MQ-2's ability to operate across various software platforms, including both 32-bit and 64-bit operating systems. These efforts have been only partially successful due to the lack of stability of the current version of the mentioned third party library.

The work showcased by this thesis has been presented in the tools section of the 8^{th} European Conference on Modelling Foundations and Applications [AS12].

The feedback received from modeling community members at this venue has been positive, with several participants declaring an interest in using the tool.

7.1 Future work

There are two possible main directions of further development concerning the MQ-2 implementation: extensions concerning exclusively the tool and extensions carried out in tandem with an evolution of VMQL.

The first category includes adapting MQ-2 to support modeling languages other than UML. Immediate candidates would be other MOF-based languages, as well as business process modeling languages such as BPMN. Due to the fact that the current implementation relies to a very little extent on knowledge of the UML meta-model, such adaptations are expected to be relatively undemanding in terms of development effort. A second category of refinements concerning exclusively the MQ-2 implementation are related to the tool's usability. While efforts have been made to maintain a lightweight and intuitive user interface, the ultimate confirmation of these efforts' success should come in the form of a proper usability evaluation, including empirical user studies.

The second direction for future work must take advantage of the considerable extension potential offered by VMQL. A first step in this direction has already been taken with the proposal of a set of additional VMQL constraints supporting the expression of model constraints [Stö11a]. These additional constraints are not currently supported by MQ-2. However, considering the fact that the query execution algorithm lying at the core of MQ-2 has been designed to be easily extended, the new constraints can be implemented without incurring any changes to the processing of existing constraints. A second VMQL extension that could constitute an appropriate match for this implementation concerns the problem of model version control. It is possible to envision MQ-2's existing query solution highlighting options being used to display the differences between two versions of a model. Finally, a more consistent extension to VMQL that would enable its usage as a model transformation language may also have MQ-2 as a basis for its implementation.

Appendix A

Installation instructions

System requirements:

- Operating system: Windows 7 (x86 or 64-bit), Linux
- Java SE 6 or higher
- MagicDraw 16.9
- SWI-Prolog 6.0.2 or higher (32-bit OS), SWI-Prolog 6.1.9 or higher (64-bit OS)

Note 1: *64-bit versions of Java SE, MagicDraw and SWI-Prolog must be installed on 64-bit machines.*

Note 2: *SWI-Prolog 6.1.9 and higher currently require Java SE 7 on 32-bit machines.*

Installation steps:

1. Obtain a copy of the `mq2.zip` archive and extract its contents (a single directory named `mq2`) to a location of your choice.

2. Determine the installation directory of MagicDraw on your machine. Refer to this directory as `%MD_Home%` in what follows.

3. Determine the installation directory of SWI-Prolog on your machine. Refer to this directory as `%SWI_Home%` in what follows.

4. Copy the extracted `mq2` directory to the MagicDraw plug-ins folder, located at `%MD_Home%/plugins`.

5. Replace the version of the `jpl.jar` archive distributed with MQ-2 (found at `%MD_Home%/plugins/mq2/lib/jpl.jar`) with the version included in your local SWI-Prolog installation (found at `%SWI_Home%/lib/jpl.jar`). Linux users are encouraged to compile a version of `jpl.jar` specific to their distribution[1].

6. Create the `SWI_HOME_DIR` environment variable with `%SWI_Home%` as value.

7. Add `SWI_HOME_DIR/bin` to the local path environment variable.

8. Re-start your system.

[1] For instructions on compiling `jpl.jar`, see *https://code.google.com/p/javanaproche/wiki/HowToJPL* (an external tutorial not related to MQ-2)

Bibliography

[AB01] D. H. Akehurst and Behzad Bordbar. On Querying UML Data Models with OCL. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML '01, pages 91–103, London, UK, 2001. Springer-Verlag.

[AHM05] Vasco Amaral, Sven Helmer, and Guido Moerkotte. Formally Specifying the Syntax and Semantics of a Visual Query Language for the Domain of High Energy Physics Data Analysis. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '05, pages 251–258, 2005.

[AS12] Vlad Acretoaie and Harald Störrle. MQ-2: A Tool for Prolog-based Model Querying. In *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*, pages 328–331, Kongens Lyngby, DK, 2012. Technical University of Denmark.

[AWW11] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Visually specifying compliance rules and explaining their violations for business processes. *Journal of Visual Languages and Computing*, 22:30–55, February 2011.

[BEKM08] Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying business processes with BP-QL. *Information Systems*, 33:477–507, September 2008.

[BKPPT01] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A Visualization of OCL Using Collaborations.

In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML '01, pages 257–271, London, UK, 2001. Springer-Verlag.

[BPM10] Business Process Model and Notation, Object Management Group (OMG) Standard, Version 2.0, 2010.

[COL08] Joanna Chimiak-Opoka and Christian Lange. Querying UML Models using OCL and Prolog: A Performance Study. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, ICSTW '08, pages 81–88, 2008.

[Dat99] C.J. Date. *An introduction to database systems.* Addison Wesley Publishing Company, August 1999.

[DP05] Dolev Dotan and Ron Y. Pinter. HyperFlow: an Integrated Visual Query and Dataflow Language for End-User Information Analysis. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '05, pages 27–36, 2005.

[GT08] Anca Ghitescu and Evaldas Taroza. *ActiveXML documentation*, JUL 2008.

[Hal85] Joseph Y Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30:1–24, 1985.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[KBC04] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language mola. In *Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004)*, pages 14–28, 2004.

[Ken97] Stuart Kent. Constraint diagrams: visualizing invariants in object-oriented models. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 327–341, New York, NY, USA, 1997. ACM.

[MC99] Luis Mandel and Maria Victoria Cengarle. On the expressive power of ocl. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*, pages 854–874. Springer, September 1999.

[MOF06] Meta Object Facility (MOF) Core Specification, Object Management Group (OMG) Standard, Version 2.0, 2006.

[OCL11] Object Constraint Language, Object Management Group (OMG) Standard, Version 2.3.1, 2011.

[Ope10] *MagicDraw Open API user guide version 16.9*, 2010.

[QVT03] Revised submission for MOF 2.0 Query/Views/Transformations RFP, Technical Report, Version 1.1, August 2003.

[QVT11] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Object Management Group (OMG) Standard, Version 1.1, 2011.

[SHU04] Dominic Stein, Stefan Hanenberg, and Reiner Unland. Query Models. *Proceedings of the Seventh International Conference on Unified Modeling Language (UML'04), Lecture Notes in Computer Science*, 3273:98–112, 2004.

[SHU05] Dominic Stein, Stefan Hanenberg, and Reiner Unland. On Relationships Between Query Models. *Proceedings of the European Conference on Model Driven Architecture — Foundations and Applications (ECMDA-FA2005), Lecture Notes in Computer Science*, 3748:254–268, 2005.

[Stö07] Harald Störrle. A PROLOG-based Approach to Representing and Querying Software Engineering Models. In *International Workshop on Visual Languages and Logic*, volume 274 of *VLL '07*, pages 71–83. CEUR, 2007.

[Stö09] Harald Störrle. A logical model query interface. In *International Workshop on Visual Languages and Logic*, volume 510 of *VLL '09*, pages 18–36. CEUR, 2009.

[Stö10] Harald Störrle. Structuring very large domain models: experiences from industrial MDSD projects. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 49–54, New York, NY, USA, 2010. ACM.

[Stö11a] Harald Störrle. Expressing Model Constraints Visually with VMQL. In *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '11, pages 195–202. IEEE, 2011.

[Stö11b] Harald Störrle. VMQL: A visual language for ad-hoc model querying. *Journal of Visual Languages and Computing*, 22:3–29, February 2011.

[UML11] OMG Unified Modeling Language (OMG UML), Infrastructure and Superstructure, Object Management Group (OMG) Standard, Version 2.4.1, 2011.

[Win09] M. Winder. MQ—Eine visuelle Query-Schnittstelle für Modelle, Bachelor's Thesis, Innsbruck University, 2009.

[WSB07] Web Services Business Process Execution Language, Organization for the Advancement of Structured Information Standards (OASIS) Standard, Version 2.0, 2007.

[XMI11] OMG MOF 2 XMI Mapping Specification, Object Management Group (OMG) Standard, Version 2.4.1, 2011.

[XPa10] XML Path Language (XPath) 2.0 (Second Edition), World Wide Web Consortium (W3C) Recommendation, 2010.