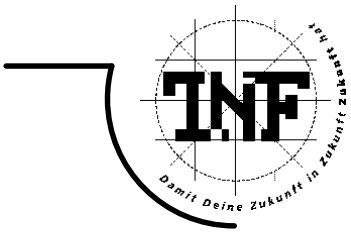
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



A Framework for Publishing and Discovering Mathematical Web Services

Dissertation

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

Angefertigt am *Institut für Symbolisches Rechnen*

Betreuung:

A.Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Schreiner

Beurteilung:

A.Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Schreiner
Univ.-Prof. Dr. Roland Wagner

Eingereicht von:

M.Sc. Eng. Rebhi S. Baraka

Linz, August 2006

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Rebhi S. Baraka
Linz, August 2006

”... رَبِّ أَوْزِعْنِيٓ أَنْ أَشْكُرَ نِعْمَتَكَ ٱلَّتِيٓ أَنْعَمْتَ عَلَيَّ وَعَلَىٰ وَٰلِدَيَّ وَأَنْ أَعْمَلَ صَٰلِحًا تَرْضَىٰهُ وَأَدْخِلْنِي بِرَحْمَتِكَ فِي عِبَادِكَ ٱلصَّٰلِحِينَ“ (النمل 19)

“... My Lord! Inspire me and bestow upon me the power and ability that I may be grateful for Your Favors which You have bestowed on me and on my parents, and that I may do righteous good deeds that will please You, and admit me by Your Mercy among Your righteous slaves.” (The Holy Qur'an: An-Naml 19)

“... Mein Herr, gib mir ein, dankbar für die Gnade zu sein, die Du mir und meinen Eltern gewährt hast, und (gib mir ein,) Gutes zu tun, das Dir wohlgefällig sei, und nimm mich in Deiner Barmherzigkeit unter Deine rechtschaffenen Diener auf.” (Der Heilige Koran: An-Naml 19)

This work is dedicated to the memory of my mother

Zusammenfassung

Die Veröffentlichung und das Auffinden von Web-Diensten ist eine der entscheidenden Fragen, die von gegenwärtigen Web-Architekturspezifikationen angesprochen werden. Im Falle mathematischer Web-Dienste ist dieser Punkt noch etwas subtiler, da solche Dienste in semantisch reichen Bereichen auf Objekten arbeiten, die eine entsprechende Codierung und Spezifikation benötigen.

Diese Dissertation entwickelt einen Software-Rahmen für die Veröffentlichung und das Auffinden mathematischer Web-Dienste, bestehend aus einem Register für die Veröffentlichung dieser Dienste und einer Abfragesprache für ihr Auffinden nach syntaktischen und semantischen Kriterien.

Die Basis dieses Rahmens ist die Sprache MSDL (Mathematical Service Description Language) für die Beschreibung verschiedener Aspekte eines mathematischen Dienstes. Diese Beschreibung enthält Informationen zum Typ des von dem Service gelösten Problems, des eingesetzten Verfahrens, der Implementierung des Verfahrens, der Computer, auf denen diese Implementierung läuft, und verwandter Probleme.

Darauf aufbauend wird eine Sprache für die Suche nach in dem Register publizierten mathematischen Diensten entworfen, formalisiert und implementiert. Diese "Mathematical Services Query Language" MSQL erlaubt die Suche nach solchen Diensten aufgrund ihrer MSDL-Beschreibungen. Die Implementierung von MSQL basiert auf einer formalen Definition der Sprache in denotationaler Semantik, was eine Referenz für die Korrektheit der Realisierung darstellt.

MSQL unterstützt Abfragen auf der syntaktischen Struktur von Dienstbeschreibungen, wurde aber dahin gehend erweitert, auch den semantischen Inhalt dieser Beschreibungen zu berücksichtigen. Dazu wird die Sprache um Konstrukte zur Formulierung prädikatenlogischer Formeln erweitert, die mit der Hilfe eines Softwaresystems zum automatischen Schließen bearbeitet werden.

Der Software-Rahmen wird experimentell evaluiert und mit einigen verwandten Ansätzen zum Abfragen von XML und zum Auffinden mathematischer Dienste verglichen.

Abstract

Web service publishing and discovery is one of the crucial issues addressed by contemporary Web service architectural specifications. In the case of mathematical Web services this issue is more subtle due to the fact that they operate within semantically rich domains on objects that need proper encoding and specification.

This dissertation develops a framework for publishing and discovering mathematical Web services consisting of a registry for publishing mathematical services and a query language for syntactically and semantically discovering mathematical services published in the registry.

The basis of the framework is a mathematical services infrastructure and a language (Mathematical Services Description Language) that describes the different aspects of a mathematical service. The description of a mathematical service in MSDL may contain information related to the type of the problem, the algorithm(s) used to solve the problem, the implementation of the algorithm, machines executing the implementation, and related problems.

On this basis, a language for querying registry published mathematical services is designed, formalized and implemented. The Mathematical Services Query Language (MSQL) provides the functionality to query services based on their MSDL descriptions. The implementation of MSQL is based on a formal definition of the language using denotational semantics. This definition provides a reference for the correctness of the language implementation.

MSQL supports queries on the syntactic structures of service descriptions but has been extended to address also the semantic information contained in service descriptions. This extension adds a number of constructs to the language in order to express predicate logic formulas that are processed with the help of an automated reasoner.

The framework is evaluated experimentally and is compared to some related approaches in XML querying and mathematical service discovery.

Acknowledgements

I would like to thank Wolfgang Schreiner, my dissertation supervisor, for his continuous guidance throughout my time at the Research Institute for Symbolic Computation (RISC). His creative ideas and feedback have shaped the results of this work.

Studying and doing research at RISC was a fruitful experience. I would like to acknowledge its rule as a stimulating research environment with dedicated and hardworking people.

Special thanks goes to my wife Asmaa and my children Alaa, Ahmad, Abdulla, Amira, and Affnan for their time and support.

Work on this dissertation was supported by RISC and by the Austrian Science Funds (FWF) under the projects P15183 and P17643-NO4.

Software Credits

I would like to give my credit to the people behind the following software.

- ebXMLrr Reference Implementation. The Registry implementation in Chapter 4 is an extension of the ebXMLrr; its management functionality is an extension to that of the ebXMLrr. Also its information model is an extension to that of the ebXMLrr.
- The RISC ProofNavigator. MSQL uses a component of it as the interface to the reasoner.
- CVC Lite (the cooperating Validity Checker Lite). MSQL uses it as the automated reasoner
- The OpenMath Library. Used in MSDL and MSQL to represent and encode mathematical content.
- ANTLR (ANother Tool for Language Recognition). The MSQL grammar is written in ANTLR to generate the Lexer and Parser of MSQL.
- DOM (the Document Object Model). Used to model MSDL documents when queried by MSQL.
- Other software tools include: Apache TOMCAT servlet, Apache Ant build tool, Apache Axis, Sun JWSDP (Java Web Services Developer Pack) including its library of tools, and the eclipse IDE.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The interest of the mathematical community in using the Internet and the Web to facilitate the use of mathematics has paved the way for the emergence of mathematical Web services. A mathematical Web service is defined in line with the definition of a Web service as a solution to a mathematical problem that is available on the Web and can be accessed by a user or another service or program.

Web service publishing and discovery is one of the crucial issues addressed by contemporary Web service architectural specifications. Service publishing is any operation that makes a service description available for a user. Such an operation is the submission of a service description to a registry. Any operation allowing access to the description of a Web service can be viewed as a service discovery. Such an operation is a query for a registry-published service description. In the case of mathematical Web services the issue of service publishing and discovery is more subtle than that of, e.g., e-Business due to the fact that they operate within semantically rich domains on objects that need proper encoding and description.

1.1 Framework Overview

In this dissertation, we develop a framework for publishing and discovering mathematical Web services.

The basis of the framework is an infrastructure for building mathematical services and a language for describing such services developed in the frame of the MathBroker project [73]. The infrastructure allows building and deploying mathematical Web services that use computational systems such as computer algebra engines and reasoner systems such as automated provers over the Web. The description language, the Mathematical Services Description Language (MSDL), is based on an information model that describes the different aspect of a mathematical service. The description of a mathematical service may contain information related to the type of

the problem, the algorithm(s) used to solve the problem, the implementation of the algorithm, machines executing the implementation, and related problems.

On this basis, the developed framework is composed of:

- A *registry framework* for publishing and discovering mathematical Web services based on the ebXML registry standard [45] and its reference implementation [44]. We extend the ebXML registry information model to include the MSDL information model and extend the ebXML *Registry Service* management functionality to include mathematical object management functionality. We implemented these extensions on top of the ebXML Registry Reference implementation (ebXMLrr). A service is published in the registry by submitting its MSDL description. The registry management functionality extracts from the description the relevant information (metadata) needed to classify the service according to a predefined classification schemes in the registry, associating it or its constituting entities to other services or entities, and saving its submitted description in the repository of the registry. The registry provides the facilities to browse or query the stored services according, e.g., to their names, unique identifiers (IDs), types, or classifications. We call this kind of querying metadata-based querying. This kind of querying is not sufficient in the case of mathematical services which are based on semantically rich content. Therefore, we also develop a content-based service discovery framework.

- A *discovery framework* that includes the design, formalization, and implementation of the Mathematical Services Query Language (MSQL) for querying the contents of registry-published mathematical services. MSQL has a minimal set of constructs needed to interface to the registry in order to retrieve candidate service descriptions, to compose query expressions that address the different information in these descriptions, to filter these descriptions based on the composed query expressions, and to return the filtered descriptions as the results of the query.

  We base the formal definition of MSQL on denotational semantics, a technique for rigorously specifying the semantics of a programming language. This gives a clear meaning to MSQL constructs and therefore provides a correct reference for the implementation. The core component of the implementation is the MSQL engine which receives queries, performs them, and returns MSDL documents as results. This functionality of MSQL is exposed through the MSQL API.

- A *semantic-based extension* of the discovery framework. MSQL supports queries on the syntactical structure of service descriptions. To

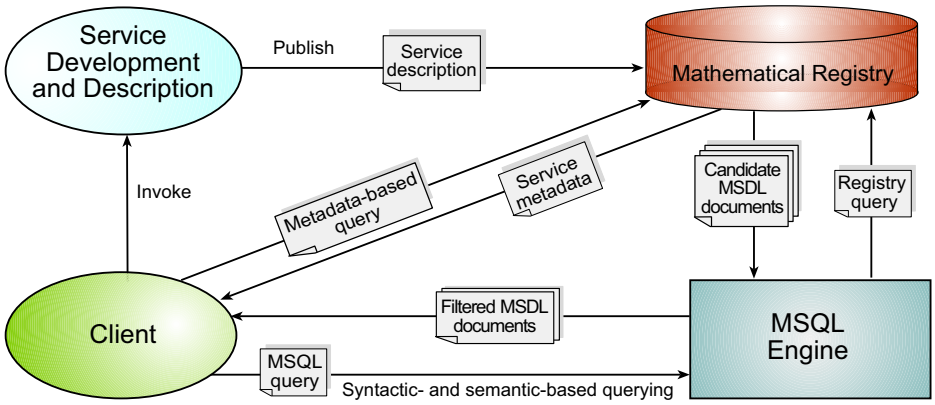


Figure 1.1: An Abstract View of the Framework

exploit the semantical structures underlying these descriptions, we extend MSQL to include semantic-based queries that would require true reasoning. The semantical structures are described in OpenMath [27], a standard for representing mathematical content in a semantic-preserving way. The extension adds a number of constructs to MSQL so that predicate logic formulae can be formed on these semantical structures. The query engine of MSQL performs semantic-based queries with the help of an automated reasoner which takes predicate logic formulae, decides their validity, and returns the answer to the engine.

An abstract view of the envisioned framework is shown in Figure 1.1. A more detailed view of the framework which represents the whole architecture to be introduced in the rest of the dissertation is show in Figure 1.2. It consists of the following components:

- *Service Development and Description* represents the infrastructure for mathematical services. This is where a service is developed, described, deployed to a Web server, and its description is published in the mathematical registry.
- *Mathematical Registry* represents the registry framework for publishing and metadata-based querying. It provides the functionality needed for publishing service descriptions in the registry such that they can be discovered.
- *MSQL Engine* represents the two parts of the content-based discovery framework; the syntactic discovery and the semantic discovery.
- *Client* represents a service requester (client application) that either uses the registry directly where metadata-based querying can be performed or uses the MSQL engine where content-based syntactic and

semantic querying can be performed. Based on the results of the query, the client can invoke the service at its address provided in the query resulting description.

A possible interaction scenario among these components is as follows:

- The service developer deploys the service in the Web server and publishes its description into the registry.
- The client may query the registry directly for a service based on known metadata such as its name or classification in the registry. Alternatively, if the request cant be expressed as a metadata-based query, the Client forms content-based syntactic or/and semantic queries in the syntax of MSQL. Then the client sends this query to the MSQL engine.
- The MSQL Engine analysis the query, contacts the registry to retrieve a candidate set of descriptions, filters them and returns some of them to the Client as a result to the query.
- The Client (application) based on the returned description may invoke the service from its location.

1.2 Originality and Contributions

The results of this dissertation contribute to the synergy between computer mathematics and the Web. They demonstrates how current Web standards and technologies can be used to promote mathematical software and its integration into the Web. Current Web software and standards are e-Business oriented, using them in a field such as computer mathematics shows their limitations and thus the need for further improvements and generalizations.

In this regard, few projects have addressed the issue of mathematical Web services and their publication and discovery (see Section 2.5). Among these, the MONET project [87] has developed an architecture where ontologies are used to model service descriptions as well as queries on these descriptions. These ontologies are ontological conversions of MSDL descriptions written in OWL [99]. The ontology-based discovery process in MONET is based on a restricted form of first order logic which is more tractable for automated reasoning but strictly less expressive. In our semantic queries (see Chapter 6), we use full predicate logic which is a highly expressive language. In another part of MONET, a matching-based discovery approach [93] to registry-published mathematical services performs matchmaking between representations of tasks (client requests) and capabilities (service descriptions). This matching process is based on the syntactic similarity traced

between tasks and capabilities. In our case, the decision is based on logical implications between statements extracted from descriptions, which is strictly more general.

In summary, the main contributions of this dissertation are:

- A registry for publishing and discovering mathematical Web services.
- A content-based query language for discovering registry-published services.
- An extension to the query language supporting semantic-based discovery of registry-published services.

These contributions have been documented and published through three RISC technical reports [15, 10, 12], two software packages (the Mathematical Registry API [11] and the MSQL API [13]), and three papers at the following refereed international conferences:

- The IEEE International Conference on e-Technology, e-Commerce, and e-Service (EEE-05) [14].
- The IEEE 20th International Conference on Advanced Information Networking and Applications (AINA 2006) [16].
- The Third International Workshop on Web Services and Formal Methods (WS-FM 2006) [17].

The registry framework (Chapter 4) has been pursued in the frame of the MathBroker project[1]. The discovery framework (Chapters 5, 6) has been pursued in the frame of the MathBroker II project[2]. These projects have been sponsored by the Austrian Science Funds (FWF) under the contracts P15183 and P17643-NO4.

1.3 Organization of the Dissertation

The dissertation is organized as follows:

Chapter 2 (State of the Art) presents related approaches, technologies, and standards that have played remarkable roles in integrating mathematics into the Web. It highlights some early attempts to use the Internet to communicate mathematics and some current Web technologies and standards related to Web services and their semantics, publishing and discovery. It also presents some approaches that have used these Web technologies and standards to develop mathematical Web services.

[1] http://poseidon.risc.uni-linz.ac.at:8080/mathbroker/index.html
[2] http://www.risc.uni-linz.ac.at/projects/mathbroker2/

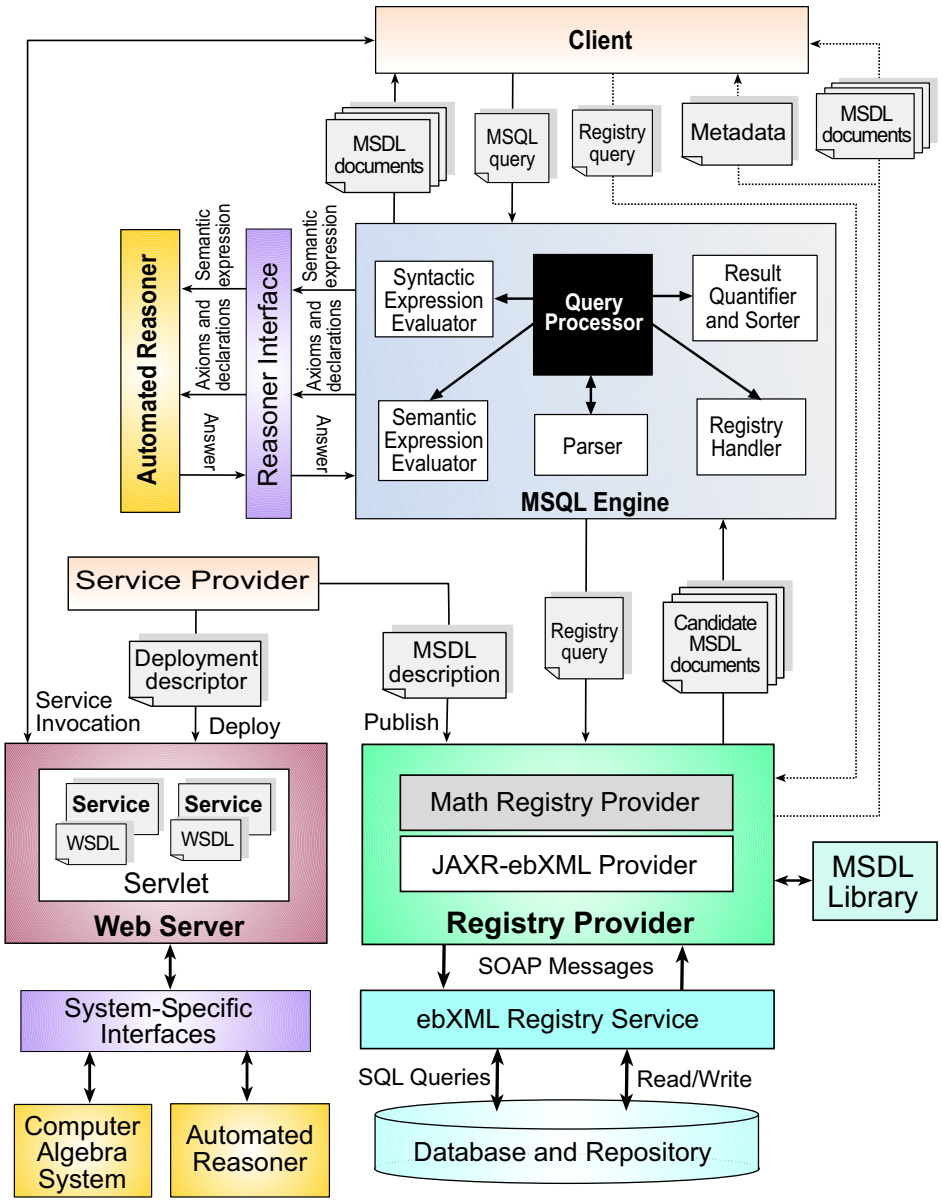


Figure 1.2: The Mathematical Services Framework Architecture

Chapter 3 (Service Development and Description) presents the basis of the framework (see Figure 1.2). The mathematical services infrastructure and the Mathematical Services Description Language (MSDL) developed in the frame of the MathBroker project [73].

Chapter 4 (Service Publishing: The Registry Framework) presents the framework for publishing and metadata-based discovery of mathematical services (see Figure 1.2). It describes the architecture, the information model, and the reference implementation of the ebXML registry standard on which the mathematical registry is based. Then it presents the development of the registry extension including the MSDL information model in context of the ebXML registry information model, the architecture, and the prototype implementation. It also gives some examples demonstrating the use of the resulting registry for publishing and discovering MSDL service descriptions.

Chapter 5 (Service Discovery I: The Mathematical Services Query Language Framework) presents the content-based discovery framework (see Figure 1.2). It presents the design, the formalization, and the implementation of the Mathematical Services Query Language (MSQL) used for the syntactic querying of MSDL descriptions stored in the registry. It presents the architecture and the implementation of the MSQL engine with example queries demonstrating the use of resulting MSQL (engine) API.

Chapter 6 (Service Discovery II: Semantic Extension of MSQL) presents the MSQL semantic extension (see Figure 1.2), its formal definition, the extended architecture, and the implementation of the MSQL engine to incorporate semantic based querying. It presents some example semantic queries demonstrating the use of the extended MSQL (engine) API.

Chapter 7 (Evaluation) presents an evaluation of the framework based on an experimental performance analysis and a comparison with related approaches. The analysis considers the effect of factors such as the framework configuration, the size of the queried repository, and the type of query on the performance of the publishing and discovery in the framework.

Chapter 8 (Conclusions) presents conclusions and recommendations of the dissertation.

Chapter 2

State of the Art

With the advent of the Internet and the World Wide Web (Web), there have been ongoing efforts of the mathematical community to use these technologies for the purpose of mathematics. Mathematical software has experienced rapid development from standalone software systems that are used and accessed directly by a user, to client server systems, to networked applications that interact with each other to satisfy the user's computational demands.

In this chapter, we highlight some of the most relevant approaches, technologies, and standards that have played remarkable roles in integrating mathematics into the net. We start by highlighting some of the early attempts to use the Internet to communicate mathematics. We then present Web technologies and standards that have shaped the current view of the field, in particular the concept of Web services, their architecture, registration, and discovery. We then proceed to the Semantic Web and standards used for the semantic description and discovery of information. Finally, we discuss some of the activities that have used Web technologies and standards to develop mathematical Web services.

2.1 Mathematics on the Net: Pre-Web Approaches

One of the pre-Web network services for mathematics is the Netlib [94] repository of freely available software, documents, and databases of mathematics. It provides mathematical documents and software by means of ftp or email. Recent improvements allow the use the Web for accessing the repository. Netlib is considered as general purpose repository in terms of its contents and tools used to search and retrieve them.

The Network-accessible Extensive Table of Integrals [47] approach is based on a server-client communication through sockets: a server program implemented in Lisp listens on a particular Internet port for queries to the table. A client program submits a query by connecting to that port via TCP/IP. The server program reads the query from the port and looks up

the table for an answer that it writes to the port.

The SymbolicNet [124] provides the possibility to access demonstrations of computer algebra systems by accessing dedicated remote servers.

The NetSolve project [95] makes some computational packages available to users through a variety of interfaces. The packages are provided as computational resources with known capabilities. NetSolve agents refer client requests to computational resources based on their capabilities.

The Internet Accessible Mathematical Computation (IAMC) [129] framework uses a HTTP-like protocol called the Mathematical Computation Protocol (MCP) to provide mathematical information and computation on the Internet via the interactive use of remote *compute servers* that individually provide specific computational services. These servers may exchange intermediate computational results for further processing. IAMC uses MathML [79] (see Section 2.5.1) for the representation of mathematical data together with a proprietary binary format for compact encoding.

Although these systems are available and accessible via the Internet, they fall short in terms of interoperability and generality in providing and facilitating mathematical computation over the Web. In Section 2.5, we present more promising approaches that overcome these shortcomings by employing technologies such as those presented in Sections 2.2 to 2.4.

2.2 Web Services Architecture

The Web has witnessed three remarkable faces of development [111]:

- The first is the *Document Web* which is supported by Web servers that used Hypertext Transfer Protocol (HTTP) to deliver information marked up in Hypertext Markup Language (HTML) for display in Web browsers.
- The second is the *Application Web* which is supported by Web application servers with which users interact and dynamically generate HTML documents from server-side applications.
- The third is the *Service Web* which is supported by a stack of standards and protocols allowing applications distributed over the Web to form interacting Web services.

Web services have recently gained more attention as a promising solution for the building and integration of distributed software components across the Internet. They provide means for application-to-application communication and interoperability using Web standards. A commonly used definition of a Web service is [131]:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an*

> *interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

Concerned communities (e.g., the World Web Consortium (W3C) [128] and the Organization for the Advancement of Structured Information Standards (OASIS) [96]) have defined and deployed a stack of standards and languages that are forming the infrastructure of Web services. Among these standards are the *eXtensible Markup Language (XML)* [25], the *Simple Object Access Protocol (SOAP)* [118], the *Web Services Description Language (WSDL)* [134], and the *ebXML Registry Standard (ebXML)* [45] and the *Universal Description, Discovery and Integration (UDDI)* [126] registry standard.

XML is a flexible and widely accepted format on many differing platforms and languages for representing data to be communicated among (with the goal of interoperability) interacting applications, particularly those that communicate across the Internet. Almost every standard or work mentioned in this thesis relies on XML.

SOAP is a standard for communicating XML messages in a decentralized, distributed environment such as the Internet. It sets the rules governing the format and the processing of a message. SOAP comprises three parts: an XML envelope for describing the contents of a message, a set of encoding rules for encoding data as XML, and a mechanism (binding) for using SOAP (e.g., in an RPC style) over a transport protocol such as HTTP.

Before a client can access a Web service, the service interfaces must be formally described in an XML format. *WSDL* is a standard that uses XML to describe the protocol used to access the service, the operations that can be performed, and the input and output message formats.

The WSDL interfaces are static in the sense that they are not expressive enough to allow two Web services to interact on behalf of a client. Standards for Web service composition are being defined to support such interactions dynamically. The Web Services Conversation Language (WSCL) [133], The Web Services Choreography Interface [132], and the Business Process Execution Language for Web Services (BPEL4WS) [24] are such standards for describing service compositions and the dynamics of message protocols.

ebXML [46] and *UDDI* [126] are standards for defining registries where Web services providers can publish services, and clients can discover them (see Section 2.3 for more details).

The W3C "Web Services Architecture" specification [131] presents a typical services oriented architecture that involves a stack of the above standards: XML for representing data, SOAP for exchanging messages, WSDL for describing Web services, and a registry for publishing and discovering

Web service. The architecture describes three service roles: *service provider*, *service requester* and *service broker*; and three basic operations: *publish*, *find*, and *bind*.

A *service provider* hosts a network accessible implementation of the service and defines a description for the web service and *publishes* it to the *broker*. A *service requester* can be a software component that requests the execution of a service. It uses a discovery mechanism to *find* required services using a *service broker* and *bind* to them using a *service provider*.

In the next section we discuss the process of service registration and discovery in more detail.

2.3 Web Service Brokering

The *brokering* of a Web service is the process of publishing[1] a description of the service (e.g., to a registry) by a service provider and the discovery and invocation of the service by a service requester.

Any operation that makes a service description available for a user can be viewed as service publishing and any operation allowing access to the service description for a Web service can be viewed as a service discovery. There is a variety of service discovery approaches available with three leading approaches [131]: based on indices, peer-to-peer (P2P) technologies, and registries.

- An *index* is a compilation or guide to information that exists elsewhere. It is not authoritative and does not centrally control the information that it references. The search engine Google [54] is often cited as an example of the index approach.

- A *peer-to-peer (p2p)* discovery is based on a decentralized network of Web service peers that discover each other dynamically.

- A *registry* is a web-based shared resource that enables the creation, deployment, and discovery of Web services. Information shared by Web services is maintained as objects in a repository and managed by registry services with specified interfaces.

A registry is the most powerful concept because it can encompass both the index and P2P approaches. For instance a registry could be used as a means to implement an individual index [131], or a set of registries that act in a federation as a P2P system [112].

Currently there are two predominant specifications for a registry: the *Universal Description, Discovery and Integration (UDDI)* registry standard [126] and the *Electronic Business using XML (ebXML)* registry standard [46].

[1] The terms publishing and registration are used interchangeably.

The *UDDI* registry manages information about service providers, service implementations, and service metadata. Web Service providers register their services such that clients find them. A network of UDDI registries can be used such that a provider only has to register a Web service at one of the registries and the registration and update will be replicated in all the other registries in the network.

The *ebXML* registry provides a set of services (most important: registration and discovery) that enable sharing of information (content and metadata) between party entities in a federated environment, i.e., a loosely coupled system of ebXML registries that appear to clients as a single logical registry. Metadata are pieces of information about a published object stored in the registry's database. Content is a descriptive information in some format, e.g. XML, accompanying the published object and is stored as a repository item (as a file) in the file system of the registry.

The two registry standards are different in terms of architecture and information model. An UDDI registry is focused and specific, while an ebXML registry is generic. For example, the *findBusiness* operation in UDDI is a business processes intended operations, while the *findObject* operation in ebXML is a registry object oriented. This means that the design focus of UDDI is a high level business registry and the design focus of ebXML is a general purpose distributed content and metadata registry. These focuses are reflected on the information models of each registry where the information model of UDDI is closed while the information model of ebXML is extensible. The UDDI registry is focused on the discovery aspect of "businesses", i.e., to make their Web services and technical interfaces available. ebXML is focused on both discovery and collaboration [56]. UDDI has a tightly coupled architecture that requires tight contracts between registry operators. ebXML has a loosely coupled architecture which requires no prior contract between registry operators.

Despite of these differences between the two standards, there is a possibility of interoperability between them, e.g., it is possible for an ebXML registry to (through its *Registry Service Interface*) interact with a UDDI registry [56].

There have been some efforts to enhance the registry framework so it can be widely accepted as a service publishing and discovery paradigm. The METEOR-S Web Services Discovery Infrastructure (MWSDI) [127] aims at providing efficient mechanisms for publishing and discovery in a multi registry environment. One such environment of MWSDI is a peer-to-peer (P2P) network where for the different kinds of functionality, different kinds of peers are implemented. The *Gateway Peer* allows registries to join MWSDI, The *Operator Peer* operates a joining registry, The *Auxiliary Peer* provides the criteria for partitioning, and The *Client Peer* allows a user to use the MWSDI. In [116] an attempt is made to leverage MWSDI to provide transparent access to private and public Web service registries by creating

a domain-based registry federations.

Since the ebXML registry is generic and its information model is extensible, we base our mathematical registry framework on the ebXML registry (see Chapter 4). In the next section, we introduce briefly the process of registration and discovery in the ebXML registry. In Chapter 4, we will present in more detail the architecture and the functionality of the ebXML registry.

2.3.1 Service Registration and Discovery

The ebXML registry architecture [46] consists of an ebXML *Registry Service* and an ebXML *Registry Client*. The ebXML *Registry Service* provides the ability for managing a repository. An ebXML *Registry Client* is an application for accessing the registry using registry protocols.

Service Registration

Service registration is the act of submitting to the registry the metadata of a service combined, optionally, by some related content as a repository item. A *Registry Service* provides the *LifeCycleManager* interface that provides a collection of operations for the management of metadata and content. These operations include publishing, update, approval, and deletion of metadata and content. Metadata are stored in the registry database while content (e.g., service descriptions) is stored in the repository of the registry. Architecturally, the registry is distinct from the repository. However, all access to the registry as well as the repository is through the operations defined by the *Registry Service* interfaces.

Service Discovery

Service Discovery is the act of locating a machine-processable description of a Web service. It involves querying the registry for metadata and content. The registry's *QueryManager* interface provides a collection of operations for the query and retrieval of metadata and content within the registry.

The ebXML registry currently supports two kinds of query capabilities [46] that allow a client (an object of type *QueryManager Client*) to search for or query different kinds of registry objects in the registry:

- *A Filter Query* has an XML syntax [46] with alternatives each directed against a single class defined by the information model of the registry. It aims at reducing the amount of processing on the server side.
- An *SQL Query* allows a client to submit complex SQL queries using the syntax of the SELECT statement as defined by [62] and [119].

The following scenario takes place when performing a query:

- A client submits a query to the *Query Manager* of the registry by sending a *Query Request*.
- The *Query Request* contains a sub-element that defines a query in one of the supported registry query capabilities.
- The *Query Manager* sends a *Query Response* back to the client.
- The *Query Response* returns a collection of objects whose element type depends upon the *Response Option* attribute of the *Query Request*.
- Any error in the query request message is indicated in the corresponding *Query Response* message.

The *Registry Client*, which may be a Graphical User Interface (GUI), typically accesses the registry using the SOAP with Attachments [123] protocol. A *Registry Client* may run on a client machine or may be a Web service running on a server and may be accessed by a web browser. It may access the registry interface directly or using a registry client API such as the Java API for XML Registries (JAXR) [64] which is designed with the intention of easy-to-use abstraction API to access any XML registry including UDDI registries.

The ebXML registry standard (also the UDDI registry standard) does not support content-based discovery. The capabilities of ebXML supports queries based on the metadata of the registered services but not based on the content contained in the underlying descriptions. XML content-based query languages and Semantic Web query languages can be used for such purposes. In Section 2.3.2, we present some the query languages for XML-based content. In Section 2.4, we present some query languages for semantic-based content.

2.3.2 Querying XML Content

Existing XML query languages range from ones that support simple node finding and path expressions to more comprehensive ones that support processing, transformation, and querying tasks of XML documents. The QUILT [31] query language attempts to unify concepts from some of these query languages in order to exploit the full versatility of XML. Its proposal has been adopted latter as the basis for the development of XQuery [22].

XQuery is intended primarily as a query language for querying collections of XML documents or documents viewed as XML (e.g., a SOAP representation of any data source). The language is designed to be broadly capable of dealing with many sources of XML and not only query them, but also process them, and have new XML structures as results. XQuery has a formally defined operational semantics [33] to provide a processing model for the language. This semantics uses value inference rules to relate

XQuery expressions to values and specify the order in which expressions are evaluated.

XQuery expressions include path expressions, element constructors, list expressions, FLWOR (for-let-where-orderby-return) expressions, conditional expressions, quantifies expressions, data type expressions, and function call expressions. XQuery operates on a data model [32] that is based on heterogeneous sequence of nodes and atomic values. The data model describes inputs to and outputs from an XQuery processor. A data model instance may contain one or more XML documents or fragments of such documents and a query provides a mapping from one instance of the data model to another.

2.4 Web Service Semantics

The Semantic Web [115] is an extension of the current Web in which the semantics of data formally described, enabling data to be shared and processed in a machine understandable manner. It uses XML-based standards and technologies such as the Resource Descriptions Framework (RDF) Core Model [105], the RDF Schema [26] language, and the Web Ontology Language [99]. A major goal of the Semantic Web is to provide service discovery. Service descriptions are presented in formal languages or ontologies (an ontology is a representation of the meaning of terms and their interrelationships [81]) whose meaning is well defined and unambiguous. For example, given a set of service descriptions and a job specification written using a suitable collection of ontologies, a software agent takes care of selecting the appropriate service for the job and facilitates the interaction between the client and the service.

2.4.1 Semantic Representations

We present a short introduction to RDF [105] and OWL [99] for semantic content representation. Then we present three common frameworks for Semantic Web Services including OWL-S [100] and WSMO [136] as languages for semantic description of Web services.

The Resource Description Framework (RDF)

The Resource Description Framework (RDF) [105] is a language for representing information about resources in the Web and expressing this information so it can be exchanged between applications without loss of meaning. The model of RDF is based on the idea of identifying resources using Uniform Resource Identifiers (URIs), and describing them in terms of simple properties and property values. Statements about resources are represented as a graph of nodes and arcs. RDF/XML [109] defines the XML syntax for such

RDF graphs. An RDF statement is a triple of subject, predicate, and object. Statements describing application-specific resources and specific properties in describing those resources may need to define vocabularies (terms) that are not part of RDF. They can be described as an RDF vocabulary using RDF Schema.

Web Ontology Language (OWL)

The Web Ontology Language (OWL) [99] builds on top of RDF and RDF Schema (RDF-S) with more constructs for expressing meaning and semantics providing greater machine interpretability of data. It is designed to be used by applications that need not only present but also process the content of documents. It provides vocabularies with explicit representation of terms in these vocabularies along with their relationships (this is called ontology). OWL is a revisal of the DAML+OIL Web Ontology Language [37] integrating insights gained from the design of DAML+OIL. OWL has three sublanguages:

- OWL Lite – uses only some of the OWL language features. It is used in describing classification hierarchies, i.e. ontologies, and simple constraints.

- OWL DL – provides maximum expressiveness while conclusions are guaranteed to be computable in finite time. It includes all OWL language constructs, but they can be used only under certain restrictions. OWL DL is so named due to its correspondence with Description Logics [8], a field of research that has studied the logics that form the formal foundation of OWL.

- OWL Full – provides maximum expressiveness and the syntactic freedom of RDF. An ontology in OWL Full can extend the meaning of a vocabulary of RDF or OWL. Automated reasoning is not supported for every feature of OWL Full [101].

RDF and OWL are Semantic Web enabling standards that build up a rich set of constructs for describing the semantics of Web-oriented information. For describing the semantics of Web services (so they can be automatically discovered, composed, and executed), there have been several approaches. An early approach [82, 83] to bring semantics to Web service uses DAML [58, 59] for capturing data and metadata of a service together with specifications of the service properties and capabilities, the interface for its execution, and the prerequisites and consequence of its use. Currently there are three main approaches to Semantic Web services: the IRS-II (Internet Reasoning Service) [88], OWL-S [100], and WSMF [48].

Internet Reasoning Service (IRS-II)

The Internet Reasoning Service (IRS-II) [88] is a Semantic Web services framework which allows applications to semantically describe and execute Web services. Its architecture consists of the *Server*, the *Publisher*, and the *Client*. The *Server* keeps ontology-based descriptions of Web services. The *Publisher* links Web services to their descriptions in the *Server* and generates service access endpoint. The *Client* can be an IRS-II browser that is used for service description navigation and editing and for service publishing and invocation.

The Ontology Web Language for Services (OWL-S)

The Ontology Web Language for Services (OWL-S) [100] (previously known as DAML-S) is an ontology for Web services. It is a step forward in combining Semantic Web technologies and Web services. It allows semantic Web tools to process and treat not only Web content (such as documents) but also Web services in order to enable software agents in a semantic web setting to automatically discover, invoke, compose, and monitor Web services. OWL-S has three main subontologies: the *service profile* for publishing and discovering services; the *process model*, which gives a detailed description of a service's operation, and the *service grounding*, which provides details on how to interoperate with a service via messages.

Web Service Modeling Framework (WSMF)

The Web Service Modeling Framework (WSMF) [48] aims at describing the various aspects related to Web services by applying Semantics Web technologies. WSMF includes the Web Service Modeling Ontology (WSMO) [136] which is a formal language for semantically describing Web services to facilitate the automation of discovering, composing, and invoking such services. WSMO has four elements: ontologies, which provide the terminology used by other WSMO elements; descriptions, which describe the functional and behavioral aspects of a Web service; goals that represent user desires; and mediators, which aim at automatically handling interoperability problems between different WSMO elements. WSMO uses a formal logic language, the Web Services Modeling Language (WSML) [135], for formally describing all the elements defined in WSMO.

Among the advantages of WSMO when compared to OWL-S are its separation of the requester and provider roles at the conceptual model, more natural descriptions of user requests, and support of mediators as a modeling element in WSMO (instead of being an element of the underlying Web service infrastructure as in OWL-S [102]).

2.4.2 Discovery in the Semantic Web

Semantic descriptions of Web services can be exploited in the process of service discovery.

The METEOR-S project [84] aims to create an infrastructure for complete lifecycle management of Semantic Web services. It includes METEOR-S Web Service Discovery Infrastructure (MWSDI) [127] to provide efficient discovery mechanisms.

An approach [72] to bring semantics to Web services uses OWL-S to foster the deployment of Semantic Web services. OWL-S is used in conjunction with WSDL to add semantic descriptions to WSDL description of a Web service. When a registry is used for the publishing and the discovery of a service, OWL-S is used in this respect to add capability, what a service can do, matching to a registry.

A registry can be used as a basis for the (semantic) discovery of web services. In a UDDI registry, one approach [23] proposes to add semantics to the interactions of roles including service provider and client. While the discovery process does not change, service provider provides a semantic description of the service and client uses a semantic approach in the service discovery.

An approach to registry-based semantic discovery [39] proposes to enrich an ebXML registry with OWL ontologies to describe registered Web services. The various constructs of OWL are mapped to ebXML classifications; stored semantic description are queried using the ebXML query facility.

Another approach[116] to semantic based service discovery uses DAML-S service descriptions and matchmaking in a federated registries environment.

Next we present specific query languages that are used to leverage discovery mechanisms for RDF and OWL based descriptions.

RDF Query Languages

There are various RDF query languages [108] with many similarities among them. Many of these languages describe an RDF graph with parts missing, assign those parts variable names, and get a series of bindings to those variables. We present here some of such query languages; a comprehensive survey is given in [108].

Algae [3] is a query language that provides query and assertion access to RDF graphs. It is designed to query a graph, insert data into a graph, or write rules to automatically insert data when a query is matched. One of its main uses is with the Annotea [4] which is a protocol that enhances collaboration via shared metadata based Web annotations (such as comments, notes, explanations, or other types of external remarks that can be attached to any Web document), bookmarks, and their combinations.

SquishQL [120] is an SQL-like query language for RDF. It supports a

query model based on a graph pattern formed from variables for nodes, arcs and literals. Additionally it uses filter functions in the form of Boolean expressions over the variables to restrict their values.

RDQL [110] is a language that evolved from several query languages specially from SquishQL. Its purpose is to extract information from an RDF graph by treating RDF as data and providing query with triple patterns (each triple pattern is made of named variables and RDF values) and a set of constraints on the values of the variables. RDQL is used as the query language for RDF in Jena [65] models.

RDFQL [107] is an SQL-like language that is used by RDF Gateway [106] applications for querying RDF data models. "Agents" that submit queries to RDF Gateway service use RDFQL to describe the information they need as a *query condition* and it is processed by the service to find documents on the Internet containing the exact information described by it. RDFQL uses SQL commands such as *INSERT*, *DELETE* and *SELECT* to perform query operations on RDF triples.

OWL Query Languages

Several query languages and tools developed for RDF can be used as a basis to query documents presented in OWL. For instance, in addition to providing facilities for querying RDF structured data, an important feature of RDFQL is its ability to infer new statements from existing ones by using user-defined inference rules of the form "if A is the case then so is B" for powerful deductive searches.

OWL Query Language (OWL-QL) [49] is intended to be a candidate standard language and protocol for query-answering dialogues among Semantic Web computational agents (answering agents (servers) derive answers to question posed by querying agents (clients)) using knowledge represented in OWL. It specifies the semantic relationships among a query, a query answer, and the knowledge base. The knowledge base may comprise multiple knowledge bases on the Semantic Web that are not necessarily specified by the client.

2.5 Mathematics on the Web

Most of the pre-Web mathematical systems (see Section 2.1) suffer from the pre-Web architectural designs and obsolete technological solutions [5] affecting their integration into the Web. New approaches have adapted current Web technologies for integration of mathematics on the Web. In this section we highlight some of these approaches that started with the advent of the Web, Web services, and recently the Semantic Web.

2.5.1 Mathematical Representations

XML has been used extensively for the representation, storage, retrieval, and manipulation of mathematical content. XML-based standards provide portability and easy accessability to content-based mathematical representation. In this respect there are two major standards: the Mathematical Markup Language (MathML) [79] and OpenMath (OM) [27] standard.

The primary objective of MathML [79] is to present mathematical formulas on the Web. It has presentation encoding (P-MathML) for displaying mathematical formulas and content encoding (C-MathML) to capture the semantic structure of mathematical formulas. MathML is supported by some of the recent Web browsers.

The OpenMath (OM) [27] standard has been developed primarily for representing the semantics of mathematical objects. OM posses the feature of converting, relying on Content Dictionaries (CDs), between the XML representation of mathematical objects and their system-specific representations. This conversion can be done be a tool called Phrasebook [28].

MathML and OM can be used together such that MathML representations include OpenMath to present semantic annotations and OpenMath representation include MathML for formula presentation.

OMDoc (Open Mathematical Documents) [97] is an extension of the OpenMath standard to markup mathematical structures at the document level. It provides a standardized way for the representation of the semantics of mathematical documents.

Almost any approach aims to serve mathematics on the Web nowadays uses one or all of these standards. In the next subsection, we present some approaches to integrate mathematics in the Web including Web services. These works also employ standards and technologies introduced earlier (see Section 2.2).

2.5.2 Mathematical Services

A mathematical Web service can be defined in line with the definition of a Web service as a solution to a mathematical problem that is available on the Web and can be accessed by a user or another service or program. The description of a mathematical service may contain information related to the type of the problem, algorithm(s) used to solve the problem, related problems, machines executing the problem, etc. A characteristic that makes mathematical services differ from other kinds of services is the fact that they are designed to deal with the mathematical domains that are known to be semantically rich with objects that need proper encoding and specifications.

In the following, we identify some standards and tools [21] for implementing mathematical Web services. The standards are required by the three proposed components of a mathematical Web services: the message

exchange component where standards such as SOAP are used, the description component where standards such as WSDL are used, and the discovery component where registry-based approaches can be used.

A proposal on integrating mathematics into the Semantic Web [71] has identified RDF, OWL, XML-Schema [137], and XQuery [22] as key standards that can foster such an integration.

The Hypertextual Electronic Library of Mathematics (HELM) [57] project has developed some software tools for the creation and maintenance of a virtual, distributed, hypertextual library of structured and formal mathematical knowledge. The main emphasis of HELM is on rendering, browsing, management, searching and retrieving issues. It is more oriented towards proof assistant applications than computer algebra systems. It uses the Coq proof assistant [34] and some XML-based technologies such as MathML and RDF as essential components. Section 2.5.3 presents the searching approach of HELM. A related is the MoWGLI (Mathematics On the Web: Get it by Logic and Interfaces) project [89] which aims at developing a machine-understandable representation of mathematical information and a tool for its exploitation. It builds on the MathML, OpenMath, and OMDoc standards integrated with Sematic Web standards such as RDF.

The MathWeb project [80] has developed the MathWeb Software Bus (MathWeb-SB) a software system that connects a range of mathematical systems using a *common mathematical software bus*. It provides some functionalities to integrate a set of running theorem proving systems into a networked of proof services. As one mathematical service of the MathWeb that is part of the MathWeb-SB, the MBASE [69] system offers a distributed repository of formalized mathematics represented in OMDoc. MathWeb-SB can be accessed remotely via XML-RPC remote calls. ActiveMath [2] is an OMDoc-based Semantic Web application for mathematical learning that can be connected to the the MathWeb-SB.

Some computer algebra systems such as Mathematica [75] and Maple [70] offer remote computations using programmatic interfaces such as J/Link [67] and MathLink [78] in the case of Mathematica and *socket-server* in the case of Maple. Mathematica offers the *Web Services Package* [77] for calling web services from Mathematica. The package employs Web services standards such XML, XML Schema, SOAP, and WSDL as part of its functionality. webMathematica [130] provides Mathematica computational capabilities to the Web. It adds interactive calculations and visualization to a Web site by integrating Mathematica with Web server technology and uses MathML for presenting mathematical content.

The main activity of the Mizar project [85] is the development of a database for mathematics called the Mizar Mathematical Library (MML) [9]. MML consists of Mizar articles that contain definitions and theorems and are stored as text files. An article consists of two parts: the *environment declaration* and the *text proper*. The *environment declaration* consists of di-

rectives concerning imports from the database. The *text proper* is a sequence of *sections* consisting of a sequence of *text items* such as definitions and theorems with their proofs [9]. A submitted article to MML gets published to the Web-based electronic *Journal of Formalized Mathematics* [66].

The MONET project [87] developed an architecture [121] for brokering mathematical web services. It involves: a number of software components such *brokers*, *planners*, and *servers*; various ontologies for the description of mathematical services and problem domains; and languages to communicate mathematical problems, results, and explanations. The part of the architecture concerned with brokers and planners needs further research [117]. The architecture has three groups of components: *servers*, *brokers*, and *clients*. A server advertises its services by registering them with a *broker*. A *client* sends a query to the *broker* for Web services solving specific problems. A *broker* executes a query by looking for available services and passing the client's request to them. Services carry the request by solving the problem and returns the results to the *client*.

With mutual influence from our MathBroker project [73], the MONET project has developed its version of the Mathematical Services Description Language (MSDL) [30] (see Chapter 3). Based on MSDL, MONET investigated as part of its architecture the semantic description and discovery of services using OWL ontologies. For example, it defines a problem ontology to describe a problem as posed by the user and a query ontology to describe a query for broker submission. The query ontology either consists of a problem ontology or an MSDL service description and a user preferences in RDFS. In Section 2.5.3, we discuss the way ontology-based querying takes place in MONET.

The MathBroker project [73] developed a software framework for mathematical services (see Chapter 3) based on mathematical and Web standards for Web services such as SOAP, WSDL, and OpenMath. It also developed, with influence from MONET project, the Mathematical Services Description Language (MSDL) (see Chapter 3) for describing mathematical services. In Chapter 3, we will give more details on the development of mathematical Web services and how they are described in MSDL within the MathBroker project [73]. The work in this thesis, the mathematical registry framework (Chapter 4) has been pursued within the MathBroker project [73]. The discovery framework (Chapters 5, 6) has been pursued within the MathBroker II project [74]. MathBroker II project also aims at developing, based on Semantic Web concepts such as those introduced in Section 2.4, a process model for MSDL to formally describe interactions with mathematical services [41].

2.5.3 Mathematical Searchability on the Web

In this section, we present some of the approaches for searching mathematics used in the projects mentioned above. First we give a general description of such approaches as seen by the MKM-NET [86] project.

Searching Directions in MKM-NET

The Mathematical Knowledge Management Network (MKM-NET) [86] aims at finding ways to manage mathematical knowledge. Among its objectives is mathematical knowledge representation and searchability.

The MKM-NET document, Mathematical Knowledge Management and Searchability [36], describes the state of the art for searching mathematical content. It states the following approaches for searching mathematical content ranging from basic textual methods to semantic-driven techniques.

- *Fulltext Search* uses a basic textual search approach. If the content consists of formulas, the search interface uses tools such as Latex shells to enter a formula.
- *Semi-textual Search* depends on a semi-natural structure of documents using search tools such as *grep*. A retrieval system has been integrated with *grep* as part of the Mizar project [85].
- *Semantic Search for Terms* depends on agreed upon standard semantic representation, such as OMDoc, that can be exploited to improve the search. This approach is used in MBase [69] (see Section 2.5.3).
- *Fast Filtering of the Search Space* applies a filtering procedure on a set of candidate matches to produce a small set of candidates. In this respect HELM (See 2.5.3) proposed an approach based on four phases: *data mining* to compute a small set of metadata from each theorem or definition in the library, *pattern compilation* to compile a semantic query into a low-level query, *filtering* to execute a low-level query over a database of metadata, obtaining a set of candidates, and *matching* for executing the actual semantic query on the set of candidates obtaining a precise result set.
- *Semantic Search for Knowledge* used when natural language sentences in mathematical documents have a precise meaning.
- *Key Phrase Search* evaluates the relevance of the occurrences of some mathematical content returned by a search using indexing techniques.
- *Exploiting Semantic Relation* deals with content that is related to other content where semantic relations between pieces of content are binary. For example, theorems have proofs which make use of other theorems.

- *Combination* combines semantic relations with key phrase search and types of content by means of an automated inference engine to ensure the relevance of the mathematical objects retrieved.

A recent survey to mathematical searching tools on the Web [6] identifies three groups of queries: *bibliographic searches* for retrieving documents based on bibliographic information, *mathematical services* where a request is issued for finding a Web service that solves a problem with the help of some computational tool(s); the MONET approach is identified as one candidate for satisfying such type of search, and *content-based searches* where queries are based on "fine-grained" analysis of the mathematical content of the information; the MoWGLI project is identified as a candidate for realizing this kind of search.

Searching in MBase

Mathematical Knowledge Base (MBase) [69] is a specific database (a set of collections of OMDoc documents) supporting the semantic retrieval of terms. Currently MBase has the following facilities (implemented as Web interface) for retrieving content from the database:

- *QuickSearch* finds a text fragment in any mathematical element. For example, a search for the text fragment "bool" would return as a result collections (e.g., logics, OMCDs), theories, and items where the text fragment occurs.

- *Pattern Search* is used for structural queries on the set of stored terms and formulas. for example, searching for the pattern "eq(a(F X Y) a(F Y X))" (which represents the commutativity f(x,y) = f(y,x)) would return as a result all found matching terms.

- *Specific Elements Search* is used for specific mathematical elements such as theories, symbols, definitions, axioms, and theorems.

Although it provides these search and query facilities, MBase still does not have a specialized query language yet. The open query requirements MBase is set to support are [69]:

- General SQL-like queries based on the structure of the OMDoc input.

- Queries which match formulae under a given equality theory. An example is:

```
If a + (b + c) is in MBase, then also
  (b + c) + a and (a + b) + c
should be found.
```

- Queries with meta-variables. An examples is:

```
Give me at least one lemma of the form
 Forall x.G(x) <= f(x) <= H(x)  where
  G and H should be some Meta-Variables
    which match some term in x. f(x)
    is a given term, e.g. sin(x).
```

Information Retrieval in Mizar

Information Retrieval in Mizar Mathematical Library (MML) [9] amounts to searching, browsing, and presentation of content. It mainly provides:

- HTML based browsing.
- Text based searching which uses tools from the *grep* family and includes searching for theorems and searching for definitions.
- Semantic based retrieval which makes use of operations on *resources*. These are pieces of information stored in internal MML files such as *constructors*, *patterns*, *statements*, *theorems*, etc. For example, the following query returns the list of resources where both the listed constructors occur (`& occur`) and then filters all theorems from the list (`| th`).

```
{XBOOLE_0:func 1, XBOOLE_0:func 3} & occur | th
```

Searching and Retrieving in HELM

HELM [57] distinguishes between two kinds of basic queries:

- Not specific to mathematical domains using information such as names, keywords, or authors. An example query is "Search and retrieve all the theories in which a given knowledge item is used/referenced".
- Specific to mathematical domains (such as proof-searching). It allows to search for objects including terms satisfying some given constraints expressed as match patterns on types. Examples are:
  - Search and retrieve all the theories regarding Linear Algebra.
  - Search and retrieve all the theorems that are applicable to a given set of hypotheses.
  - Search and retrieve all the theorems whose conclusion matches a given type pattern.
  - Search and retrieve all the proofs of a given statement.

The first kind is implemented in MathQL-1 (see below). The second kind of search is a work under progress (MathQL-1, MathQL-2, and MathQL-3). HELM uses metadata to help this kind of search by associating to every theory, collection of objects or a single object an XML file (a metadata model) with the relative meta-information. Metadata model can be expressed in RDF. RDF metadata can be queried using two approaches:

- Inserting RDF metadata in a relational or XML database and use SQL for querying.
- Treating RDF metadata as a knowledge base and apply knowledge representation and reasoning techniques.

HELM has an MathQL proposal which aims at the development of a set of query languages:

- MathQL-1 is focused on querying an arbitrary RDF database. The peculiar aspects of this language concern the query results.
- MathQL-2 will include content-based pattern-matching.
- MathQL-3 will include other forms of formal matching involving for instance isomorphism, unification and definitions expansion.

Querying in MONET

MONET [87] defines a set of ontologies to model service descriptions (which are basically ontological conversions of MSDL [38, 90] descriptions) as well as queries on those descriptions. These ontologies are, written in OWL [99] and they are used by a component within the MONET architecture called *Instance Store* [60]. *Instance Store* uses a Description Logic reasoner called RACER [55] for matching queries to appropriate services.

MONET uses two classes of ontologies: those describe models internal to MONET (e.g., problem and software ontologies) and those describe models external to MONET (e.g., OpenMath and GAMS ontologies). Individual ontologies of both classes are imported into one MONET ontology.

When a service is submitted to the MONET broker, its description is presented in MSDL. This description is transformed into the OWL Abstract Syntax [19] by means of an XSLT stylesheet. Service matching is then performed by submitting a query to the *Instance Store* in the form of an OWL description. The *Instance Store* answers the query by using a combination of Description Logic reasoning and database queries.

The following query asks for all individuals whose GAMS classification is *GamsG*. The Instance Store answers the query by returning *nagopt* and *nagopt-variation*.

```
restriction(<http://monet.nag.co.uk/owl#service_class>
     someValuesFrom(
        restriction(<http://monet.nag.co.uk/owl#gams_class>
          someValuesFrom(<http://gams.nist.gov#GamsG>))))
```

A matching-based approach [93] for discovering registry-published mathematical services in MONET performs matchmaking between representations of tasks (client requests) and capabilities (service descriptions). The approach applies a normalization process on a task. It then compares the normalized task with a registered capability calculating a similarity value that is used in the matchmaking process. Task normalization amounts to carrying out a sequence of transformations on the task description rewriting all logical parts in disjunctive normal form, flattening arguments of n-associative operations, and consistent variable renaming.

The similarity value is calculated based on the matching of the capability precondition (or the task postcondition) and the capability postcondition (or the task precondition). Matchmaking is performed by: registering capabilities in the database, taking a description of a task normalizes it, and returns an ordered list of the capabilities from the registry database based on their calculated similarity. The matching process used in the discovery is ultimately based on the syntactic similarity traced between tasks and capabilities.

2.6 Summary

We have reviewed the state of the art in integrating mathematics into the Web. Our survey has highlighted some of the early attempts to use the Internet to communicate mathematics and have then presented Web technologies and standards that have shaped the current view of integrating mathematics into the Web. Emphasis was on Web service architectures and technologies that are used to integrate mathematics into the current Web and in the Semantic Web. There are numerous project, approaches, and works to integrate mathematics in the Web. They range from standards for the representation and semantic-encoding of mathematics to those for brokering mathematical problems and solutions in traditional Web approaches and on approaches that use the Semantics Web technologies. Discovery and information searchability for mathematics have evolved from simple text finding tools to ontology-based tools. In the light of this review, we are going to present our own framework for the registration and discovery of mathematical Web services which introduces some new ideas to the field.

Chapter 3

Service Development and Description

The work reported in this thesis is carried out in the frame of the MathBroker project [73, 74] and constitute an important part of it. The aim of this project is to develop a framework in which mathematical (Web) services are developed, described, and brokered. In this chapter we present the approach used for developing and describing mathematical services. Of particular interest for this thesis is the description approach based on the Mathematical Services Description Language (MSDL) [30, 38]. We use MSDL as the target format for service publishing and discovery and publish MSDL service descriptions in a registry; service discovery makes use of the syntax as well as the semantics of these descriptions.

3.1 Service Development

The MathBroker framework uses current Web service standards and technologies referenced by the Web services architectures, e.g. XML, SOAP, and WSDL (see Section 2.2). It also uses mathematical specific standards such as OpenMath (see Section 2.5.1) as the infrastructure for developing and deploying mathematical Web services.

3.1.1 The Service Architecture

The service architecture supports the scenario of developing a mathematical service, deploying it in a service container (servlet) hosted by a server, facilitating contact with a mathematical software system to perform the required computations through a system-specific interface, and exposing the service to clients through a well-defined interface. The architecture is illustrated in Figure 3.1 and consists of the following components:

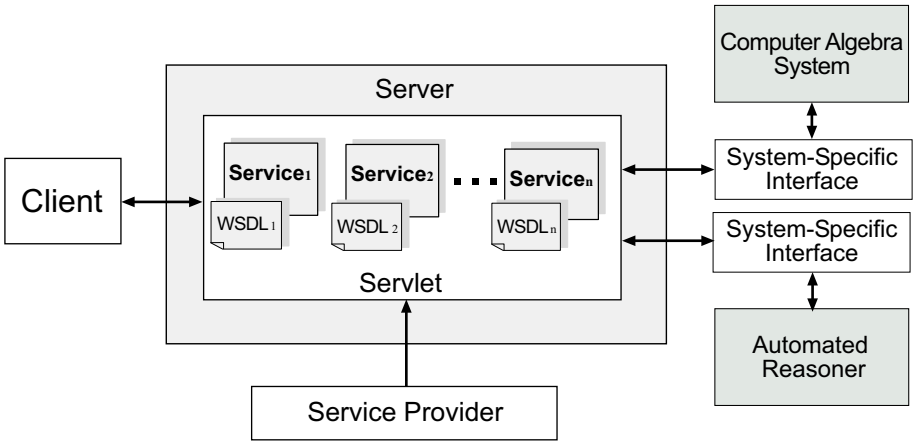


Figure 3.1: The MathBroker Web Services Architecture

The Service Provider implements and deploys mathematical Web services.

The Service is the implementation deployed in the servlet and is identified by a URL.

The Server hosts the servlet that administers the deployed mathematical services. The server provides the URLs at which clients can access the deployed services.

The Servlet an engine that allows the deployed services to communicate with the clients. It allows to deploy/undeploy services, lists them, provides descriptions of their interfaces, supports client-side code generation from the descriptions of the deployed services, and handles client requests to the services.

The Mathematical Software System can be any mathematical software system that performs mathematical computations, e.g. a computer algebra system such as Mathematica [75] or GAP [52] or an automated reasoner such as the Cooperating Validity Checker Lite (CVCL) [35]. Every system accepts mathematical input expressions and returns results in a system-specific syntax.

The System Specific Interface allows a service to apply a mathematical software system independent of the syntax used by the system by translating requests and results from/to a common format.

The Client can be a Web based or a command line client program or in general any application that accesses the service via its URL using its interface.

3.1.2 An Implementation of the Architecture

The MathBroker project has realized the above architecture by using Web technologies and standards (see Section 2.2).

The server used is the Apache Tomcat Web server [125] which is a highly configurable server for executing Java servlets. This server hosts the AXIS (Apache eXtensible Interaction System) [7] servlet, which is a service container that implements the SOAP protocol; within this container multiple Web services can be deployed. For each service, a Java class representing the executable code of the service and a deployment descriptor describing the interface of the service have to be provided. Based on this infrastructure, we have implemented mathematical services that use as mathematical software systems the computer algebra systems Mathematica and GAP and the automated reasoning system CVCL.

As the common language for communicating with these services, we use the OpenMath language [27]; the system-specific interfaces are implemented with the help of:

- the RIACA OpenMath library [98] which provides *phrasebooks* [76, 53] for Mathematica and GAP. A *phrasebook* is used to convert from OpenMath format to a mathematical system specific syntax and vice versa.
- the RISC ProofNavigator [114] which provides an interface to CVCL.

A client, whether it is a Web-based or a command line program or an application, invokes the service at its URL via its WSDL-defined interface with OpenMath expressions as arguments.

In the next section, we demonstrate the architecture realization by two example services, one uses the computer algebra Mathematica and the other uses the automated reasoner CVCL.

3.1.3 Example Services

The Symbolic Integrator Service

The *symbolic integrator service* [29] is a service for performing symbolic indefinite integration. The architectural settings of the service is shown in Figure 3.2. The main part of it is a Java class (`SymbolicIntegrator`) which represents the actual implementation of the service. The service is deployed in the AXIS servlet running under the Tomcat Web server using a *Web Service Deployment Descriptor* (WSDD). This makes the service available on the Web where it can be accessed through its URL[1]

If the AXIS servlet is is contacted at the service URL extended by the string `?wsdl` it returns the WSDL description of the corresponding service.

[1] `http://perseus.risc.uni-linz.ac.at:8080/axis/services/SymbolicIntegration`

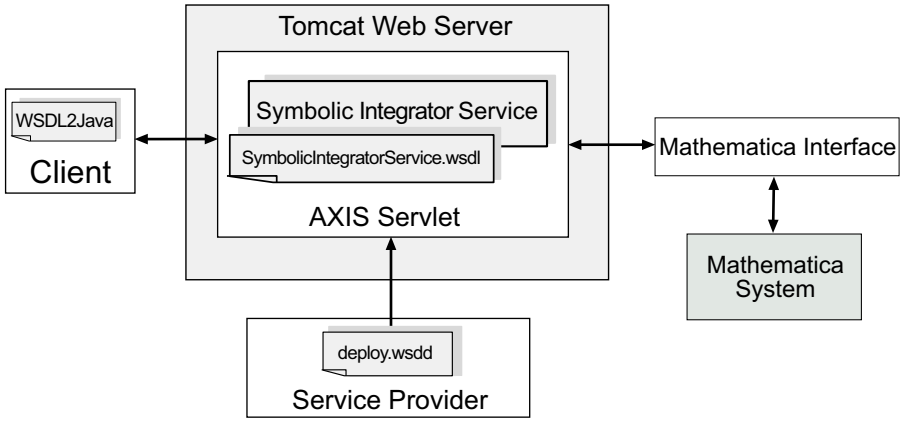


Figure 3.2: Architecture Realization: The Symbolic Integrator Service

This description includes the required information to access the service such as the protocol used to access the service, the operations that can be performed, and the input/output message format. The AXIS tool *WSDL2Java* is used to generate from the service WSDL description the client side code that is used to connect to and access the service.

The service accepts the client's integration expression in OpenMath format and then hands it to the Mathematica specific interface. The interface is the RIACA *Mathematica service* which uses a *Mathematica phrasebook* to convert this OpenMath expression to an expression in the Mathematica syntax and hands it to the Mathematica system. The Mathematica system performs the integration and returns the result to *Mathematica service* where the *Mathematica phrasebook* converts this result to OpenMath. The result in OpenMath is then returned to the service which returns it to the client.

The client for this service is implemented as a Web application. It includes A JSP (JavaServer Pages) GUI page where the user inputs the integration expression and gets the integration result displayed. The client parses the expression (using a parser devised for this purpose) and converts it to OpenMath. With this OpenMath expression as the argument, the client invokes the service via the previously (using *WSDL2Java*) client-side generated code. When it gets the result from the service, it displays it in MathML.

The Prover Service

The *prover service* is a service for performing reasoning using the Cooperating Validity Checker Lite (CVCL) [35] reasoner. The architectural setting of the service is shown in Figure 3.3. As with the *symbolic integrator service*, the *prover service* is deployed in the AXIS engine running under the Tomcat

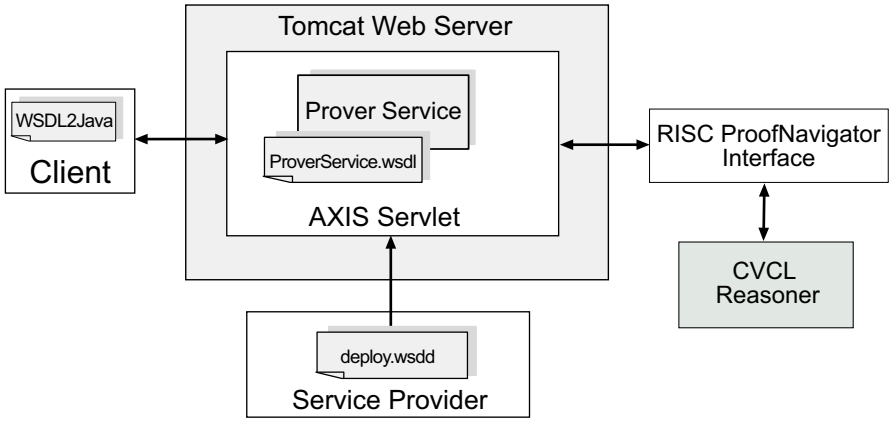


Figure 3.3: Architecture Realization: The Prover Service

Web server. Clients access it by the WSDL-based approach where the AXIS tool *WSDL2Java* is used to generate from the service WSDL description the client side code that is used to connect to and access the service. The RISC ProofNavigator library [114] is used by the service as the interface to the CVCL reasoner.

The service receives remote client requests that include three parameters: the formula to be proved as an OpenMath string, the declarations of variables used in the formula as an OMDoc string array, and the axioms used as the basis for proving the formula as an OpenMath string array. The service takes these parameters and invokes the CVCL via the RISC ProofNavigator interface. CVCL performs the required reasoning on the formula based on the given declarations and axioms. The Web service gets the answer from the CVCL reasoner through the RISC ProofNavigator interface and sends it to the client.

The client in our setting is the Mathematical Services Query Language (MSQL) engine that uses the *prover service* as one of two approaches to reason about its own semantic expressions (see Section 6). It invokes the service with the three parameters (formula, declarations, and axioms) using the client-side generated Java classes. It gets the result and returns it to the MSQL engine.

3.2 Service Description

Current Web service architectures use WSDL for describing the syntactic interfaces of services but lack means for describing their behavior. This is insufficient in semantically rich domains such as mathematics, where an exact description of the meaning of a service is crucial. Thus the MathBroker project has introduced the Mathematical Services Description Language

(MSDL) [30, 38] for describing mathematical Web services and their related objects. In this section we first present the information model underlying MSDL, then we sketch the MSDL structure, finally we show examples of service descriptions in MSDL.

3.2.1 The MSDL Information Model

The information model for describing mathematical Web services is based on a modular view of the various aspects of a mathematical service. It determines the type of entities and their related information that constitute a service description. Additionally it determines the kind of relationships and dependencies among these entities.

Figure 3.4 shows the kind of entities that can be combined to a service and the relationships among them.

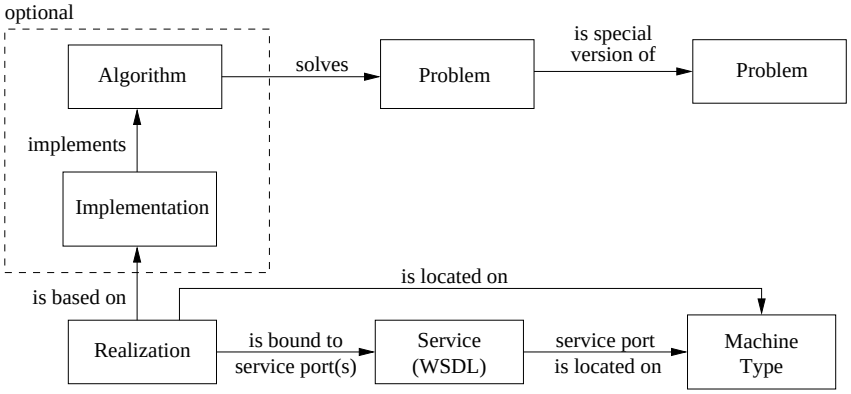


Figure 3.4: MathBroker Information Model

The following entities comprise the model:

Problem A problem can be specified by input parameters, an input condition, output parameters, and an output condition. Optionally, a problem can be declared as a special version of another problem (with a stronger input condition and/or a weaker output condition).

Algorithm An algorithm can be described by (a link to the description of) the problem it solves, by time and memory complexity, by its termination condition, etc.

Implementation An implementation can be described by (a link to the description of) the algorithm on which it is based (or alternatively by a link to the problem it solves), by the software used for the implementation, and by the time and memory efficiency of the implementation

(the constant factors to be associated to the algorithm's complexity) with respect to some reference architecture.

Realization A realization is described by the underlying software implementation (or alternatively the algorithm or problem), by the type of machine on which it is running, and by a WSDL description of the service interface.

Machine A machine can be described by the specifications of its hardware such as processor type, speed, memory size, etc.

Some of the important characteristics of the information model are:

- Problems are organized in a hierarchy where generalizations and specializations are taken into account.
- Algorithms are in relation to the problems they solve. Several algorithms may solve the same problem.
- Implementations are in relation to the algorithm they implement. Several programs may implement the same algorithm.
- Realizations are in relation to the implementation they are based on and to the service interface they are bound to. Additionally they might carry information about hardware details of the machine running the service. Several realizations may be based on the same implementation and several service interfaces may be bound to one realization.

3.2.2 The Mathematical Services Description Language

The Mathematical Services Description Language (MSDL) [30] is an implementation of the information model presented in the previous section. It is a highly structured language for describing the various entities of a service and the relationships among them. A description in MSDL is decomposed into multiple interlinked descriptions at the entity level. This makes these descriptions reusable. MSDL is flexible in the sense that a description of an entity can include any number of descriptive elements ranging between the minimum and the maximum number of elements as defined by the MSDL schema [92].

The MSDL schema is an XML schema defining the grammar and data types of the language. The current version of MSDL schema is an extension of the MONET schema [38] which in turn has been influenced by an earlier version of the MSDL schema. A skeleton of a service description in MSDL containing the above entities and their relationships is shown below. The description includes the entities: *machine_hardware*, *problem*, *algorithm*, *implementation*, and *realization* (service). The *implementation* entity includes,

e.g., a link to the *algorithm* on which it is based. The *realization* (service) entity includes a link to the *problem* it solves and a link to the underlying *implementation*. The Mathematical content in such descriptions is represented in OpenMath [27] (see the problem description in Section 3.2.3). The MSDL API [91] provides the needed functionality to process MSDL descriptions.

```
<monet:definitions>
  <mathb:machine_hardware name="perseus">
    ...
  </mathb:machine_hardware>
  <monet:problem name="integration">
    ...
  </monet:problem>
  <monet:algorithm name="RischAlg">
    ...
  </monet:algorithm>
  <monet:implementation name="RImpl">
    ...
   <monet:hardware href=".../perseus"/>
   <monet:algorithm href=".../RischAlg"/>
  </monet:implementation>
  <monet:service name="RRISC">
      ...
   <monet:problem href=".../integration"/>
   <monet:implementation href=".../RImpl"/>
  </monet:service>
</monet:definitions>
```

3.2.3 Example Service Descriptions

The Symbolic Integrator Service Description

We present the description for each entity of the *symbolic integrator service* separately, but they can be combined into one complete service description following the description skeleton shown above.

- The *problem* is described by the following parameters represented in OpenMath syntax.
  - Input: $f : \mathbb{R} \to \mathbb{R}$ (lines 4 to 14)
  - Output: $i : \mathbb{R} \to \mathbb{R}$ (lines 15 to 25)
  - Post-condition: $i = indefint(f)$ (lines 26 to 37)

```
1 <problem name="indefinite-integration">
```

```
 2  <header></header>
 3  <body>
 4    <input name="f">
 5      <signature>
 6        <OMOBJ>
 7          <OMA>
 8            <OMS cd="sts" name="mapsto"></OMS>
 9            <OMS cd="setname1" name="R"></OMS>
10            <OMS cd="setname1" name="R"></OMS>
11          </OMA>
12        </OMOBJ>
13      </signature>
14    </input>
15    <output name="i">
16      <signature>
17        <OMOBJ>
18          <OMA>
19            <OMS cd="sts" name="mapsto"></OMS>
20            <OMS cd="setname1" name="R"></OMS>
21            <OMS cd="setname1" name="R"></OMS>
22          </OMA>
23        </OMOBJ>
24      </signature>
25    </output>
26    <post-condition>
27      <OMOBJ>
28        <OMA>
29          <OMS cd="relation1" name="eq"></OMS>
30          <OMV name="i"></OMV>
31          <OMA>
32            <OMS cd="calculus1" name="indefint"></OMS>
33            <OMV name="f"></OMV>
34          </OMA>
35        </OMA>
36      </OMOBJ>
37    </post-condition>
38  </body>
39 </problem>
```

- The *algorithm* is described by the link to the problem it solves (lines 7 to 10) and by its bibliographical information (lines 11 to 26).

```
 1 <monet:algorithm name="RischAlg">
 2   <monet:documentation>
```

```
 3     This is the metadata for the algorithm Risch.
 4     The namespace is the target namespace of this
 5     document.
 6   </monet:documentation>
 7   <monet:problem
 8    href="http://risc.uni-linz.ac.at/mathbroker/
 9     RischIndefIntegration/indefinite-integration">
10   </monet:problem>
11   <monet:bibliography
12    href="http://www.emis.de/cgi-bin/zmen/ZMATH/en/
13     quick.html?type=xml&an=0184.06702">
14     <monet:documentation>
15       Dublin Core Data
16     </monet:documentation>
17     <dc:creator>Risch,R.H.</dc:creator>
18     <dc:title>
19      The Problem of Integration in Finite Terms
20     </dc:title>
21     <dc:source>
22      Trans. A.M.S. 139 pp.167 - 189
23     </dc:source>
24     <dc:publisher>AMS</dc:publisher>
25     <dc:date>1969</dc:date>
26   </monet:bibliography>
27 </monet:algorithm>
```

- The *implementation* description includes (links to) the algorithm on which it is based (lines 15 to 18), the software on which it is based (lines 6 to 10), and the time and memory efficiency (lines 2 to 5).

```
 1 <monet:implementation name="RImpl">
 2  <mathb:efficiency_factor>
 3    <mathb:speed>1.1</mathb:speed>
 4    <mathb:throughput>0.7</mathb:throughput>
 5  </mathb:efficiency_factor>
 6  <monet:software href="http://www.wolfram.com">
 7  </monet:software>
 8  <monet:software
 9   href="http://riaca.win.tue.nl/software/ROML">
10  </monet:software>
11  <monet:hardware
12   href="http://risc.uni-linz.ac.at/mathbroker/
13   RischIndefIntegration/perseus.risc.uni-linz.ac.at">
```

```
14  </monet:hardware>
15  <monet:algorithm
16   href="http://risc.uni-linz.ac.at/
17    mathbroker/RischIndefIntegration/RischAlg">
18  </monet:algorithm>
19 </monet:implementation>
```

- The *realization* is described by (links to) the software implementation on which it is based (lines 14 to 17) (and optionally by the problem it is based on lines 9 to 12), and the WSDL of the service (lines 18 to 21).

```
 1 <monet:service name="RRISC">
 2   <monet:documentation>
 3   This is an implementation of the algorithm Risch.
 4   We use the mathb namespace to state expected
 5   performance of the concrete implementation wrt to
 6   its theoretical complexity measure.
 7   </monet:documentation>
 8   <monet:classification>
 9     <monet:problem
10      href="http://risc.uni-linz.ac.at/mathbroker/
11       RischIndefIntegration/indefinite-integration">
12     </monet:problem>
13   </monet:classification>
14   <monet:implementation
15    href="http://risc.uni-linz.ac.at/mathbroker/
16     RischIndefIntegration/RImpl">
17   </monet:implementation>
18   <monet:service-interface-description
19    href="http://perseus.risc.uni-linz.ac.at:8080/axis/
20     services/SymbolicIntegration?wsdl">
21   </monet:service-interface-description>
22   <monet:service-binding>
23     <monet:map action="exec"
24      operation="symbint:Integrator:indefInt"
25      problem-reference="indefinite-integration">
26     </monet:map>
27     <monet:message-construction io-ref="f"
28       message-name="symbint:IndefIntRequest"
29      message-part="in0">
30     </monet:message-construction>
31   </monet:service-binding>
32   <monet:service-metadata></monet:service-metadata>
```

```
33   <monet:broker-interface>
34     <monet:service-URI></monet:service-URI>
35   </monet:broker-interface>
36 </monet:service>
```

- The *machine* is described by its processor type and speed (lines 3 to 4), by its memory size and disk size (lines 5 to 6), and by the type of operating system it uses (line 7).

```
1 <mathb:machine_hardware address="193.170.37.69"
2              name="perseus.risc.uni-linz.ac.at">
3   <mathb:CPU name="Intel Celeron"></mathb:CPU>
4   <mathb:CPU_speed mhz="733"></mathb:CPU_speed>
5   <mathb:RAMsize mb="256"></mathb:RAMsize>
6   <mathb:disksize gb="40"></mathb:disksize>
7   <mathb:OS href="http://www.suse.de"></mathb:OS>
8 </mathb:machine_hardware>
```

The Prover Service Description

The *prover service* is described by three entities: the implementation, the realization, and the machine. The description of each entity does not include all the possible information defined by MSDL.

- The *implementation* is described by the software on which it is based (lines 2 to 8) and the hardware on which it is based (lines 9 to 12).

```
 1 <monet:implementation name="ProverImpl">
 2  <monet:software
 3   href="http://chicory.stanford.edu/CVCL">
 4  </monet:software>
 5  <monet:software
 6   href="http://www/research/formal/software/
 7    ProofNavigator">
 8  </monet:software>
 9  <monet:hardware
10   href="http://risc.uni-linz.ac.at/mathbroker/
11    Prover/perseus.risc.uni-linz.ac.at">
12  </monet:hardware>
13 </monet:implementation>
```

- The *realization* is described by the underling software implementation (lines 5 to 8) and by the WSDL description of the service interface (lines 9 to 12).

```
 1 <monet:service name="Prover">
 2  <monet:documentation>
 3  This service performs reasoning on behalf of the
 4   MSQL engine using the CVCL reasoner.
 5  <monet:implementation
 6   href="http://risc.uni-linz.ac.at/mathbroker/
 7                     ProverService/Prover">
 8  </monet:implementation>
 9  <monet:service-interface-description
10   href="http://perseus.risc.uni-linz.ac.at:8080/axis/
11          services/Prover?wsdl">
12  </monet:service-interface-description>
13  <monet:service-binding>
14   <monet:map action="" operation=""
15    problem-reference="">
16   </monet:map>
17   <monet:message-construction io-ref=""
18     message-name="" message-part="">
19   </monet:message-construction>
20  </monet:service-binding>
21  <monet:service-metadata></monet:service-metadata>
22  <monet:broker-interface>
23   <monet:service-URI></monet:service-URI>
24  </monet:broker-interface>
25 </monet:service>
```

- The *machine* is described by its processor type and speed (lines 3 to 4), by its memory size and disk size (lines 5 to 6), and by the type of operating system it uses (line 7).

```
1 <mathb:machine_hardware address="193.170.37.69"
2              name="perseus.risc.uni-linz.ac.at">
3   <mathb:CPU name="Intel Celeron"></mathb:CPU>
4   <mathb:CPU_speed mhz="733"></mathb:CPU_speed>
5   <mathb:RAMsize mb="256"></mathb:RAMsize>
6   <mathb:disksize gb="40"></mathb:disksize>
7   <mathb:OS href="http://www.suse.de"></mathb:OS>
8 </mathb:machine_hardware>
```

3.3 Summary

Within the frame of the MathBroker project, we have designed and implemented the infrastructure for developing and describing mathematical Web services.

- The service development is concerned with the design and implementation of an architecture for mathematical services. The architecture conforms to current Web service specifications with established standards and technologies such as XML, SOAP, and WSDL. Additionally it employs specific standards for computer mathematics such as OpenMath and employs various kinds of mathematical computation and reasoning software. We have implemented a number of example services that make use of this architecture.

- The service description is concerned with the design and implementation of a language for describing the components and various aspects of mathematical services. The Mathematical Services Description Language (MSDL) is an XML schema based language that implements the information model of mathematical services. The information model consists of the entities that comprise individually or together a mathematical service. The entities defined by the information model are: *problem*, *algorithm*, *implementation*, *realization*, and *machine*. MSDL is designed to be a structured language that addresses the various aspects of a service and captures the semantics of the mathematical objects underlying such a service.

In the rest of this thesis, we present the framework we have developed for publishing mathematical services based on MSDL descriptions (Chapter 4) and for discovering these services based on their MSDL descriptions in a syntactical (Chapter 5) as well as a semantical (Chapter 6) manner.

Chapter 4

Service Publishing: The Registry Framework

A major goal for the future Web is to provide service publishing and discovery. Any operation that makes a service description available for a user can be viewed as service publication and any operation allowing access to the service description can be viewed as a service discovery. In Section 2.3 we presented three approaches for service publishing and discovery of which the registry approach is the most powerful one because it can encompass the other ones. A registry is a software application that enables the publication and discovery of Web services; it maintains Web service descriptions as objects in a repository and provides access to them via a specific protocol. Currently there are two dominating registry standards: the Universal Description, Discovery, and Integration (UDDI) registry [126]; and the ebXML registry standard [45] (see Section 2.3).

In this chapter, we develop a registry framework for publishing and discovering mathematical Web services. The framework is based on the ebXML registry standard [45] and its reference implementation [44]. The ebXML information model [43] is more generic and extensible than UDDI; it closely follows Sun's Java API for XML registries (JAXR) [64] which provides a generic access to a variety of XML registries. We extend the ebXML registry standard information model to accommodate the MSDL mathematical information model (see Section 3.2.1). We also extend the functionality of the ebXML Registry Reference implementation (ebxmlrr) [44] to incorporate the functionality needed to process, publish, update, retrieve, classify, associate, and query mathematical objects defined by the MSDL information model.

The chapter is organized as follows: Section 4.1 briefly describes the usage scenario of a registry. Section 4.2 presents the architecture, the information model, and the reference implementation of the ebXML registry. Section 4.3 presents the registry extension we have developed including the

MSDL information model in context of the ebXML registry information model, the architecture of the mathematical registry, and a prototype implementation. In Section 4.4, we present some examples demonstrating the use of the resulting registry for publishing and discovering MSDL service descriptions. Finally, Section 4.5 states the limitation of the querying approach of the registry and the way to overcome this limitation.

4.1 A Registry Usage Scenario

A registry provides a set of functionalities to facilitate the publishing and the discovery of Web services. Figure 4.1 shows how a registry can be used for publishing and discovering services.

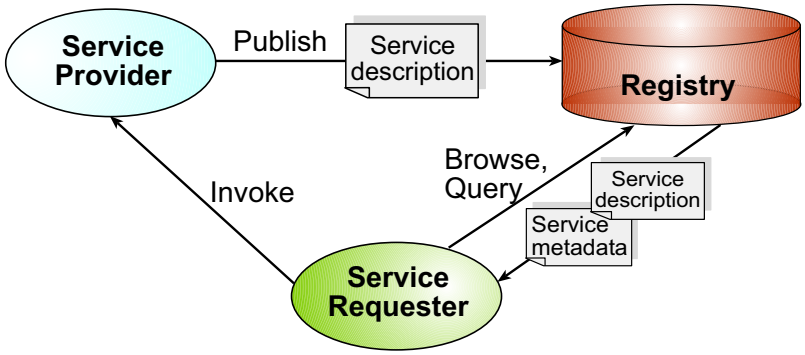


Figure 4.1: A Registry Usage Scenario

1. The service provider deploys a service to a Web server and publishes its description in MSDL to a registry. Two kinds of service related information are published: metadata which are pieces of information such as the name, association(s), and classification(s) that are stored in the database of the registry, and a description document which is stored in the repository of the registry.

2. The client (service requester) browses or queries the registry for desired services. It can access the service metadata and descriptions and based on the information contained on them decides on the suitable service.

3. The client accesses the suitable service from its location using information such as the URL of the service, and the number and kind of required parameters defined in its interface.

We aim to build a mathematical registry to publish mathematical service descriptions in MSDL and be able to discover them using a functionality

suited to the nature of these descriptions in addition to the functionality of the ebXML registry.

4.2 The ebXML Registry

In order to make this thesis self-contained, we describe in some detail the ebXML registry functionality and information model. We introduce those components and entities that are needed for our extensions presented in Section 4.3.

4.2.1 The Registry Architecture

The ebXML registry architecture, shown in Figure 4.2, consists of an ebXML *Registry Service* and an ebXML *Registry Client*. The ebXML *Registry Service* provides the ability for managing the registry and its repository. An ebXML *Registry Client* is an application used to access the registry.

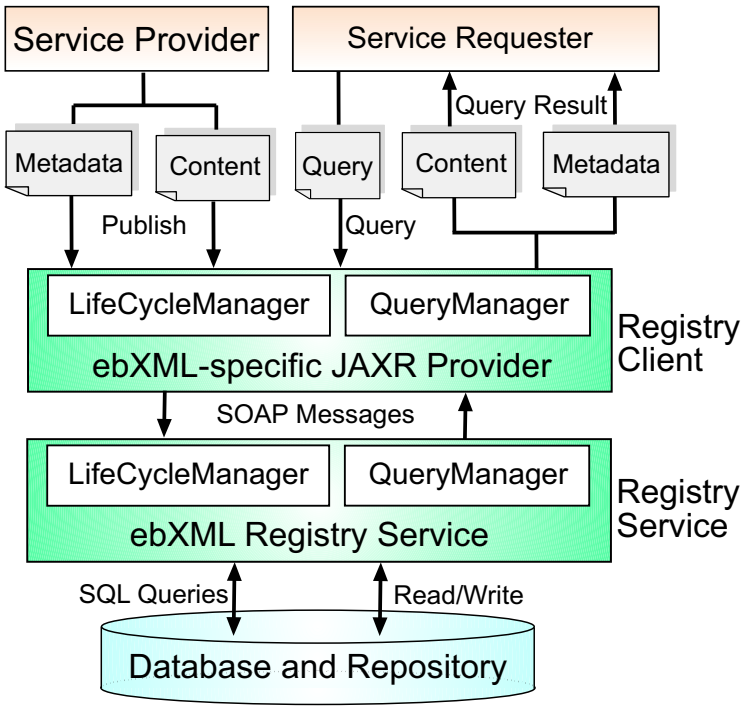


Figure 4.2: ebXML Registry Architecture

The Registry Service

The ebXML *Registry Service* fundamentally manages objects and queries associated with the ebXML registry. The two primary interfaces for the *Registry Service* are:

- The *LifeCycleManager* interface that provides a collection of functionalities for managing metadata and content related to objects within the registry. This includes publishing, updating, approval, and deletion of metadata and content of a registry object.

- The *QueryManager* interface that provides a collection of functionalities for the discovery and retrieval of metadata and content related to objects within the registry.

The Registry Client

A *Registry Client* utilizes the services of the registry by using the functionalities provided by the above interfaces. The client uses these functionalities to submit objects, to classify and associate objects, to remove objects, to browse objects, and to query for objects and their related repository items.

The *Registry Client* interfaces may be local to the registry or local to the user. In the first case the registry provides a Web based "thin client" application for accessing the registry that is available to the user using a common Web browser. In this scenario the *Registry Client* interfaces are accessed across the Internet and are local to the registry from the user's view. The client in this case uses the HTTP protocol to communicate with the registry.

In the second case the user uses a "fat client" registry application to access the registry. In this scenario the *Registry Client* interfaces are local to the user. The *Registry Client* interfaces communicate with the registry over the Internet using the SOAP protocol.

To enable clients to access registries in an implementation-independent way, the JAVA API for XML Registries (JAXR) [64] has been designed. The JAXR information model describes content and metadata within XML registries. The JAXR specification includes detailed bindings between the JAXR information model and the ebXML Registry. A *Registry Client* implementation of the JAXR API (which provides access to a specific registry or to a class of registries that are based on a common specification) is called a *registry-specific JAXR Provider*. The ebXML architecture presented here uses an *ebXML-specific JAXR Provider*. It implements the *Registry Client* side *LifCycleManager* and *QueryManager* interfaces corresponding to those of the *Registry Service*. It also implements the information model of the registry.

Database and Repository

The ebXML registry is both a registry of metadata and a repository of content. A typical ebXML registry implementation (see Section 4.2.4) uses a database to store its metadata and uses a file system (repository) to store content. Architecturally, the registry is different from the repository. However, all accesses to the registry as well as the repository are done through the operations defined by the *Registry Service* interfaces.

Service Provider

The *Service Provider* provides an implementation of the service and defines its description. It uses the ebXML-specific JAXR Provider to publish the description to the registry.

Service Requester

The *Service Requester* can be a software component that requests a service. It uses the ebXML-specific JAXR Provider to discover services in the registry.

4.2.2 The Registry Service Protocols

The ebXML registry standard defines a number of registry protocols supported by the *Registry Service* interfaces. These protocols provide the functionality required by registry clients to manage the lifecycle of registry objects and repository items as well as to query the registry for registry objects and repository items. Registry protocols supported by the *LifeCycleManager* interface include submit, update, remove, and approve protocols. Registry protocols supported by the *QueryManager* interface include SQL query and Filter query protocols.

The generic message exchange patterns that are common to all registry protocols consist of request and response messages. The *Registry Client* sends an element derived from *RegistryRequestType* based on the used protocol to a registry, and the registry generates an element derived from *RegistryResponseType*, as shown in Figure 4.3

A registry request is atomic. In the case of success, the registry sends a *RegistryResponse* with "Success" status back to the client. In the case of failure, the registry sends a *RegisryResponse* with "Failure" status back to the client.

Publishing in the ebXML Registry

Publishing to the registry is supported by the *LifeCycleManager* interface. The *LifeCycleManager* protocols provide the needed functionality for the

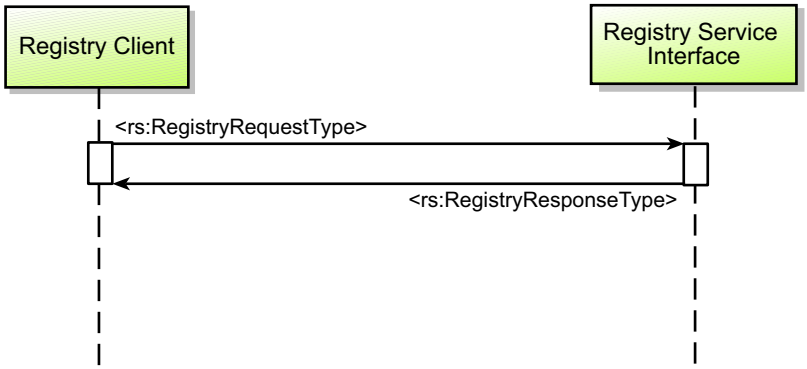


Figure 4.3: Registry Protocol Request-Response Pattern

Registry Client to manage the lifecycle of registry objects and repository items. Next we present one of these protocols, the *Submit Objects Protocol*. This Protocol allows a *Registry Client* to submit one or more objects and/or repository items. Figure 4.4 illustrates the actions taken by the *Registry Client* and the *LifeCycleManager* interface of the registry.

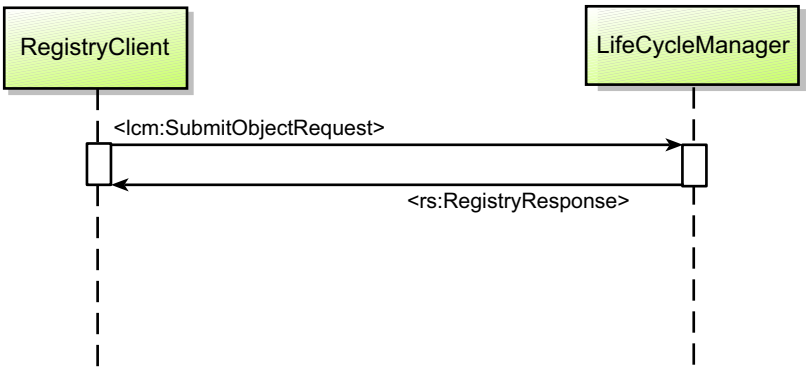


Figure 4.4: Submit Objects Protocol

The client uses the *SubmitObjectsRequest* to submit a registry object and/or repository items to the registry. The syntax of *SubmitObjectsRequest* is as follows:

```
<element name="SubmitObjectsRequest">
  <complexType>
    <complexContent>
      <extension base="rs:RegistryRequestType">
        <sequence>
          <element ref="rim:RegistryObjectList"/>
```

```
          </sequence>
        </extension>
      </complexContent>
    </complexType>
  </element>
```

The *RegistryObjectList* element specifies a collection of registry objects to be submitted to the registry. The registry objects in the list may be new objects or existing objects in the registry. In case of existing objects the registry must update the existing objects. A submitted registry object is usually assigned a unique id which is a valid Universally Unique Identifier (UUID) by the registry.

The other protocols supported by the *LifecycleManager* service (update, remove, approve, etc.) work in a similar manner as the submit protocol.

Discovery in the ebXML Registry

Discovery querying supported by the *QueryManager* interface. The *QueryManager* interface provides the *Ad Hoc Query Protocol* that allows a client to query the registry and retrieve registry objects and/or repository items. As shown in figure 4.5, a *Registry Client* submits an ad hoc query request (*AdhocQueryRequest*) which specifies a query in one of the query capabilities supported by the registry. The supported query capabilities are the Filter query capability and the SQL query capability. The QueryManager sends an *AdhocQueryResponse* back to the client as a response. This response returns a collection of objects that match the query.

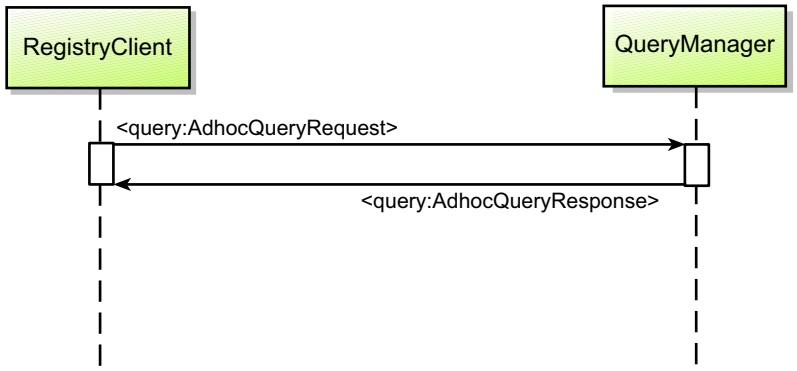


Figure 4.5: Ad Hoc Query Protocol

The *AdhocQueryRequest* syntax (as defined in the information model schema [42]) is as follows:

```
<element name="AdhocQueryRequest">
 <complexType>
  <complexContent>
   <extension base="rs:RegistryRequestType">
    <sequence>
     <element ref="query:ResponseOption"/>
     <choice>
      <element ref="query:FilterQuery"/>
      <element ref="query:SQLQuery"/>
     </choice>
    </sequence>
   </extension>
  </complexContent>
 </complexType>
</element>
```

The *FilterQuery* and *SQLQuery* parameters specify a registry filter query and a registry SQL query respectively. The required *ResponseOption* parameter allows the client to control content of the *AdhocQueryResponse*. The *AdhocQueryResponse* syntax is as follows:

```
<element name="AdhocQueryResponse">
 <complexType>
  <complexContent>
   <extension base="rs:RegistryResponseType">
     <choice>
      <element ref="query:FilterQueryResult"/>
      <element ref="query:SQLQueryResult"/>
     </choice>
    </extension>
   </complexContent>
  </complexType>
 </element>
```

The *FilterQueryResult* and *SQLQueryResult* parameters specify the result of a registry *Filter Query* and *SQL Query* respectively. Next we present these two supported query capabilities.

The Filter Query Capability

The *Filter Query* capability has an XML syntax that provides a simple query for the registry. It aims at reducing the amount of processing on the server side.

The *Filter Query* views the collection of registry objects in the registry as one virtual XML document and traverses a specified part of the document

tree and filters objects from it. The query returns a collection of objects that are left after filtering out all objects that do not match the filters specified in the query.

The SQL Query Capability

The optional *SQLQuery* element in the *AdhocQueryRequest* allows a client to submit complex SQL queries using the syntax for the *SQLQuery* of the registry as defined by subset of the SELECT statement as defined by [62] and [119]. The syntax of the registry query language is defined by a BNF grammar [45]. The result of a *SQL Query* is a collection of objects depending on the *ResponseOption* parameter specified by the client on the *AdHocQueryRequest*.

4.2.3 The Registry Information Model

The information model of the ebXML registry provides information on the type of metadata that is stored in the registry as well as the relationships among metadata classes. Figure 4.6 illustrates part of the ebXML registry information model. It shows the following object types that are concerned with metadata and content:

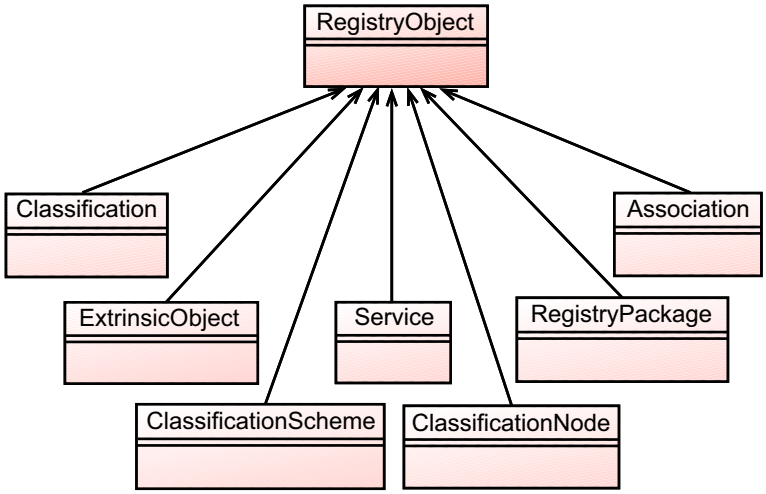


Figure 4.6: Part of the ebXML Information Model

RegistryObject: The *RegistryObject* class is an abstract base class used by most classes in the model. It provides minimal metadata for registry objects. It also provides methods for accessing related objects that provide additional metadata. *Slot* instances provide a dynamic way to add arbitrary attributes to *RegistryObject* instances.

Association: An *Association* class is a *RegistryObject* instance that is used to associate any two *RegistryObject* instances. An *Association* instance represents an association between a source *RegistryObject* and a target *RegistryObject* referred to as *sourceObject* and *targetObject*. An *Association* also has an *associationType* attribute that identifies its type.

Our extension to the ebXML information model includes the associations among mathematical objects defined by the MSDL information model.

ClassificationScheme: A *ClassificationScheme* instance describes a structured way to categorize *RegistryObject* instances. An example of a classification scheme (taxonomy) in mathematics is the Guide to Available Mathematical Software (GAMS).

Classification: A *Classification* instance is a *RegistryObject* instance that is used to classify other *RegistryObject* instances by referencing a node defined within a particular classification scheme.

ClassificationNode: A *ClassificationNode* instance defines a tree structure under a classification scheme. Each node in the tree is a classification node and the root is the classification scheme. A classification node has zero or one parent and zero or more immediate classification node children. There is a unique path identified leading from the classification scheme to a particular classification node.

ExtrinsicObject: An *ExtrinsicObject* instance provides metadata that describes submitted content whose type is not intrinsically known to the registry and therefore is described by additional attributes. Submitted content whose type is not intrinsically known to the registry is called repository item. Examples of repository items described by *ExtrinsicObject* include XML schemas, GIF images, and MSDL descriptions.

RegistryPackage: A *RegistryPackage* instance provides for grouping of logically related *RegistryObject* instances.

Service: A *Service* instance provides metadata about a service such as a Web service. A *Service* instance has a collection of bindings which are *ServiceBinding* instances that represent information on the way to access a specific interface provided by the *Service*.

Section 4.3.1 shows how the ebXML registry information model is extended to include the MSDL information model such that mathematical objects of the MSDL information model can be published and discovered in the registry.

4.2.4 The ebXML Registry Reference Implementation

The ebXML Registry Reference implementation (ebXMLrr) [44] is a Java based implementation of the ebXML registry service and information model specifications [46, 43]. It aims to provide a complete registry functionality as defined by these specifications. We based the implementation for the extension to the specifications on ebXMLrr version 3.0 (called OMAR: Object, Metadata and Artifacts Registry). It includes the following components:

- A server that implements the *Registry Service* interfaces and the information model as defined by the ebXML registry specifications [46, 43].
- An *ebXML-specific Provider* that implements the JAXR [64] specification.
- A Registry Browser Java (Fat client User Interface) application that is a JAXR client and that serves as a User Interface for the registry.
- A Registry Browser Web (Thin client User Interface) application that is a JAXR client and that serves as a User Interface for the registry.

4.3 The Mathematical Registry

In this section, we describe our extension of the ebXML registry information model [43] to include the MSDL information model (see Section 3.2.1). We describe the extension of the *Registry Service* [45] interfaces so that MSDL objects can be managed in the same way other ebXML registry objects are managed. We also describe the extension of the ebXML Registry Reference implementation (ebXMLrr) [44] which basically implements the registry information model [43], the *Registry Service* [45], and the *Registry Client*.

4.3.1 Extending the ebXML Information Model

We have exploited the extensibility feature provided by the ebXML registry information model to include the MSDL information model. One aspect of the extensibility lies in the *ExtrinsicObject* interface. The *ExtrinsicObject* interface allows to submit content whose type is not intrinsically known to the registry. Object types of the MSDL information model conform to this kind of content.

Figure 4.7 illustrates the MSDL information model in the context of the ebXML registry. It consists of the following objects:

MathBrokerObject: The *MathBrokerObject* class is an extension of the ebXML *ExtrinsicObject* class. It is used as a super class by the other MSDL information model interfaces and defines the functionality to process an MSDL object (for example extracting the different fields of

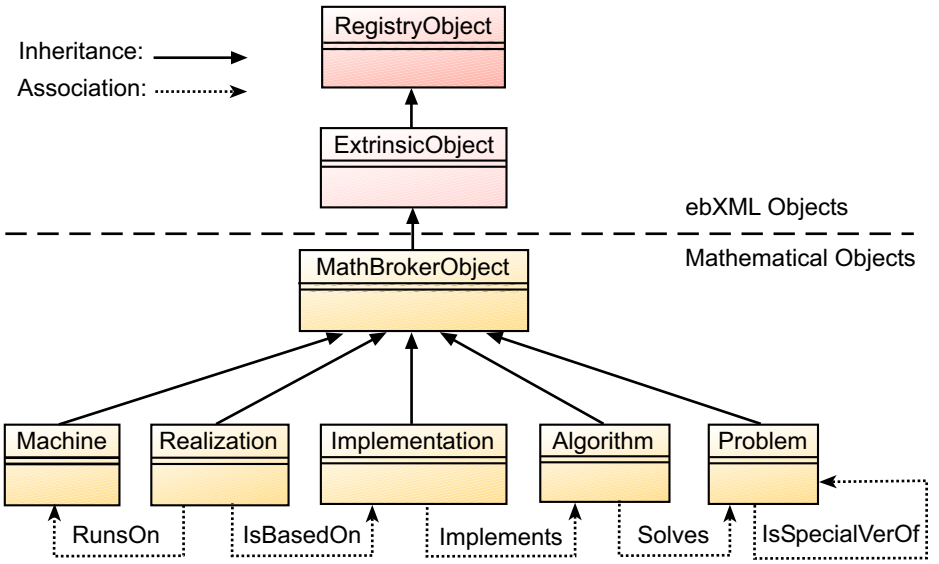


Figure 4.7: The Mathematical Services Information Model in the Context of the ebXML Information Model

the description of an object). This functionality as explained later is managed by the *MathBrokerLifecycleManager* interface.

Problem: A *Problem* instance provides information about an MSDL object of type problem whose description contains the detailed specification of the problem such as input/output parameters, pre/post conditions, associations to other objects, etc.

Algorithm: An *Algorithm* instance provides information about an MSDL object of type algorithm whose description contains the detailed specification of the algorithm such as its efficiency factors, associations to other objects, e.g., the problem it solves, etc.

Implementation: An *Implementation* instance provides information about an MSDL object of type implementation whose description contains the detailed specification of the implementation such as the algorithm it implements, its time and memory efficiency, associations to other objects, etc.

Realization: A *Realization* instance provides information about an MSDL object of type *Service* whose description contains the detailed specification of the service such as its underlying implementation, the WSDL description of its interface, etc.

Machine: An *Machine* instance provides information about an MSDL ob-

ject of type machine whose description contains the detailed specification of the machine such as its hardware characteristics.

The basic functionality of these components as registry objects is implemented as part of the mathematical registry API [11]. The rest of their functionality, such as get and set methods as defined by the MSDL schema are implemented in the MSDL Library API [91].

It is the responsibility of the *Mathematical Registry Provider* (see Section 4.3.5) to publish and discover mathematical objects having the above types and also handle them through their Lifecycle.

4.3.2 Associating Mathematical Objects

An essential characteristic of the MSDL information model is the ability to relate different objects that comprise a service to each other. Figure 4.7 also illustrates the relation among the entities of the information model. We modeled these relations as registry associations. We used the *Association* class of the ebXML information model to add them to the predefined registry associations. The MSDL information model in the context of the ebXML information model defines the following associations:

IsSpecialVersionOf: Problem P "is special version of" Problem P′.

Solves: Algorithm A "solves" Problem P.

Implements: Implementation I "implements" Algorithm A.

IsBasedOn: Realization R "is based on" Implementation I.

RunsOn: Implementation I "runs on" Machine M.

Figure 4.8 shows a more detailed view of the kind of associations that is used to associate two MSDL objects.

For every association there exists a source object and a target object. For example, in the association: Algorithm A "Solves" Problem P, the source object is Algorithm A and the target object is Problem P.

These associations are considered as registry objects and are defined to the registry like its predefined association types. In the implemented registry (see Section 4.3.6) they are published under the ebXML registry's *AssociationType* classification scheme.

4.3.3 Classifying Mathematical Objects

The ability to classify objects is one of the main features of the ebXML registry. This is because classifications facilitate the process of discovering objects within the registry. An object in the registry may be classified along multiple classifications. A *ClassificationScheme* instance in the

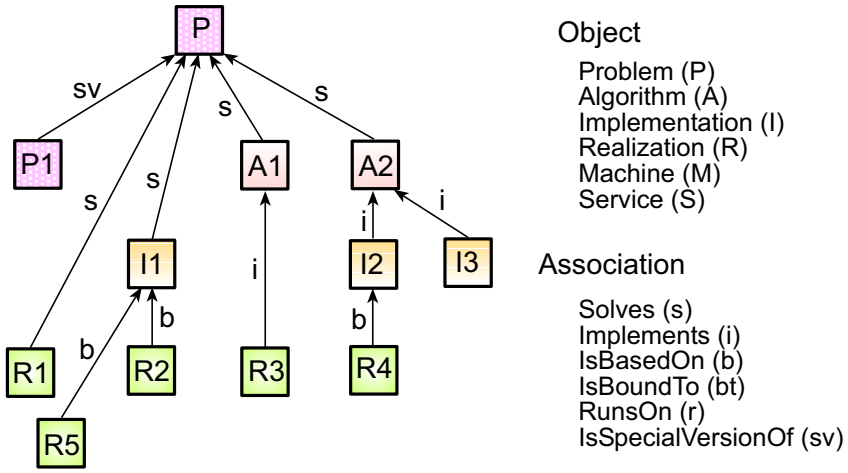


Figure 4.8: Mathematical Associations

registry information model represents taxonomies and *ClassificationNode* instances provide values to classify objects under these taxonomies. The ebXML registry provides the *SubmitObject* protocol to submit a taxonomy to the registry. Once a taxonomy is submitted to the registry, objects published can be classified against nodes under that taxonomy.

We exploited this feature to submit some classification schemes in the installed registry implementation (see Section 4.3.6). We submitted the Guide to Available Mathematical Software (GAMS) [51] classification scheme to the registry (see Section 4.4.2).

4.3.4 Extending the ebXML Registry Service

In section 4.2.1, we presented the two interfaces of the ebXML *Registry Service* that facilitate the management and querying of registry objects. We extended these two interfaces by the *MathBrokerLifeCycleManager* and *MathBrokerQueryManager* interfaces. These interfaces provide the functionality for managing the publishing and the discovery of MSDL objects in the registry. Section 4.3.5 presents more details on these two interfaces.

4.3.5 The Registry Architecture

In Section 4.6, we presented the extension to the ebXML registry information model. In this section we present the extension to the ebXML registry architecture. At the abstract level (where there is no distinction between the management interfaces of the *Registry Client* and that of the *Registry Service*) we extend the interfaces of the *Registry Service*. At the architectural level (as shown in Figure 4.9), we specifically extend the *ebXML-specific*

JAXR Provider. The architecture of the mathematical registry is shown in Figure 4.9. It consists of the following components:

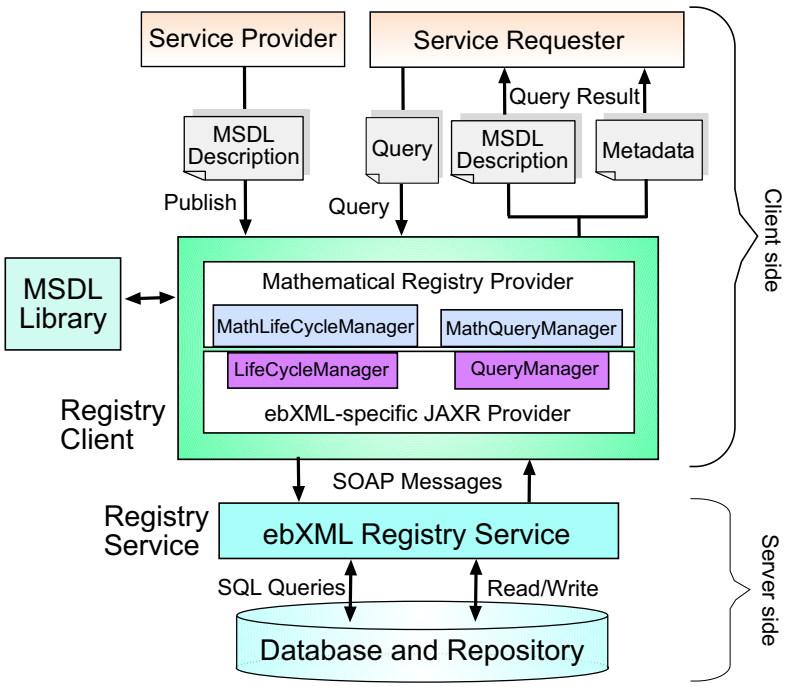


Figure 4.9: The Mathematical Registry Architecture

ebXML Registry Service

The ebXML *Registry Service* is the server-side component that manages registry objects and queries using the interfaces defined in Section 4.2.1.

ebXML-specific JAXR Provider

The *ebXML-specific JAXR Provider* implements the JAXR API specification in the ebXML registry-specific manner. The JAXR API (see Section 4.2.1) is a standard registry client API to access the registry. The ebXML-specific JAXR Provider has its own implementation of the ebXML *Registry Service* interfaces (the *LifeCycleManager* and the *QueryManager*).

Mathematical Registry Provider

Mathematical Registry Provider is an extension of the *ebXML-specific JAXR Provider*. It extends the *LifeCycleManager* and the *QueryManager* interfaces of the *ebXML-specific JAXR Provider* so MSDL objects respectively

their descriptions can be processed, published, and discovered in the registry. It mainly includes the *MathBrokerLifeCycleManager* and the *MathBrokerQueryManager* interfaces.

MathBrokerLifeCycleManager Interface

In order to publish mathematical objects respectively their MSDL descriptions to the registry and to manage them through their lifecycle, we extended the *LifeCycleManager* interface of the *ebXML-specific JAXR Provider* by the *MathBrokerLifeCycleManager* interface. When a service is published to the registry (i.e., its description is submitted to the registry), the *MathBrokerLifeCycleManager* performs the following steps:

- It processes the description of the service and determines the type of each entity contained in the description.
- It creates, for each entity, a registry object and extracts the corresponding metadata from the description to uniquely name the object in the registry (aside from the ID given to it by the registry). The unique name is formed by concatenating the name of the entity to the target name space of the description.
- It extracts, for each entity, the corresponding description as a separate repository item linked to the created registry object by means of a URI field in the object metadata.
- It creates the required associations and classifications as specified in the description.
- It stores the metadata of each created object in the database and the corresponding repository item in the *Repository*.
- If the object already exists in the registry, *MathBrokerLifeCycleManager* updates it by the new version.

The implementation of the *MathBrokerLifeCycleManager* interface uses the functionality provided by the *MSDL Library* API [91] (see below) to process MSDL descriptions in the above steps.

MathBrokerQueryManager Interface

The *MathBrokerQueryManager* interface (respectively its subinterface *MathBrokerFocusedQueryManager*) extends the ebXML *QueryManager* interface by the functionality to query the registry for mathematical objects, to retrieve their respective MSDL descriptions from the repository, and to extract

the required information from these descriptions. Inheriting the functionality and the query capabilities (see Section 4.2.2) of the ebXML *QueryManager* interface, it perform queries by object name, object type, association, classification, ID, or a combination of these criteria.

When an MSDL description is returned as the result of a query, the *MathBrokerQueryManager* can expose methods provided by the interface of the object type to which the description belongs to display the information contained in the description. To perform this, it uses the functionality provided by the *MSDL Library* API [91] (see below).

The query functionality is provided by the methods defined in the *MathBrokerFocusedQueryManager* subinterface of the *MathBrokerQueryManager* interface.

MSDL Library

The MSDL Library API [91] defines the functionality needed to process MSDL documents. It is based on the MSDL schema [92] where the grammar and data types of the language are defined. The API and its implementation are based on JAXB [63] which derives the MSDL interfaces and classes from the MSDL schema. The *Mathematical Registry Provider* uses the MSDL Library API to retrieve content from MSDL descriptions.

Service Provider

The *Service Provider* uses the *Mathematical Registry Provider* to publish a service description to the registry. It uses the functionality exposed by the *MathBrokerLifeCycleManager* interface to submit and manage service descriptions in the registry (see Section 4.4).

Service Requester

The *Service Requester* uses the *Mathematical Registry Provider* to discover services in the registry. It uses the functionality exposed by the *MathBrokerQueryManager* interface to query the registry for service descriptions. In Section 4.4, we present an example of service discovery where a client application is used to issue queries to the registry and get their results back. It also shows an example of using the ebXML registry browser to browse the mathematical objects in the registry.

Database and Repository

Each mathematical object in the registry is represented by metadata and content (see Section 4.2.1). Metadata are defined attributes of the object such as name, unique identifier, classification(s), association(s), etc. Content is the MSDL description of the object. The registry uses a rational database

system for storing metadata and a repository system for storing content. The repository is basically a file system managed by a server-side component called *RepositoryManager*. The content of an object in the repository system is known as a repository item. Each repository item is assigned a UID by the *LifeCycleManager* which servers as its name in the repository system.

4.3.6 The Registry Implementation

The implementation of the registry is basically the implementation of the MSDL information model and the *Mathematical Registry Provider* functionality centered in the *MathBrokerLifeCycleManager* and *MathBrokerQueryManager* interfaces. The implementation is based on the ebXML Registry Reference implementation (ebXMLrr) [44] (see Section 4.2.4). We extended the ebXMLrr implementation by the class implementations of the interfaces of the entities of the information model and the interfaces of the *Mathematical Registry Provider* in addition to other supporting classes. The extension resulted in the Mathematical Registry API [11].

The Mathematical Registry API consists of two parts:

- The interfaces defining the MSDL information model and the Mathematical Registry Provider.
- The classes implementing these interfaces.

4.4 Publishing and Discovering in the Registry

In this section we first present a sample installation of the registry. Then we show examples for publishing and discovering in the registry.

4.4.1 A Sample Installation of the Registry

The registry is installed according to the architecture (see Figure 4.9). The server side includes the ebXML *Registry Service*, the Database and Repository. The *Registry Service* is installed under the Apache Tomcat Web server [125]. The Database system used is the PostgreSQL [104] installed in a separate machine but the *Registry Service* is configured to access it. The repository is part of the file system under the directory of the *Registry Service* and is administered by a component of the *Registry Service*.

The client side includes *Registry Client* (*ebXML-specific JAXR Provider*, and *Mathematical Registry Provider* as two separate targets), and MSDL Library API. The *Registry Client* includes a registry browser application that can be used to manage and browse the registry (see Section 4.4.4). It also includes two client applications we developed to publish and query mathematical objects. Examples illustrating the use of these two applications and the registry are given in Section 4.4.3 and Section 4.4.4.

4.4.2 Submitting A Classification Scheme

We submitted the Guide to Available Mathematical Software (GAMS) [51] classification scheme to the registry. Figure 4.10 shows the scheme viewed in the registry browser.

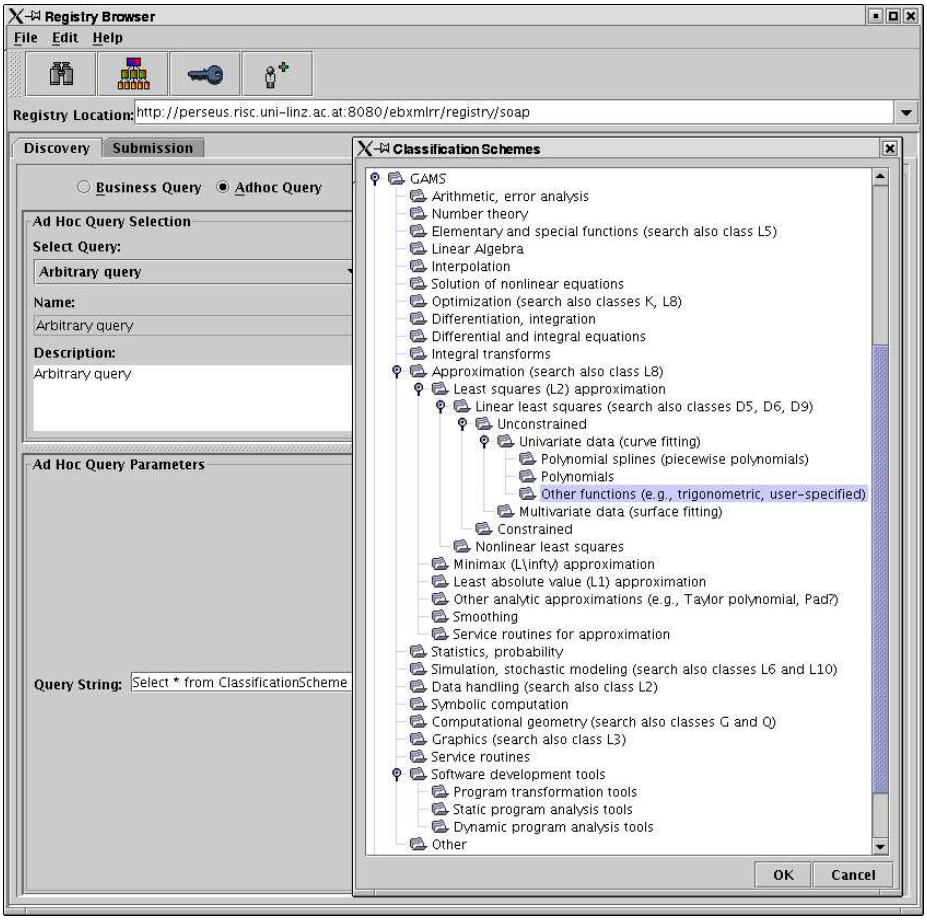


Figure 4.10: A Screenshot of the GAMS Classification Scheme

Originally the GAMS scheme was in the following generic XML format:

```
<gamslist>
  <title>GAMS classification</title>
  <gamslist>
    <gamscode>A</gamscode>
    <title>Arithmetic, error analysis</title>
    <gamslist>
      <gamscode>A1</gamscode>
      <title>Integer</title>
```

```
        </gamslist>
        <gamslist>
          <gamscode>A2</gamscode>
          <title>Rational</title>
        </gamslist>
        <gamslist>
          <gamscode>A3</gamscode>
          <title>Real</title>
          <gamslist>
            <gamscode>A3a</gamscode>
            <title>Standard precision</title>
          </gamslist>
          <gamslist>
            <gamscode>A3c</gamscode>
            <title>Extended precision</title>
          </gamslist>
          .
          .
          .
    </gamslist>
```

We wrote a XSLT [68] stylesheet (see Appendix A.1) to transform the scheme to the ebXML acceptable *SubmitObjectRequest* XML format shown below. Then we submitted it to the registry using the *SubmitObject* protocol mentioned above.

```
<?xml version="1.0" encoding="UTF-8"?> <rs:SubmitObjectsRequest
  xmlns = "urn:oasis:names:tc:ebxml-regrep:rim:xsd:2.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "urn:oasis:names:tc:ebxml-regrep:rim:xsd:2.0
  http://www.oasis-open.org/committees/regrep/documents/2.0/
         schema/rim.xsd
  urn:oasis:names:tc:ebxml-regrep:registry:xsd:2.0
  http://www.oasis-open.org/committees/regrep/documents/2.0/
         schema/rs.xsd"
  xmlns:rim = "urn:oasis:names:tc:ebxml-regrep:rim:xsd:2.0"
  xmlns:rs = "urn:oasis:names:tc:ebxml-regrep:registry:xsd:2.0">
 <LeafRegistryObjectList>
  <ClassificationScheme
    id="urn:uuid:62aebe64-866a-4706-982f-eec59a32e7ad"
    isInternal="true" nodeType="UniqueCode"
    xmlns="urn:oasis:names:tc:ebxml-regrep:rim:xsd:2.0">
    <Name>
        <LocalizedString charset="UTF-8" value="GAMS"/>
    </Name>
    <Description>
      <LocalizedString charset="UTF-8"
```

```
        value="This is the classification scheme for GAMS"/>
    </Description>
   <ClassificationNode
     id="urn:uuid:62aebe64-866a-4706-982f-eec59a32e7ad/A"
     code="A">
     <Name>
       <LocalizedString
         charset="UTF-8"
         value="Arithmetic, error analysis"/>
     </Name>
            .
            .
            .
   </ClassificationNode>
  </ClassificationScheme>
 </LeafRegistryObjectList>
</rs:SubmitObjectsRequest>
```

The *SubmitObjectRequest* includes a *RegistryObjectList* which contains the *ClassificationScheme* object to be submitted a long with a number of *ClassificationNodes* preserving the parent-child relation determined by the ID. For example the parent ID is "`urn:uuid:62aebe64-866a-4706-982f-eec59a32e7ad`" and its name is "`GAMS`", while the ID of its immediate child is "`urn:uuid:62aebe64-866a-4706-982f-eec59a32e7ad/A`" and its name is "`Arithmetic, error analysis`"

In addition to submitting the *GAMS* classification scheme, we also submitted the *MathBroker* classification scheme which includes the MSDL information model entities and associations as *ClassificationNodes*. Additionally we submitted the *CPU* classification scheme for classifying machines used by services according to the type of the CPU.

4.4.3 Publishing to the Registry

We wrote a sample client application that demonstrates publishing in the registry by submitting service descriptions in MSDL.

A Sample Service Description: An MSDL service description can contain one or more entities of the MSDL description. A complete service description would include all the entities introduced in Section 3.2.1 including associations and classifications as attribute nodes in these individual descriptions. A sample service description is shown in Appendix A.2.

The complete code of the client application, *MathBrokerPublish*, is shown in Appendix A.3.1. The statements below represent the important actions taken by the client application. It makes a connection to the registry using the URL of the registry (lines 6 to 10), uses this connection to obtain the *Registry Service* (line 12), and utilizes the service by accessing the *MathBrokerLifeCycleManager* and *MathBrokerFocusedQueryManager* interfaces (lines 13 and 14). It uses the *publishMathBrokerObject* method (line 16) of the *MathBrokerLifeCycleManager* interface to perform all actions required for the publishing process (see Section 4.3.5) such as extracting the description of each entity, creating a registry object embedding

that description, creating required associations and classifications as stated in the description.

```
1  MathBrokerRegistryService mrs = null;
2  MathBrokerConnection connection = null;
3  MathBrokerFocusedQueryManager fqm = null;
4  MathBrokerLifeCycleManager mlcm = null;
5  // make connection to the registry server
6  ConnectionFactory factory =
7             MathBrokerConnectionFactoryImpl.newInstance();
8  factory.setProperties( registryURL );
9  MathBrokerConnection connection =
10           (MathBrokerConnection)factory.createConnection();
11 //get the Registry Service and its two managers
12 mrs = connection.getMathBrokerRegistryService();
13 mlcm = mrs.getMathBrokerLifeCycleManager();
14 fqm = mrs.getMathBrokerFocusedQueryManager();
15 //do the publishing of the service description
16 mlcm.publishMathBrokerObject(fqm, description
```

Figure 4.11 illustrates the MSDL objects belonging to the published service in the registry.

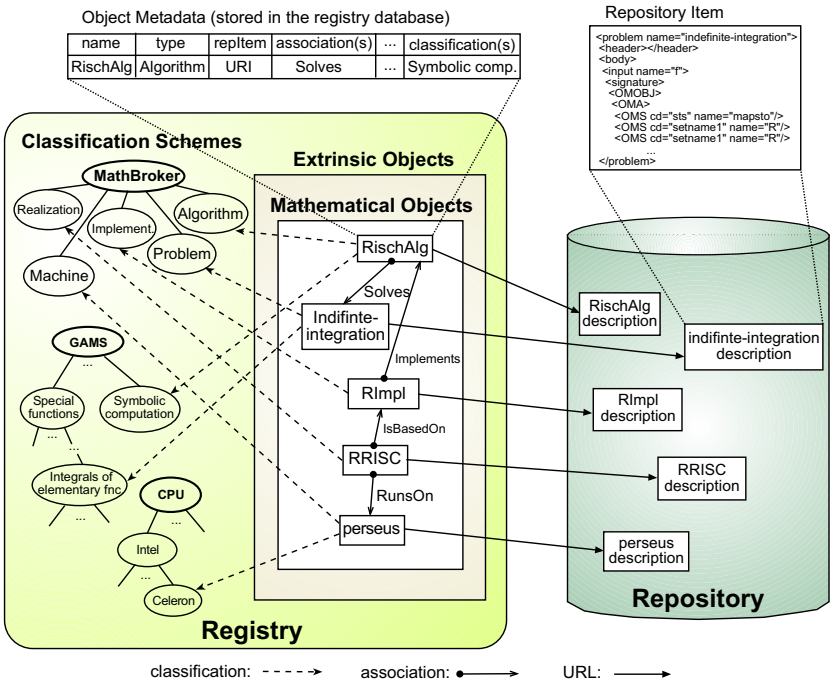


Figure 4.11: Mathematical Objects in the Registry

Each published object has its metadata stored in the database of the registry and its MSDL description stored in the repository as a repository item. One of

the metadata fields represents the URI address pointing to the object's description in the repository. The other fields represent information such as the name, the classification(s), the associations(s), etc. of the object. They are extracted from its description. Names of objects shown are actually prefixed with the corresponding namespace (see, e.g., Figure 4.12).

Objects are classified against classification nodes under the *MathBroker* classification scheme. Some of them are classified against other classification nodes in the *GAMS* classification scheme and the *CPU* classification scheme. For instance, the algorithm object "RischAlg" is classified against the "Algorithm" classification node and at the same time is classified against the "Symbolic computation" classification node.

Objects belonging to the service are associated to each other using the shown associations. For instance, the service Realization "RRISC" (the source object) is associated with the machine "perseus" (the target object) by the **RunsOn** association. We say Realization "RRISC" **RunsOn** on Machine "perseus".

4.4.4 Discovering in the Registry

We demonstrate discovering in the registry by means of a client application that performs several kinds of queries and by using the registry browser.

Querying

Depending on the kind of query the user wants to perform, the client application takes an argument from the user and prints the resulting MSDL descriptions or prints the information fields contained in these descriptions.

The complete code of the client application, *MathBrokerQuery*, is shown in Appendix A.3.2. The statements below represent the main steps taken by the client application. It makes a connection to the registry using the URL of the registry (lines 2 to 6), uses this connection to obtain the registry service (line 8), and utilizes the service by accessing the *MathBrokerLifeCycleManager* and *MathBrokerFocusedQueryManager* interfaces (lines 9 and 10). It uses the *executeQueryByClassification* method (line 12), one of several methods performing different kinds of queries, to perform a query for MSDL objects according to their classifications.

```
1  // Create the connection to the registry server
2  ConnectionFactory factory =
3          MathBrokerConnectionFactoryImpl.newInstance();
4  factory.setProperties(registryURL);
5  MathBrokerConnection connection =
6          (MathBrokerConnection)factory.createConnection();
7  //get the Registry Service and its two managers
8  rs = connection.getMathBrokerRegistryService();
9  mlcm = rs.getMathBrokerLifeCycleManager();
10 fqm = rs.getMathBrokerFocusedQueryManager();
11 //performquery according to classification
12 fqm.executeQueryByClassification(classificationArgument
```

The following methods defined by the *MathBrokerFocusedQueryManager* interface perform different kind of queries:

```
executeQueryById(argument)
executeQueryByName(argument)
executeQueryByClassification(argument)
executeQueryByClassificationConcept(argument)
executeQueryByClassificationConceptAndElementType(arg1, arg2)
```

Each of these methods calls a *showContent()* method specific to the type of the returned MSDL description (e.g. problem, machine, etc.) to displays individual fields from the description. It also shows the whole description in MSDL indented properly on the screen of the user.

The *MathBrokerFocusedQueryManager* interface defines similar methods that return only collections of MSDL object descriptions. They can be used by other applications that need to process the returned MSDL documents by their own. The Mathematical Services Query language (MSQL) engine (see Chapter 5) is an example of such an application.

Browsing Objects Using the ebXML Registry Browser

Figure 4.12 shows some MSDL objects in the registry browser. It shows the service object "RRISC", the objects related to it, their classifications, and their associations.

The name of an object is formed from the corresponding namespace and the element name as it appears in the MSDL description. For example, the namespace for Problem

> http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/indefinite-integration

is

> http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/

and its element name is "indefinite-integration".

The figure shows classifications as arrows labeled by the word "classifications". For example the object

> http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/RImpl

is classified against the "Implementation" classification node which is an indication of its type. The classification nodes shown belong to the MathBroker classification scheme.

Associations in the figure are shown as arrows starting from the source object and ending at the target object. They are interpreted as follows: Service

> http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/RRISC

Solves Problem

> http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/indefinite-integration

and **IsBasedOn** Implementation

> http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/RImpl

which **Implements** Algorithm

http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/RischAlg

and **RunsOn** Machine

http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/perseus.risc.uni-linz.ac.at

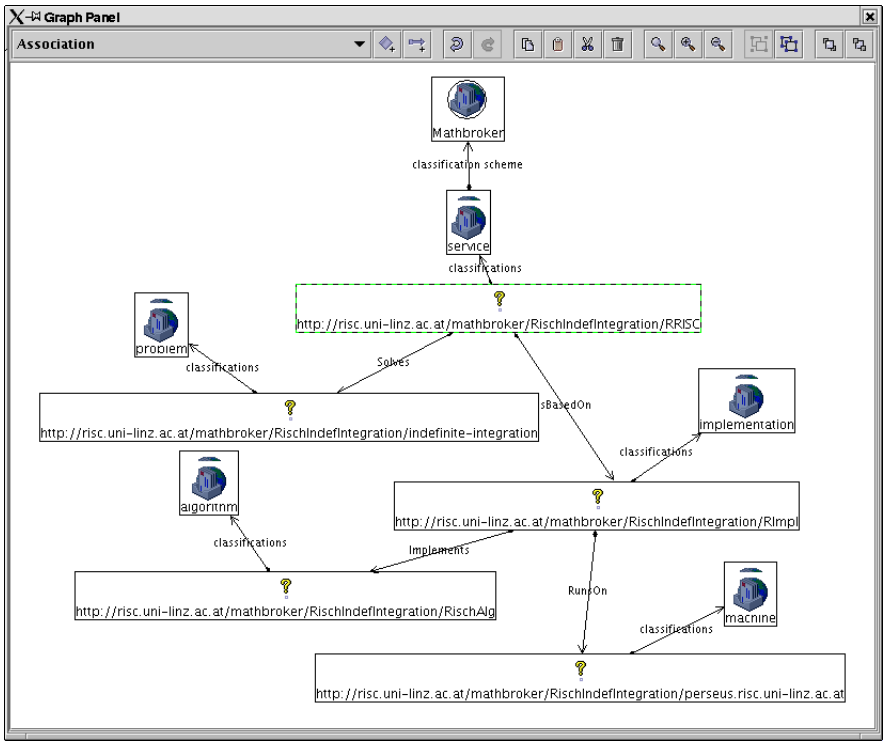


Figure 4.12: MSDL Objects with their Classifications and Associations

Asking for the details of an object shows its metadata. The metadata of the object

http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration/RRISC

is shown in Figure 4.13. Some of the metadata fields shown are the name, the unique identifier, the description, the classification, the type. Additionally asking for the repository item of an object shows its MSDL description.

4.5 The Querying Limitation of the Registry

Querying capabilities of the registry are rather limited, they only allow to query metadata accompanying the service when published in the registry. Although metadata such as the name, the unique identifier, the classifications, and the associations of a service are useful in some cases, e.g. when we are seeking a service based on its associations to other services, they are insufficient in many discovery cases involving

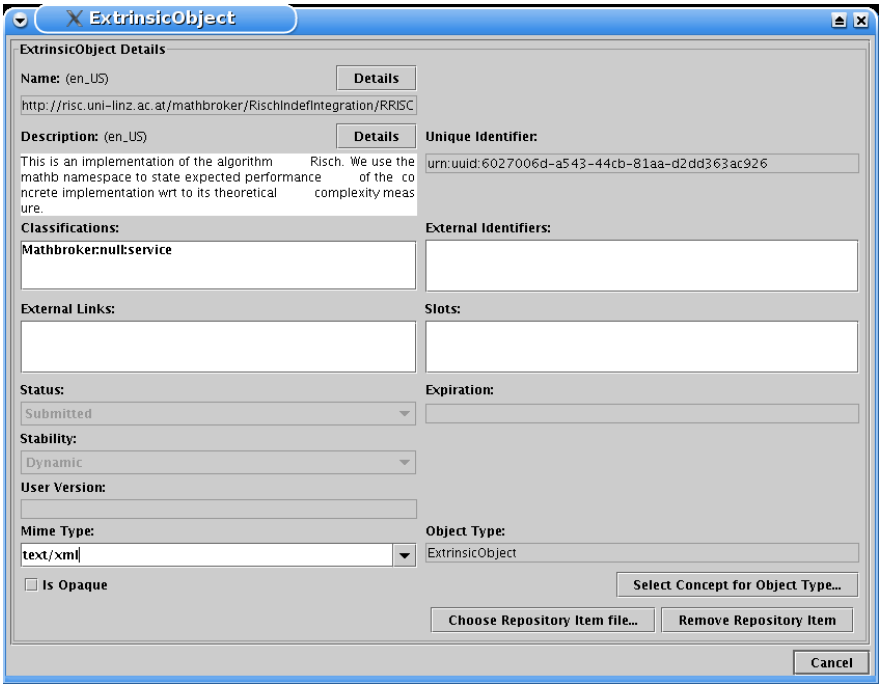


Figure 4.13: An MSDL Object Metadata

more information that is only found in the description of a service e.g., when we are seeking a service whose problem has a certain input precondition. Therefore it is necessary to resort to the contents of the MSDL description of a service where the complete information can be found.

To overcome the limitation of the current query capabilities of the registry, in Chapter 3 we develop the basis for a more advanced discovery framework. Namely, the Mathematical Services Query Language (MSQL) for querying registry published mathematical objects based on their contents.

4.6 Summary

In this chapter, we developed a registry framework where mathematical Web services can be published and discovered. The framework is based on the ebXML registry standard and its reference implementation. We extended the ebXML registry information model to include the MSDL information model and extended the ebXML *Registry Service* management interfaces to include mathematical object management interfaces. These extensions are implemented on top of the ebXML Registry Reference implementation (ebXMLrr).

The ebXML information model is extensible and this feature lies in the *ExtrinsicObject* where it allows external object types to be included in the information model. We included the entities of the MSDL information model as extension of the *ExtrinsicObject*.

The ebXML *Registry Service* specification specifies two main interfaces for registry objects management; the *LifeCycleManager* for managing registry objects through their lifecycle and the *QueryManager* for performing queries on registry objects. We extended these two interface by two interface counterparts to allow the registry to manage and query MSDL objects in the registry.

The implementation of the MSDL information model and that of the management interfaces constitute the *Mathematical Registry Provider* which is added as a layer to the ebXML Registry Reference implementation (ebXMLrr). It provides a set of functionalities for processing, classifying, associating, publishing, and querying mathematical services and their MSDL descriptions in the registry. We demonstrated this implementation by client applications for publishing and querying mathematical objects in the registry.

The resulting registry framework demonstrate the fact that standards and technologies developed for a particular application area (such as eBusiness) can be used in a more sophisticated application area such as computer mathematics.

This framework serves as the foundation for our next goal (see Chapter 5) to enhance the discovery mechanism of the registry by designing and implementing a query language for querying registry objects based not only on their metadata but also based on their contents.

Chapter 5

Service Discovery I: The Mathematical Services Query Language Framework

A registry published mathematical object is known by two pieces of data. The first is its metadata such as the name, the unique identifier, classifications, and associations. The second is its content, i.e., its MSDL description saved in the repository of the registry as an XML document.

Based on these kinds of information, two kinds of querying can be performed:

- **Metadata-based querying** which involves the metadata fields of the object. Queries are performed, e.g., by name, by unique identifier, or by classification depending on the available metadata of the object. This kind of querying is supported by the query capabilities of the registry (see Chapter 4).

- **Content-based querying** which involves the information contained in the MSDL descriptions stored in the repository. This information usually describes all aspects of the object and is more comprehensive than its registry published metadata. This kind of querying also depends on the first kind of data because in order to access the stored description we need to perform a metadata-based querying for the objects owning these descriptions, e.g., querying the registry for descriptions of a given object (entity) type classified under a given classification concept. The registry framework (see Chapter 4) itself does not support content-based querying.

Our goal in this chapter is to develop a query language that performs content-based querying on top of metadata-based querying.

The rest of the chapter deals with the design, formalization, architecture and implementation of the Mathematical Services Query Language (MSQL). Section 5.1 presents the design of the language stating its design goals, its structure, and some use cases. Section 5.2 describes the abstract language and defines its formal semantics. Section 5.3 presents the architecture and the implementation of MSQL. Section 5.4 describes the use of the MSQL API with some examples for supporting such queries.

5.1 Design of MSQL

In Section 2.3.2, we have briefly presented XQuery [22]; a general purpose language for querying collections of XML documents. Unlike XQuery, MSQL is not intended to be a general purpose XML query language. It is a domain-specific querying language that addresses the discovery needs of the presented mathematical framework. Although XQuery has a rich functionality, it cannot address some of our requirements such as dealing with classification schemes and types of objects stored in the registry. So, instead of using an XQuery based query engine that might not be compatible with our semantic extensions (see Chapter 6) and induce extra overhead, we designed our own query language maintaining a uniform framework. However, MSQL has some features similar to those of XQuery and of its predecessors, e.g., like XQuery (respectively XPath [20]), MSQL has the syntax for navigating in the hierarchical structure of MSDL documents. Furthermore, like SQL (Structured Query Language) [119] , MSQL utilizes the idea of a series of clauses based on keywords that provide a query pattern (the select-from-where pattern).

Based on the need to query the contents of MSDL documents published in the registry, we develop in this section the language building blocks stating its design goals and the constructs addressing these goals.

5.1.1 Motivation

Given a set of MSDL documents (such as those presented in Section 3.2) published in the registry, we would like to perform for example the following query:

> Find all problems under classification concept "/GAMS/Symbolic Computation" whose first input argument has type "integer".

To accomplish this request, we would do the following:

- consider the classification "/GAMS/Symbolic Computation" and fetch each document with entity type "problem" beneath it;
- process each document and return it if it satisfies the following criteria: *the first "input" argument occurring in the "problem" node is of type "integer".*

The returned documents may need to be sorted before being returned as a result.

Performing the first step involves contacting the registry and fetching the candidate documents from it. Performing the second step involves processing the returned candidate documents to see if they satisfy the stated criteria. It also involves formulating the criteria in a format that is suitable for the contents of the MSDL documents. Our objective is develop a query language to accomplish these steps. We next state them as requirements for the language and formulate its syntax based on these requirements.

5.1.2 Requirements of MSQL

We state a minimal set of requirements for a mathematical query language that stems from its anticipated use as a query language for MSDL documents published in the mathematical registry. These requirements are the following:

- **Precise semantics:** The language should have a formal semantics to make the intended meaning unambiguous. This semantics should also provide a straight forward reference for evaluating the correctness of the implementation.
- **Functional:** The language should specify what is to be done; how is it done is left to the implementation. It should support different kinds of expressions and functions.
- **Registry interfacing:** The language should have the necessary constructs to access the registry and use its management and query functionality in order to find and retrieve the candidate collection of MSDL documents.
- **Composing a query:** The language queries should be composed in a concise human-readable query syntax.
- **Query operations:** The operations that have to be supported by MSQL are:
  - **Retrieval:** Retrieving an MSDL document based on the type of entity it describes and on the registry classification concepts under which this entity is classified.
  - **Selection:** Choosing an MSDL document from the candidate documents based on content, structure, or attributes.
  - **Evaluation:** Ultimately selecting a document based on the evaluation of a semantic condition expressed by a logical formula.
- **Input and output:** The input to a query should be MSDL documents satisfying the "Retrieval" requirement. The output of a query are MSDL documents.
- **Result views:** The language should support ordered and unordered views of query results.

It is not required that the language is capable of restructuring a document or returning portions of a document as a result. This is not a disadvantage because our goal is to return whole documents that can be further processed by other applications.

Before presenting the design of the language constructs, we present the underlying data model.

5.1.3 The Data Model

As stated in the requirements, the input and output to an MSQL query are MSDL documents. In Section 3.2 we presented the MSDL information model with sample descriptions. An MSDL description is basically a schema-based XML representation stored in the registry as an XML document. This document is modeled as a tree of nodes where each node may have a sequence of child nodes. A child node can be an element node, an attribute node, or a text node. Attribute and text nodes are always leaf nodes.

In the MSDL fragment below (also illustrated in Figure 5.1), `problem` is the root node, `OMOBJ` is a child node with `problem` as its parent (assuming the "..." represent siblings of `OMOBJ`). The `OMS` node (on line number 5) has an `OMA` node as its parent

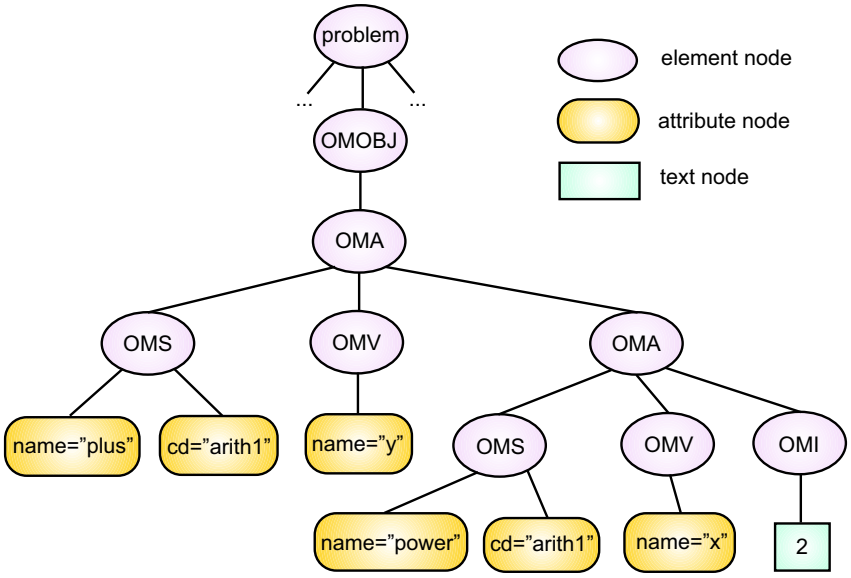


Figure 5.1: An MSDL Document Data Model

and has two attribute nodes as its children; `cd=‘‘arith1’’` and `name=‘‘plus’’`. The `OMI` element node (on line number 9) has the text node `2` as its child.

```
 1 <problem>
 2  ...
 3  <OMOBJ>
 4   <OMA>
 5     <OMS cd="arith1" name="plus"/>
 6     <OMA>
 7       <OMS cd="arith1" name="power"/>
 8       <OMV name="x"/>
 9       <OMI>2</OMI>
10     </OMA>
11     <OMV name="y"/>
12   </OMA>
13 </OMOBJ>
14 ...
15 </problem>
```

5.1.4 The Query Structure

The grammar of MSQL is shown in Appendix B.1. A query conforming to this grammar has the following structure:

```
select every | some <entity>
(from <classificationConcept>)?
(where <expression>)?
(orderby <expression> ascending | descending)?
```

The query has four main clauses: the *select* clause, the *from* clause, the *where* clause, and the *orderby* clause. The *from* clause and part of the *select* clause, namely `entity`, are registry-oriented, i.e., their functionality is applied to the registry. This satisfies the "Registry interfacing" requirement by determining the type of document, its classification in the registry, and retrieving it from the registry. This is a crucial issue of the language from the point of efficiency role since it limits the range of documents to be queried to those who are of type `entity` and classified under `classificationConcept`. The `entity` types as stated in the MSDL information model are *problem*, *algorithm*, *implementation*, *realization*, and *machine*. `ClassificationConcept` is a node in a given classification taxonomy of the registry, e.g., "/GAMS/Symbolic Computation" in the GAMS classification of mathematical subjects. The *select* clause also determines whether to return some or all of the resulting documents to the user by its *some* or *every* clause.

The *from clause* is optional and has three alternatives: The first is when the *from clause* is absent from the query, then the default "/Mathbroker/Entities" classification node followed by the `entity` type of the *select clause* is considered as the `classificationConcept`. The second is using a `classificationConcept` path value, e.g., "/urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/A/A1". The third is using a `classificationConcept` path name, e.g., "/GAMS/Arithmetic, error analysis/Integer".

The *where* and the *orderby* clauses apply their `expression` parts to each candidate document retrieved from the registry. The expression of the *where* clause is a logical condition: if it is evaluated to true, the document is considered as (part of) the result of the query. The *orderby* clause sorts the resulting documents in *ascending* or *descending* order based on comparison criteria resulting from the evaluation of its expression on each document.

The *where clause* is optional. If it is absent, then no expression is evaluated on the retrieved documents, i.e., no documents filtering will take place and all retrieved documents are returned as a result. The *orderby clause* is also optional and if it is absent, then no ordering on the resulting documents will take place.

We illustrate the structure described so far by a concrete query use case.

Use Case 1. *Find all problems under the classification concept "/GAMS/Symbolic Computation" with first input having type integer and order them according to their names in descending order.*

```
select every problem
from /GAMS/"Symbolic Computation"
where //body/input[1]/signature/OMOBJ/
       OMS[1][(( @name = "Z" )
        and ( @cd = "setname1" ))]
orderby /problem/@name descending
```

This query asks for *every* "problem", i.e., every document of type "problem" classified under "/GAMS/Symbolic Computation" that satisfies the *where* expression. The resulting documents are to be sorted in descending order according to their names. The core of the query is its *where* expression which allows us to express the first input and check its type. The structure of such expressions is explained in the following subsection.

5.1.5 MSQL Expressions

Since we are designing a light-weight query language, we have specified a minimal set of expressions that are necessary to address the contents of the target MSDL documents. MSQL expressions include: path expressions that can access every part of an MSDL document; expressions involving logical, arithmetic, and comparative operators; conditional expressions; quantified expressions; functions; and variable bindings. They satisfy the "Functional" requirement of the language. The various kinds of expressions are described below.

Path Expressions

A path expression is basically a special XPath [20] expression which locates nodes within a document tree (see Section 5.1.3.) Path expressions transform transforming a sequence of nodes of a given document to another sequence of nodes by navigating through the given hierarchical structure.

A path expression in MSQL takes the following form:

```
<pathExpr>::= ((/ | //) <step> | . | ([<expression>])*)+
<step>::= <string> | @<string>
```

It consists of a series of one or more "steps", separated by "/" or "//", and optionally starting with "/" or "//". The result of each step is a set of nodes in the document.

A "/" at the start of a path expression denotes the root of the current document (sub)tree (i.e., the document root itself, if the path expression occurs at the top-level of the query expression).

A "//" at the start of a path expression denotes every (descendent) node in a subtree of the current document root that is named as indicated by the step immediately following. This means that the entire root node tree hierarchy is to be navigated with every node in turn acting as the current node. Every subsequent step in a path expression takes the set of nodes constructed from the predecessor step and constructs a new set of nodes.

We are now going to present some examples, based on the MSDL document shown below:

```
 1 <problem name="indefinite-integration">
 2   <header></header>
 3   <body>
 4     <input name="f">
 5       <signature>
 6         <OMOBJ>
 7           <OMA>
 8             <OMS cd="sts" name="mapsto"></OMS>
 9             <OMS cd="setname1" name="R"></OMS>
10             <OMS cd="setname1" name="R"></OMS>
11           </OMA>
12         </OMOBJ>
13       </signature>
14     </input>
```

```
15      <output name="i">
16        <signature>
17          <OMOBJ>
18            <OMA>
19              <OMS cd="sts" name="mapsto"></OMS>
20              <OMS cd="setname1" name="R"></OMS>
21              <OMS cd="setname1" name="R"></OMS>
22            </OMA>
23          </OMOBJ>
24        </signature>
25      </output>
26      <post-condition>
27        <OMOBJ>
28          <OMA>
29            <OMS cd="relation1" name="eq"></OMS>
30            <OMV name="i"></OMV>
31            <OMA>
32              <OMS cd="calculus1" name="indefint"></OMS>
33              <OMV name="f"></OMV>
34            </OMA>
35          </OMA>
36        </OMOBJ>
37      </post-condition>
38    </body>
39  </problem>
```

- The expression

  `/problem/body/input`

  selects the `input` node (i.e., the `input` node from lines 4 to 14) as follows: The first "/" establishes the root node (i.e., the document node) as the current node for the step immediately following. The result of this step is the `problem` node. Similarly, the "body" step selects the `body` node that is the child of the `problem` node. The same applies to the "input" step which results in selection of the `input` node which is the child of the `body` node.

- The expression

  `//input`

  returns the same result (the `input` node) as the previous example. It selects it as follows: the "//" establishes any (descendant) `input` child node of the root node for the step "input".

- The expression

  `//OMA`

  selects all `OMA` (descendent) nodes of the root node (i.e., the `OMA` node from lines 7 to 11), the `OMA` node from lines 18 to 22, and the `OMA` node from lines 28 to 35).

- The expression

`//OMA/OMV`

selects all `OMV` nodes that are children of an `OMA` node (the `OMV` node on line 30 and the `OMV` node on line 33).

A ".", i.e., "self", denotes the current node itself. For example:

- If the current node is problem, then the expression

  .

  selects the problem node.

Attributes are specified with a preceding `@` (e.g. `@String`). For example:

- The attribute in the expression

  `/problem/@name`

  selects the name attribute of the `problem` node.

The resulting document set may be filtered by a predicate which consists of an expression enclosed in square brackets `[ ]`. The predicate expression can either be logical which evaluates to a truth value or numeric which evaluates to a numeric value. The following examples illustrates both of them:

- The expression

  `/problem/body/input//OMA/OMS[@cd = ‘‘sts’’]`

  has the logical predicate `[@cd = ‘‘sts’’]`. It selects only the first (on line 8) `OMS` node with the `cd` attribute value "sts" among the four available `OMS` child nodes of node `OMA`.

- The expression

  `/problem/body/input//OMA/OMS[1]`

  has the numeric predicate `[1]`. It selects the first `OMS` node (on line 8) from the four available child nodes of the `OMA` node.

- The expression

  ```
  /problem/body/input//OMA/OMS[3][(@name = ‘‘R’’) and
                                  (@cd =‘‘setname1’’)]
  ```

  uses both a logical and a numeric predicate. The first predicate is numeric and it specifies the third `OMS` child node of the `OMA` node. The second one is logical and uses the `=` and the `and` operators to evaluate to a truth value. It operates on the `OMS` node to check if its `name` attribute has value equal to "R" and its `cd` attribute has value equal to "setname1". The value of the second predicate represents the value of the expression; if its value is `true` then the third `OMS` node is selected.

Note that in complete MSQL queries the results of such path expressions are only used in the context of `where` expressions to select result documents from a candidate set of documents; the result of an MSQL query always consists of full documents.

Operators

MSQL supports arithmetic, logical, and comparison operators. They are used inside predicates to perform arithmetic, logical, and comparison operations. Arithmetic operators include `+`, `-`, and `*`. Logical operators include `or`, `and`, and `not`. Comparison operators include `=`, `!=`, `<`, `<=`, `>`, and `>=`. In Use Case 1, the expression `(@name = ‘‘Z’’) and (@cd = ‘‘setname1’’)` in the last predicate uses the comparison operator = `and` the logical operator `and` to evaluate to a truth value. Operator `=` in the subexpression `(@name = ‘‘Z’’)` checks if the value of the attribute `name` equals to string “Z”. Also operator `=` in the second subexpression `(@cd = ‘‘setname1’’)` checks if the value of the attribute `cd` equals to string “setname1”. The operator `and` in the expression is used for the logical operation between the results of the two subexpressions.

Functions

In the context of a predicate, functions may be applied to the current node to extract information used in some operations. MSQL supports the functions stated in the rule.

```
<msqlFunction> ::= (empty | counts | contains | doc |
                    position)
                  ‘‘(’’(<expression> | <expression>,
                                      <expression>)?‘‘)’’
```

- Function `empty()` returns *true* if its argument is an empty element.
- Function `count()` returns an integer value equivalent to the number of occurrences of its argument in the current node.
- Function `contains()` returns *true* if its first argument value contains as part of it its second argument.
- Function `doc()` returns the root node of the document whose name appear as its argument.
- Function `position()` returns the position (integer value) of the element, whose name precedes it in a path expression, in the child list of the current node.

Use Case 2. *Find all problems in “/GAMS/Arithmetic, error analysis/Integer” that have no precondition.*

```
select every problem
from /GAMS/"Arithmetic, error analysis"/Integer
where //body[empty(/pre-condition)]
```

In this Use Case, the `empty()` function takes the `/pre-condition` path expression as its argument and returns *true* if the node denoted by the path expression is empty. The query uses the `empty()` function to check if the “body” node has an empty “pre-condition” element node.

In Use Case 3 (shown below), the `contains()` function returns *true* if its first argument value (`@href` attribute value) contains as part of it its second argument

(string "perseus"). The `doc()` function returns the root node of the document whose name is its argument value (`//implementation/@href`). The argument value is URI that is used as the name of the required document in the registry.

Conditional Expressions

Conditional expressions are used when the document to be returned depends on some condition. A conditional expression takes the following form:

```
<ifExpr> ::= if <expression1>
             then <expression2>
             else <expression3>
```

It is demonstrated in the following Use Case:

Use Case 3. *Find every service in "/GAMS/Linear Algebra" such that, if it has an implementation, it runs on a machine called "perseus" and otherwise its interface is on this machine.*

```
select every service
from /GAMS/"Linear Algebra"
where
 if not (/service[empty(//implementation)])
 then
  let $d := doc(//implementation/@href) in
    $d/hardware[contains(@name, "perseus")]
 else //service-interface-description[
        contains(@href, "perseus")]
```

Use Case 3 shows a query that uses a conditional expression to decide if the current service document node has (`IsBasedOn`) an implementation. If this is the case, it takes the URI of such an implementation document and retrieves it from the registry and checks if this implementation is related to the machine `perseus`. If this is not the case, it checks (in the `else` clause) if the service has its interface on the said machine, i.e., `RunsOn` on the said machine.

Variable Bindings

An expression or a value that are used in more than one place in the same query can be bound to a variable so it does not need to be defined again. The `let` clause is used to bind a document to variable. It takes the following form:

```
<letExpr> ::= let <var> := <expression1> in <expression2>
```

The value of `expression1` is bound to variable `var` and it is substituted in `expression2`

In Use Case 3 the `let` clause binds a document to variable `d`. The value of variable `d` which is the bound document is then used in the path expression

```
$d/hardware[contains(@name, "perseus")].
```

The query in Use Case 3 aside from showing the expressiveness of MSQL in dealing with the MSDL content, it also reveals how MSQL utilizes the relations of entities as stated be the MSDL information model. It uses associations (e.g., `IsBasedOn`, `RunsOn`) among the entities of the description implicitly in the query.

Quantifiers

MSQL provides universal and existential quantifiers to test if every/some element in a document satisfies a certain condition. A quantified expression takes the following format:

```
<quantifiedExpr> ::= (some | every) <var> in <expression1>
                     satisfies <expression2>
```

It tests whether `some/every` value of `expression1`, when substituted for `var` in `expression2`, makes the result of evaluation of `expression2` *true*. It is demonstrated in the following Use Case:

Use Case 4. *Find all problems in "/GAMS/Arithmetic, error analysis/Integer" in which the OpenMath content dictionary "sts" and the "mapsto" symbol are used in the same signature.*

```
select every problem
from /GAMS/"Arithmetic, error analysis"/Integer
where
 some $s in $p//signature satisfies
  $s/MOBJ/OMA/OMS[@cd = "sts"] and
  $s/OMOBJ/OMA/OMS[@name = "mapsto"]
```

The every quantifier requires all "problem" nodes to satisfy the "some" quantifier which checks if at least one signature satisfies the condition specified.

5.1.6 Forming Queries in MSQL

The syntax of MSQL and the presented query use cases reveal an apparent difficulty in forming queries in relation to target documents. Such a difficulty stems from the need to know the MSDL document structures (e.g., the position and relations of elements inside a document when forming a path expression). This is true not only for MSQL but for any XML-based representation and any tool used for addressing and retrieving information from it. For example XPath and XQuery expressions are even more complex than those of MSQL and address XML documents that have diverse structures.

In the case of MSQL, such a difficulty is alleviated partly by the fact that MSQL operates on documents which possess a regular structure common to all documents that is imposed by the MSDL schema [92]. However, a tool (such as a user-friendly interface) for forming queries in relation to target MSDL documents is recommended (see Chapter 8).

5.2 Formal Semantics of MSQL

In this section, we present a formal definition of MSQL using denotational semantics. Denotational semantics [113, 122] is a technique for rigorously specifying the semantics of a programming language. It maps language constructs to their meaning (called denotation) by specifying semantic functions that map elements of the language syntactic domains to elements of its semantic domains. Elements of the semantic domains are mathematical values such as numbers sets, functions, etc. This clarifies the intended meaning of the language constructs and provides a correct reference for implementation.

5.2.1 Format of the Denotational Definition

The denotational definition of MSQL consists of three parts: the abstract syntax definition of the language, the semantic algebras, and the valuation function. The valuation function connects the abstract syntax and the semantic algebras, more precisely it maps abstract syntax rules to values of semantic domains. It determines the meaning of a query executed in the context of a collection of MSDL documents published in the registry by determining the set of documents retrieved from the registry.

5.2.2 Abstract Syntax

The abstract syntax consists of the syntactic domains together with a set of abstract rules which defines the MSQL grammar. The following are the syntactic domains. Capital letters and abbreviated strings such as *Q*, *E*, *V*, *Qua*, etc. are typed variables defined over syntactic domains such as *Query*, *Expression*, *Variable*, *PathExpression*, etc.

$Q \in Query$
$Qua \in Quantifier$
$E \in Expression$
$V \in Variable$
$En \in Entity$
$Cl \in ClassificationNode$
$P \in PathExpression$
$PE \in RegularPathExpression$
$Sor \in Sort$
$F \in Function$
$EL \in ExpressionList$
$N \in Name$

The following is the abstract syntax of MSQL defined in the EBNF grammar format. The left hand side of a rule symbol *::=* represents a nonterminal (i.e., a variable as defined above) and the right hand side represents a terminal and/or a nonterminal with alternative(s). Key words are written in San Serif (e.g., select, from, problem, etc.) while variables are written in capital single letters or abbreviated strings starting with capitals (e.g., *Qua*, *Cl*, *E*, *V*, etc.)

// The MSQL query.
Q ::= select *Qua En* from *Cl* where *E* orderby *E Sor*

// Result selection.
Qua ::= every
| some

// Entity types of the information model.
En ::= problem
| algorithm
| implementation
| machine
| service

// Sorting in ascending or descending order.
Sor ::= ascending
| descending

// An MSQL expression which evaluates to a value.
E ::= V // Variable.
| P // Path expression.
| $F(EL)$ // Function with its parameters as an expression list.
| some V in E satisfies E // Existential quantification.
| every V in E satisfies E // Universal quantification.
| if E then E else E // Conditional.
| let $V = E$ in E // Variable binding.
| $E + E$ // Represents arithmetic expressions, e.g., $-$ and $*$.
| $E = E$ // Comparative operators $\leq$, $<$, $\geq$, $>$, and $\neq$.
| E AND E // Represents logical connectives, e.g., OR and NOT.

// Functions.
F ::= count // Counts the number of elements in a given node.
| contains // Checks if a string contains another string.
| empty // Checks if a given element is empty.
| position // Returns the position of an element in a list.
| doc // Returns the document with the given argument name.

// Expression List as function parameters.
EL ::= E
| EL, E

// Path expression.
P ::= PE
| P PE

// Regular path expression.
PE ::= .
| $/N$ // Path step.
| $//N$ // Deep path step.
| $@N$ // Attribute step.
| $[E]$ // Predicate.

5.2.3 Semantic Algebras

The fundamental concept in semantic algebras is a semantic domain. A domain plus its operations constitute a semantic algebra.

Semantic Domains

The semantic domains for MSQL are constructed from simple sets such as integer values, boolean values, string values, and sets. More complex domains are constructed from simpler ones using, e.g., disjoint union. These provide an adequate formalism for the purpose of specifying MSQL semantics without requiring complex mathematical constructions.

The main semantic domains and their semantic algebras are as follows:

I. **Truth Values**
Domain $b \in Bool$
Operations
$true, false : Bool$
$or, and : Bool \times Bool \rightarrow Bool$
$not : Bool \rightarrow Bool$

II. **Integer Numbers**
Domain $i \in Int$
Operations
$0 \,..\, 9 : Int$
$+,\ -,\ * : Int \times Int \rightarrow Int$
$=,\ \neq,\ >,\ \geqslant,\ <,\ \leqslant : Int \times Int \rightarrow Bool$

III. **String**
Domain $s \in Str$
Operations
$a \,..\, z,\ A \,..\, Z,\ 0 \,..\, 9,\ .,\ \_ : Str$
$empty : Str$
$length : Str \rightarrow Int$
$concat : Str \times Str \rightarrow Str$
$isSubstr : Str \times Str \rightarrow Bool$ // Checks if 2nd Str is substring of 1st.
$=,\ <,\ > : Str \times Str \rightarrow Bool$ // String comparison operations.

IV. **List**
The *List* domain models lists of nodes that are produced or used by the query. It offers a set of operations for constructing and manipulating lists of nodes.

Domain $l \in List$
Operations
$newList : List$
$isEmpty : List \rightarrow Bool$
$length : List \rightarrow Int$
$append : List \times Node \rightarrow List$ // Appends a node to the list.
$get : List \times Int \rightarrow Node$ // Returns the element at position i.
$\in : List \times Node \rightarrow Bool$ // Checks if a node exists in the list.
$concat : List \times List \rightarrow List$ // Concatenates two lists.
$remove : List \times Node \rightarrow List$ // Removes node from the list.
$sort : List \times CF \rightarrow List$ // Function CF orders the values representing nodes in the list.

V. **CF**
The *CF* domain models an ordering function that compares two nodes. It is used for sorting the nodes in the query result.

$Domain\ cf \in CF = Node \times Node \rightarrow Bool$

VI. **Value**

The *Value* domain is a disjoint union of the Boolean, Integer, String, List, and Message domains. It models the possible values encountered during the evaluation of a query. Its "*in...*" operations map a given value to its respective sub-domain. Its "*is...*" operations check if a given value belongs to a certain domain.

$Domain\ v \in Value = Bool + Int + Str + List + Message$
$Operations$
$inBool : Bool \rightarrow Value$
$inInt : Int \rightarrow Value$
$inStr : Str \rightarrow Value$
$inList : List \rightarrow Value$
$inMessage : Message \rightarrow Value$
$isBool : Value \rightarrow Bool$
$isInt : Value \rightarrow Bool$
$isStr : Value \rightarrow Bool$
$isList : Value \rightarrow Bool$
$isMessage : Value \rightarrow Bool$
$boolCheck : Value \rightarrow Bool$
$boolCheck(v) = \textbf{cases}\ v\ \textbf{of}$
$isBool(b) \rightarrow true$
$isList(l) \rightarrow true$
$otherwise\ false$

$toBool : Value \rightarrow Bool$
$toBool(v) = \textbf{cases}\ v\ \textbf{of}$
$isBool(b) \rightarrow b$
$isList(l) \rightarrow not\ isEmpty(l)$
$otherwise\ false$

$<: Value \times Value \rightarrow Value$
$>: Value \times Value \rightarrow Value$
$v_1 > v_2 = \textbf{cases}\ v_1\ \textbf{of}$
$isInt(i_1) \rightarrow (\textbf{cases}\ v_2\ \textbf{of}$
$isInt(i_2) \rightarrow inBool(i_1 > i_2)$
$otherwise\ inMessage(\mathsf{error}))$
$isStr(s_1) \rightarrow (\textbf{cases}\ v_2\ \textbf{of}$
$isStr(s_2) \rightarrow inBool(s_1 > s_2)$
$otherwise\ inMessage(\mathsf{error}))$
$otherwise\ inMessage(\mathsf{error})$

The operations $<$ and $>$ with string arguments sort their arguments based on a lexicographic order of strings.

VII. **Sequence**

The *Sequence* domain is defined like the *List* domain but it has *Value* elements and it uses some additional operations.

$Domain\ seq \in Seq$
$Operations$

$newSeq : Seq$
$isEmpty : Seq \rightarrow Bool$
$length : Seq \rightarrow Int$
$append : Seq \times Value \rightarrow Seq$
$get : Seq \times Int \rightarrow Value$ // Returns the value element at position i.
$\in : Seq \times Value \rightarrow Bool$

VIII. **Registry**
The *Registry* domain models a registry as a mapping from classification nodes and entity types to sets of documents. The first operation upon the registry is to query it based on the classification node and entity type and retrieve a set of documents. The second operation is used to query the registry based on the entity name only retrieving that document with the given name.

$Domain\ r \in Registry$
$Operations$
$rQuery : ClassificationNode \times Entity \times Registry \rightarrow \mathbb{P}(Doc)$
$rQuery : Name \times Registry \rightarrow Doc$

IX. **Doc**
The *Doc* domain models an MSDL document retrieved from the registry. The *docNode* operation converts a document to a node that is to be used by the query operations.

$Domain\ doc \in Doc$
$Operations$
$docNode : Doc \rightarrow Node$

X. **Node**
The *Node* domain models a document node. It uses a set of operations to access and decide on the elements contained in a node. The *name* operation returns the string name of the node. The *position* operation returns an integer value equivalent to the node's positions relative to its parent. The *Tildren* operation returns as a list the children of the given node. The *attributes* operation returns as a list the attributes of the given node. The operations *attrName* and respectively *attrValue* return the name and the value of the given attribute.

$Domain\ n \in Node$
$Operations$
$name : Node \rightarrow String$
$position : Node \rightarrow Int$
$children : Node \rightarrow List$
$attributes : Node \rightarrow List$
$attrName : Node \rightarrow String$
$attrValue : Node \rightarrow Value$

XI. **Declaration**
The *Declaration* domain models a declaration where declared variables and their values are stored. The operation *put* declares a variable by giving it a value and *lookup* returns the value of the given variable from the declaration.

$Domain\ d \in Decl = Variable \rightarrow Value$
$Operations$
$newDecl : Decl$
$put : Decl \times Variable \times Value \rightarrow Decl$
$lookup : Decl \times Variable \rightarrow Value$

XII. **Message**
The *Message* domain models a message that is returned if something goes wrong during the query evaluation. It consists of one element, the error element.

$Domain\ m \in Message = \{\mathsf{error}\}$

XIII. **Name**
The *Name* domain models a name of, for example, an entity or a path step.

$Domain\ na \in Name$
$Operations$
$isName : String \rightarrow Bool$ // Checks if a string is a valid name.
$toName : String \rightarrow Name$ // Returns a name out of a string.

5.2.4 Semantic Functions

Semantic functions, also called valuation functions, specify the actual meaning of each syntactic construct. They map every member of the syntactic domain into a member of the semantic domain. There is a function for each rule in the abstract syntax. All the syntactic arguments are enclosed in $[\![\ ]\!]$ to show they are members of a syntactic domain. Most of the semantic functions take four arguments: a syntactic string, a declaration d, an MSDL document node n, and the registry r. An example semantic equation is

$$\mathbf{P}[\![PE]\!]\ d\ n\ r = \mathbf{PE}[\![PE]\!]\ d\ n\ r$$

In this equation, the meaning of the syntactic expression PE is defined by the semantic function **P**. **P** is a function that takes four arguments: the syntactic expression PE, a declaration d, a node n, and the registry r. The result is defined in terms of another sematic function **PE**.

In the next sections we define the following semantic functions:

- The query evaluation function
  $\mathbf{Q} : Query \rightarrow Registry \rightarrow \mathbb{P}(Doc) + Message$
- The expression evaluation function
  $\mathbf{E} : Expression \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow Value$
- The MSQL-Function evaluation function
  $\mathbf{F} : Function \rightarrow Node \rightarrow Registry \rightarrow Seq \rightarrow Value$
- The expression list evaluation function
  $\mathbf{EL} : ExpressionList \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow Seq$

- The name (of step) evaluation function

  $\mathbf{N} : Name \rightarrow Str$

- The path expression evaluation function

  $\mathbf{P} : PathExpression \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow \mathbb{P}(Node)$

- The regular path expression evaluation function

  $\mathbf{PE} : RegularPathExpression \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow \mathbb{P}(Node)$

$\mathbb{P}(S)$ denotes the powerset of S which is the set of all subsets of set S.

Query Function

We start with the function **Q** which takes a query as its first argument and the registry as its second argument. It uses the operation *rQuery* to query the registry for candidate documents satisfying the specified *classificationNode* and *Entity* type. It initializes a new declaration for use in subsequent functions. It checks first if the list of candidate documents is not empty and has at least some document satisfying the query criteria specified in the expression E_1, then based on the *some/every* clause it returns some document or every document satisfying the criteria specified in E_1. If the *every* clause is specified, it also sorts the resulting documents using an ordering function *cf*. The function *cf* operates on the list of the resulting documents to sort it in *ASCENDING/DESCENDING* order based on the criteria specified in expression E of the query.

$\mathbf{Q} : Query \rightarrow Registry \rightarrow \mathbb{P}(Doc) + Message$

$$
\begin{array}{l}
\mathbf{Q}[\![\textsf{select } Qua\ En \textsf{ from } Cl \textsf{ where } E_1 \textsf{ orderby } E_2\ Sor]\!]r = \\
\quad \textbf{let } D = rQuery(Cl, En, r) \\
\qquad d = newDecl() \\
\quad \textbf{in if } \forall doc \in D : isBool(\mathbf{E}[\![E_1]\!]\ d\ docNode(doc)\ r) \\
\qquad \textbf{then if } Qua = [\![\textsf{some}]\!] \\
\qquad\qquad \textbf{then if } \exists doc \in D : toBool(\mathbf{E}[\![E_1]\!]\ d\ docNode(doc)\ r) \\
\qquad\qquad\qquad \textbf{then } \{such\ doc : doc \in D \wedge \\
\qquad\qquad\qquad\qquad toBool(\mathbf{E}[\![E_1]\!]\ d\ docNode(doc)\ r)\} \\
\qquad\qquad\qquad \textbf{else } \{\} \\
\qquad\qquad \textbf{else if } Qua = [\![\textsf{every}]\!] \\
\qquad\qquad\qquad \textbf{then } let\ l = \{doc \in D \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid toBool(\mathbf{E}[\![E_1]\!]\ d\ docNode(doc))\ r\} \\
\qquad\qquad\qquad\qquad \textbf{in if } Sor = [\![\textsf{descending}]\!] \\
\qquad\qquad\qquad\qquad\qquad \textbf{then } sort(l,\ \lambda(doc_1,\ doc_2). \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathbf{E}[\![E_2]\!]\ d\ docNode(doc_1)\ r > \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{E}[\![E_2]\!]\ d\ docNode(doc_2)\ r)) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } sort(l,\ \lambda(doc_1,\ doc_2). \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathbf{E}[\![E_2]\!]\ d\ docNode(doc_1)\ r < \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{E}[\![E_2]\!]\ d\ docNode(doc_2)\ r)) \\
\qquad\qquad\qquad \textbf{else } \{\} \\
\qquad \textbf{else } inMessage(\textsf{error})
\end{array}
$$

Expression Function

The function **E** maps an expression *Expression* to a value in the domain *Value*. Arguments to **E** are the declaration *Decl* and the current document node *Node*.

$$\mathbf{E} : Expression \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow Value$$

- The equation for $\mathbf{E}[\![V]\!]$ looks up and returns the value of variable V from the declaration.

$$\mathbf{E}[\![V]\!]\ d\ n\ r = lookup(d,\ [\![V]\!])$$

- The equation of $\mathbf{E}[\![P]\!]$ states that its value follows from the evaluation of function **P** which is defined later for path expressions.

$$\mathbf{E}[\![P]\!]\ d\ n\ r = \mathbf{P}[\![P]\!]\ d\ n\ r$$

- The equation of $\mathbf{E}\ [\![F(EL)]\!]$ performs MSQL function evaluation. It evaluates the argument list expressions to a sequence of values and then calls the semantic function **F** to evaluate function F with the sequence of values as argument.

$$\begin{array}{l}\mathbf{E}\ [\![F(EL)]\!]\ d\ n\ r = \\ \quad \textbf{let}\ seq = \mathbf{EL}[\![EL]\!]\ d\ n\ r \\ \quad\quad \textbf{in}\ \mathbf{F}[\![F]\!]\ n\ r\ seq\end{array}$$

- The equation $\mathbf{E}[\![\mathsf{some}\ x\ \mathsf{in}\ E_1\ \mathsf{satisfies}\ E_2]\!]$ for the existential quantification tests if there is a value in the sequence we get by evaluating E_1, if substituted for x in E_2, makes the result of the evaluation of E_2 *true*.

$$\begin{array}{l}\mathbf{E}[\![\mathsf{some}\ x\ \mathsf{in}\ E_1\ \mathsf{satisfies}\ E_2]\!]\ d\ n\ r = \\ \quad \textbf{let}\ e_1 = \mathbf{E}[\![E_1]\!]\ d\ n\ r \\ \quad\quad \textbf{in if}\ not\ isList(e_1) \\ \quad\quad\quad\quad \textbf{then}\ inMessage(\mathsf{error}) \\ \quad\quad\quad\quad \textbf{else if}\ (\exists v \in e_1 : boolCheck(\mathbf{E}[\![E_2]\!]\ [[\![x]\!] \mapsto v]d\ n\ r)\ and \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad toBool(\mathbf{E}[\![E_2]\!]\ [[\![x]\!] \mapsto v]d\ n\ r)) \\ \quad\quad\quad\quad\quad\quad \textbf{then}\ inBool(true) \\ \quad\quad\quad\quad\quad\quad \textbf{else}\ inBool(false)\end{array}$$

- The equation $\mathbf{E}[\![\mathsf{every}\ x\ \mathsf{in}\ E_1\ \mathsf{satisfies}\ E_2]\!]$ for the universal quantification tests if every value in the sequence we get by evaluating E_1, if substituted for x in E_2, makes the result of the evaluation of E_2 *true*.

$$\begin{array}{l}\mathbf{E}[\![\mathsf{every}\ x\ \mathsf{in}\ E_1\ \mathsf{satisfies}\ E_2]\!]\ d\ n\ r = \\ \quad \textbf{let}\ e_1 = \mathbf{E}[\![E_1]\!]\ d\ n\ r \\ \quad\quad \textbf{in if}\ not\ isList(e_1) \\ \quad\quad\quad\quad \textbf{then}\ inMessage(\mathsf{error}) \\ \quad\quad\quad\quad \textbf{else if}\ (\forall v \in e_1 : boolCheck(\mathbf{E}[\![E_2]\!]\ [[\![x]\!] \mapsto v]d\ n\ r)\ and \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad toBool(\mathbf{E}[\![E_2]\!]\ [[\![x]\!] \mapsto v]d\ n\ r)) \\ \quad\quad\quad\quad\quad\quad \textbf{then}\ inBool(true) \\ \quad\quad\quad\quad\quad\quad \textbf{else}\ inBool(false)\end{array}$$

- The equation of $\mathbf{E}[\![\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3]\!]$ evaluates E_1; if the result is *true* the function considers E_2 for evaluation otherwise it considers E_3 for evaluation.

$$
\begin{array}{l}
\mathbf{E}[\![\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3]\!]\ d\ n\ r = \\
\quad \textbf{let}\ e = \mathbf{E}[\![E_1]\!]\ d\ n\ r \\
\quad\quad \textbf{in if}\ boolCheck(e) \\
\quad\quad\quad \textbf{then let}\ v = toBool(e) \\
\quad\quad\quad\quad \textbf{in cases}\ v\ \textbf{of} \\
\quad\quad\quad\quad\quad true \rightarrow \mathbf{E}[\![E_2]\!]\ d\ n\ r \\
\quad\quad\quad\quad\quad false \rightarrow \mathbf{E}[\![E_3]\!]\ d\ n\ r \\
\quad\quad\quad \textbf{else}\ inMessage(\mathsf{error})
\end{array}
$$

- The equation of $\mathbf{E}[\![\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2]\!]$ evaluates E_1 and binds the resulting value to variable x in the declaration d. This value is substituted for every occurrence of variable x in the evaluation of E_2.

$$
\begin{array}{l}
\mathbf{E}[\![\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2]\!]\ d\ n\ r = \\
\quad \textbf{let}\ e_1 = \mathbf{E}[\![E_1]\!]\ d\ n\ r \\
\quad\quad \textbf{in}\ \mathbf{E}[\![E_2]\!]\ [[\![x]\!] \mapsto e_1]d\ n\ r
\end{array}
$$

- The equation of $\mathbf{E}[\![E_1{+}E_2]\!]$ performs the addition on the two integer numbers resulting from the evaluation of $\mathbf{E}[\![E_1]\!]$ and $\mathbf{E}[\![E_2]\!]$.

$$
\begin{array}{l}
\mathbf{E}[\![E_1{+}E_2]\!]\ d\ n\ r = \\
\quad \textbf{let}\ e_1 = \mathbf{E}[\![E_1]\!]\ d\ n\ r \\
\quad\quad\ e_2 = \mathbf{E}[\![E_2]\!]\ d\ n\ r \\
\quad\ \textbf{in cases}\ e_1\ \textbf{of} \\
\quad\quad isInt(i_1) \rightarrow \textbf{cases}\ e_2\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad isInt(i_2) \rightarrow inInt(i_1 + i_2) \\
\quad\quad\quad\quad\quad\quad otherwise\ inMessage(\mathsf{error}) \\
\quad\quad otherwise\ inMessage(\mathsf{error})
\end{array}
$$

- The equation of $\mathbf{E}[\![E_1 = E_2]\!]$ evaluates the comparison operator $=$. It returns a boolean value, if both argument expressions evaluate to the same boolean, integer, or string and an error otherwise.

$$
\begin{array}{l}
\mathbf{E}[\![E_1 = E_2]\!]\ d\ n\ r = \\
\quad \textbf{let}\ e_1 = \mathbf{E}[\![E_1]\!]\ d\ n\ r \\
\quad\quad\ e_2 = \mathbf{E}[\![E_2]\!]\ d\ n\ r \\
\quad\quad \textbf{in cases}\ e_1\ \textbf{of} \\
\quad\quad\quad isBool(b_1) \rightarrow (\textbf{cases}\ e_2\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad isBool(b_2) \rightarrow inBool(b_1 = b_2) \\
\quad\quad\quad\quad\quad\quad\quad\quad otherwise\ inMessage(\mathsf{error})) \\
\quad\quad\quad isInt(i_1) \rightarrow (\textbf{cases}\ e_2\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad isInt(i_2) \rightarrow inBool(i_1 = i_2) \\
\quad\quad\quad\quad\quad\quad\quad\quad otherwise\ inMessage(\mathsf{error})) \\
\quad\quad\quad isStr(s_1) \rightarrow (\textbf{cases}\ e_2\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad isStr(s_2) \rightarrow inBool(s_1 = s_2) \\
\quad\quad\quad\quad\quad\quad\quad\quad otherwise\ inMessage(\mathsf{error})) \\
\quad\quad\quad otherwise\ inMessage(\mathsf{error})
\end{array}
$$

- The equation of $\mathbf{E}[\![E_1 \text{ and } E_2]\!]$ evaluates the logical connective *AND*. It returns a boolean value or an error if both of its argument expressions are not evaluated to either a boolean or a list that is not empty.

$$
\begin{array}{l}
\mathbf{E}[\![E_1 \text{ and } E_2]\!]\ d\ n\ r = \\
\quad \textbf{let } e_1 = \mathbf{E}[\![E_1]\!]\ d\ n\ r \\
\qquad\ e_2 = \mathbf{E}[\![E_2]\!]\ d\ n\ r \\
\quad \textbf{in if } boolCheck(e_1) \\
\qquad\quad \textbf{then if } boolCheck(e_2) \\
\qquad\qquad\qquad \textbf{then } toBool(e_1)\ and\ toBool(e_2) \\
\qquad\qquad\qquad \textbf{else } inMessage(\text{error}) \\
\qquad\quad \textbf{else } inMessage(\text{error})
\end{array}
$$

MSQL-Function Function

Function **F** maps an MSQL function *Function* that has a sequence of values as an argument to a value in the domain *Value*. It is called by function **E** $[\![F(EL)]\!]$ which passes to it the sequence of values as argument.

$\mathbf{F} : Function \rightarrow Node \rightarrow Registry \rightarrow Seq \rightarrow Value$

- The equation of the $\mathbf{F}[\![\text{position}]\!]$ function returns an integer value corresponding to the position of the current node in node list. It uses the operation *position* defined for the *Node* domain to determine the position of the node in the list.

$$
\begin{array}{l}
\mathbf{F}[\![\text{position}]\!]\ n\ r\ seq = \\
\quad \textbf{if } isEmpty(seq) \\
\quad \textbf{then } inInt(position(n)) \\
\quad \textbf{else } inMessage(\text{error})
\end{array}
$$

- $\mathbf{F}[\![\text{contains}]\!]$ checks if the first string evaluated argument E_1 contains the second string evaluated argument E_2. It uses the operation *isEmpty* which is defined in the **List** domain.

$$
\begin{array}{l}
\mathbf{F}[\![\text{contains}]\!]\ n\ r\ seq = \\
\quad \textbf{if } length(seq) \neq 2 \\
\quad \textbf{then } inMessage(\text{error}) \\
\quad \textbf{else cases } get(seq,\ 1) \textbf{ of} \\
\qquad isStr(s_1) \rightarrow (\textbf{cases } get(seq,\ 2) \textbf{ of} \\
\qquad\qquad\qquad isStr(s_2) \rightarrow\ inBool(isSubstr(s_1,\ s_2)) \\
\qquad\qquad\qquad otherwise\ inMessage(\text{error})) \\
\qquad otherwise\ inMessage(\text{error})
\end{array}
$$

- $\mathbf{F}[\![\text{count}]\!]$ returns as integer the length of the current node list. It uses the operation *length* which is defined in the *List* domain.

$$
\begin{array}{l}
\mathbf{F}[\![\text{count}]\!]\ n\ r\ seq = \\
\quad \textbf{if } length(seq) \neq 1 \\
\quad \textbf{then } inMessage(\text{error})
\end{array}
$$

else cases $get(seq,\ 1)$ **of**
$isList(l) \rightarrow inInt(length(l))$
$otherwise\ inMessage(\mathsf{error})$

- $\mathbf{F}[\![\mathsf{empty}]\!]$ checks if the current node list is empty. It uses the operation *isEmpty* which is defined in the **List** domain.

$\mathbf{F}[\![\mathsf{empty}]\!]\ n\ r\ seq =$
if $length(seq) \neq 1$
then $inMessage(\mathsf{error})$
else cases $get(seq,\ 1)$ **of**
$isList(l) \rightarrow isEmpty(l)$
$otherwise\ inMessage(\mathsf{error})$

- $\mathbf{F}[\![\mathsf{doc}]\!]$ retrieves a document from the registry with the given string argument as the name of the document. It uses the *rQuery* operation defined in the *Registry* domain to query the registry for the document with the given name.

$\mathbf{F}[\![\mathsf{doc}]\!]\ n\ r\ seq =$
if $length(seq) \neq 1$
then $inMessage(\mathsf{error})$
else cases $get(seq,\ 1)$ **of**
$isStr(s) \rightarrow$ **cases** s **of**
$isName(s) \rightarrow\ rQuery(toName(s),\ r)$
$otherwise\ inMessage(\mathsf{error})$
$otherwise\ inMessage(\mathsf{error})$

Expression List Function

Function **EL** maps an expression list to a sequence of values.

$\mathbf{EL} : ExpressionList \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow Seq$

- The equation of $\mathbf{EL}[\![EL,\ E]\!]$ evaluates the expression list by evaluating each expression using the function $\mathbf{EL}[\![E]\!]$ and appending it to the sequence of values.

$\mathbf{EL}[\![EL,\ E]\!]\ d\ n\ r =$
let $seq = \mathbf{E}[\![EL]\!]\ d\ n\ r$
$e = \mathbf{E}[\![E]\!]\ d\ n\ r$
in $append(seq, e)$

$\mathbf{EL}[\![E]\!]\ d\ n\ r =$
let $e = \mathbf{E}[\![E]\!]\ d\ n\ r$
in $append(newSeq, e)$

Name Function

Function **N** maps a *Name* to a value in the string domain.

$\mathbf{N} : Name \rightarrow Str$

Regular Path Expression Function

Function **PE** maps a regular path expression PE to a set of nodes.

$$\mathbf{PE} : RegularPathExpression \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow \mathbb{P}(Node)$$

- $\mathbf{PE}[\![.]\!]$ returns a list containing the context node n itself.

  $$\mathbf{PE}[\![.]\!]\ d\ n\ r = \{n\}$$

- $\mathbf{PE}[\![/N]\!]$ returns a list of nodes that are children of the current node n with the tag name (string value of N).

  $$\mathbf{PE}[\![/N]\!]\ d\ n\ r = \{m \mid m \in\ children(n)\ \wedge\ name(m) = \mathbf{N}[\![N]\!]\}$$

- $\mathbf{PE}\ [\![//N]\!]$ returns a list of nodes that are children of the current node n with tag names (string value of N) no matter where they are in the node hierarchy.

  $$\mathbf{PE}[\![//N]\!]\ d\ n\ r = p\ \mathbf{N}[\![N]\!]\ d\ n\ r$$

  $$p : String \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow \mathbb{P}(Node)$$

  $$\begin{array}{l} p(s,\ d,\ n,\ r) = \\ \quad \{m_2 : (m_2 \in children(n)\ \wedge\ name(m_2) = s)\ \vee \\ \qquad (\exists m_1 : m_1 \in children(n)\ \wedge\ name(m_1) \neq s\ \wedge \\ \qquad\qquad m_2 \in p(s,\ d,\ m_1))\} \end{array}$$

  The meaning of this recursive function definition is defined as usual by fixed point construction [113].

- $\mathbf{PE}[\![@N]\!]$ returns the attribute value of the given attribute with the given name (the string value of N).

  $$\begin{array}{l} \mathbf{PE}[\![@N]\!]\ d\ n\ r = \\ \{v \mid \exists a :\ a \in attributes(n)\ \wedge\ attrName(a) = \mathbf{N}[\![N]\!]\ \wedge \\ \qquad\qquad attrValue(a) = v\} \end{array}$$

- $\mathbf{PE}[\![\ [E]\ ]\!]$ returns a boolean value resulting from the evaluation of the predicate expression $[E]$.

  $$\begin{array}{l} \mathbf{PE}[\![\ [E]\ ]\!]\ d\ n\ r = \\ \quad \textbf{let}\ e = \mathbf{E}[\![E]\!]\ d\ n\ r \\ \qquad \textbf{in if}\ boolCheck(e)\ and\ toBool(e) \\ \qquad\quad \textbf{then}\ \{n\} \\ \qquad\quad \textbf{else}\ \{\} \end{array}$$

Path Function

Function **P** maps a path expression *PathExpression* to a power set of nodes.

$$\mathbf{P} : PathExpression \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow \mathbb{P}(Node)$$

- The equation of $\mathbf{P}[\![PE]\!]$ evaluates the **PE** function with argument PE as indicated before.

$$\mathbf{P}[\![PE]\!]\ d\ n\ r = \mathbf{PE}[\![PE]\!]\ d\ n\ r$$

- The equation of $\mathbf{P}[\![P\ PE]\!]$ applies the **P** function on the expression P and if the resulting value is a list of a single node then the function **PE** is applied on this node and the result is returned. If the result is a list of more than one node, then **PE** is evaluated on each of them and the results are gathered as a set union.

$$\begin{array}{l}\mathbf{P}[\![P\ PE]\!]\ d\ n\ r = \\ \quad \textbf{let}\ p = \mathbf{P}[\![P]\!]\ d\ n\ r \\ \quad \textbf{in}\ \bigcup\{\mathbf{PE}[\![PE]\!]\ d\ n\ r : n \in p\}\end{array}$$

5.3 MSQL Engine Architecture and Implementation

In this section, we describe the MSQL engine architecture which represents a high-level overview of the underlying implementation. We present a prototype implementation of MSQL which is based on the denotational semantics of the language. The advantage of using this approach is that it provides a reference for implementation. We also discuss some of the crucial implementation decisions such as the parser implementation and the employed data model for the representation of an MSDL document when processed by a query. We also present the resulting API and how it can be used by a client program.

5.3.1 The MSQL Engine Architecture

Figure 5.2 provides a high-level overview of the architecture incorporating the MSQL engine. It consists of the following parts:

- The *MSQL Engine* which constitutes the querying functionality of MSQL.
- The *Mathematical Registry* where MSDL documents are published and retrieved.

A client application sends a query to the MSQL engine and receives a set of MSDL descriptions. These descriptions are either returned to the user or processed further for specific tasks, e.g., to access the service having the resulting description.

The *MSQL Engine* consists of the following components which are designed according to the query structure explained in Section 5.1.4:

- **The Query Processor** which receives the query, divides it into processable parts, and hands each part to its corresponding component.

- **The Registry Handler** which receives from the processor the query part needed to retrieve MSDL documents from the registry based on their types and classifications.
- **The Expression Evaluator** which evaluates the expression part of a query against the documents retrieved from the registry, filters them, and forwards those passing the filter to the Result Quantifier and Sorter component.
- **The Result Quantifier and Sorter** which decides whether to return some or every queried document as a result and whether to sort the resulting documents.

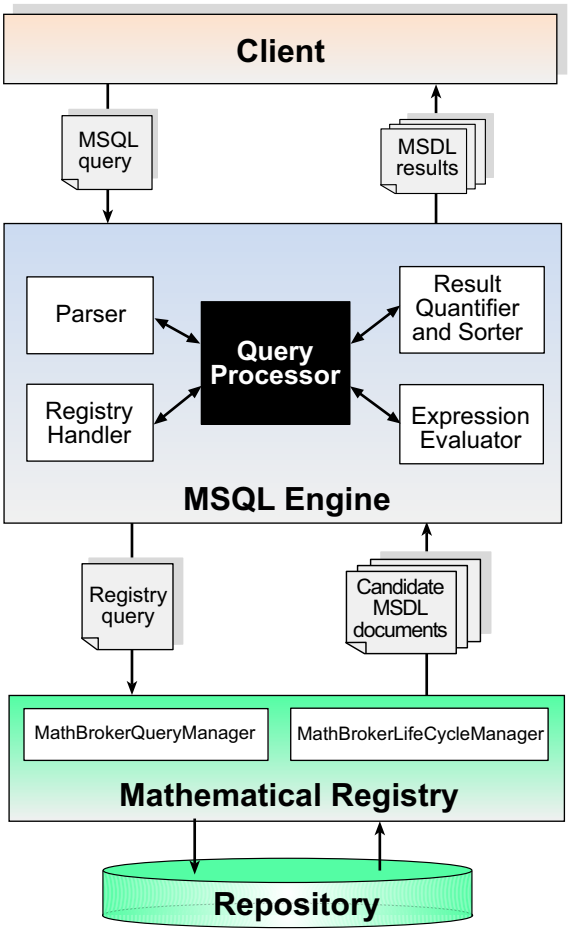


Figure 5.2: The MSQL Engine Architecture

In this architecture, an MSQL query received from a client is processed as follows:

- The query is parsed according to MSQL grammar and transformed into an abstract syntax tree.

- The engine connects to the registry and invokes the *MathBrokerQueryManager* which is part of the registry API. It queries the registry for candidate documents based on the type of document required and based on the classification node under which the required document is classified. It returns to the engine the collection of candidate MSDL documents.
- The Expression Evaluator then evaluates the condition expression of the query against each candidate document and, if the document satisfies the condition, adds it to the collection of resulting documents.
- The Result Quantifier and Sorter then quantifies and orders the resulting documents depending on the kind of query.

The engine then returns the resulting MSDL documents to the client application.

5.3.2 Semantic-Based Implementation

In Section 5.2 we presented a formal specification of MSQL using Denotational Semantics. This semantics provides a reference for the correctness of implementation of the language. Thus we have, with some minor exceptions, followed in the implemented methods the definitions of the semantics as closely as possible. One such minor exception is in the implementation of the *Query* function **Q** where we did not check that every retrieved document satisfy the stated boolean condition before evaluating the query on it.

The implementation of MSQL consists of a set of evaluation classes with a set of methods each corresponding to each semantic function. The signature of a method corresponds to the signature of the semantic function. For example, the semantic function

$$\mathbf{E}[\![V]\!]\ d\ n\ r = lookup(d,\ [\![V]\!])$$

is implemented by the Java method

```
static private Value evaluateVariableExpr(ChildAST expr,
                        Declaration declaration,
                        Node node) throws MsqlException {
  VarExpr varExpr = (VarExpr)expr;
  return declaration.lookup(varExpr);
}
```

Each semantic domain (see Section 5.2) is implemented as a separate class providing as methods the semantic functions of this domain.

5.3.3 Implementation Choices

We state the most important choices which influenced the implementation of MSQL.

Lexer, Parser, and Tree Walker Implementation

We used ANTLR (ANother Tool for Language Recognition) [103] to implement the grammar of the lexer and the parser of MSQL. ANTLR is a lexer/parser generator tool that lets one define language grammars in a EBNF (Extended Backus-Naur

Form) [1] from which it generates lexers, parsers, and data representation of Abstract Syntax Trees (ASTs). An AST is constructed during a parsing phase and used as the internal representation of an input statement.

We expressed the MSQL grammar in ANTLR syntax making the necessary changes to comply with the rules of grammar definition in ANTLR.

MSQL Lexer The role of the lexer, known also as tokenizer, is to receive a query string and construct the corresponding language tokens. The lexer grammar expressed in ANTLR syntax can be found in Appendix B.2.

MSQL Parser The role of the parser is to check if the generated tokens form a valid sentence according to the language syntax and then construct the AST. The parser grammar expressed in ANTLR syntax can be found in Appendix B.2.

MSQL Tree Walker The resulting AST of MSQL grammar belongs to what is called in ANTLR terminology a "heterogeneous AST", i.e., AST in which the nodes can have various types (corresponding to the various syntactic domains of the abstract syntax). In order to traverse (walk) an AST, we implemented a class for each kind of node in this AST.

The lexer, parser, and tree walker API can be found in [13] as part of the MSQL API.

Modeling MSDL documents as DOM objects

Like in XML, an MSDL document is modeled as a tree of nodes (see Section 5.1.3.) A node can be a root, an element node (element), an attribute, or a text. We used the DOM (Document Object Model) [40] to represent such a tree by an object of class *Node* with an interface that allows us to access the required parts of the tree.

MSQL API packages

We structured the implementation in packages:

1. The *msqlParser* package contains the lexer and parser of MSQL.
2. The *TreeWalker* package contains all the classes needed for walking the AST generated by the parser.
3. The *msqlSemantics* package (MSQL Engine) contains the implementation of the denotational semantics of MSQL.

The complete Java API of MSQL can be found in [13].

5.4 Querying in MSQL

In this section we present the MSQL API usage with some example queries based on a sample installation.

5.4.1 A Sample Installation of the MSQL Engine

We have installed the engine together with the registry and the client according to the MSQL architecture (see Figure 5.2). As one installation scenario, we installed the registry on one machine (see Section 4.4) and a client application together with the MSQL engine on another machine. The way the MSQL engine API is used by the client applications is described next.

5.4.2 Using the MSQL API

The interface `msqlQuery` represents the starting point for using the MSQL API. Its use is outlined as follows:

```
MsqlQuery msqlQuery = new MsqlQueryImpl();
MathBrokerConnection connection =
      msqlQuery.makeConnection(connectionProps);
ChildAST queryTree = msqlQuery.parseQuery(queryString);
Collection resultsCollection =
      msqlQuery.performQuery(queryTree, connection);
```

A connection is made to the registry by the `makeConnection` method. A received MSQL query is parsed using the `parseQuery` method. The query is processed and the results are returned as a collection using the `performQuery` method. Returning a collection means that all resulting documents are returned to the user.

An alternative for using the API is:

```
MsqlQuery msqlQuery = new MsqlQueryImpl();
MathBrokerConnection connection =
      msqlQuery.makeConnection(connectionProps);
ChildAST queryTree = msqlQuery.parseQuery(queryString);
Iterator resultsCollection =
      msqlQuery.iterateQuery(queryTree, connection);
```

The method `iterateQuery` is used instead of `performQuery`. It allows the client to iteratively ask for one document satisfying a query after the other. If the client is satisfied with some result, then the query needs not be performed in the rest of the candidate documents.

A complete client application demonstrating the use of the API can be found in Appendix B.5.

5.4.3 Examples

We have designed and executed a number of use cases to demonstrate the different constructs and expressions of MSQL. They are shown in Appendix B.3. Each use case starts with a textual description followed by the corresponding MSQL query. The following are two example queries with their resulting MSDL documents.

Example 1.

Find a service under the classification concept "/GAMS/Symbolic Computation" that solves the "indefinite-integration" problem.

```
select some service
from /GAMS/Symbolic Computation
where
 let $d := doc(//classification/problem/@href) in
     $d/problem[contains(@name, "indefinite-integration")]
```

The MSQL engine performed the query and returned the following MSDL document as a result:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<monet:service name="RRISC"
   xmlns:monet="http://monet.nag.co.uk/monet/OpenMathDC">
  <monet:documentation>This is an implementation of the algorithm
    Risch. We use the mathb namespace to state expected performance
    of the concrete implementation wrt to its theoretical
    complexity measure.
  </monet:documentation>
  <monet:classification>
      <monet:problem href="http://risc.uni-linz.ac.at/mathb
        roker/RischIndefIntegration/indefinite-integration"/>
  </monet:classification>
  <monet:implementation href="http://risc.uni-linz.ac.at/ma
       thbroker/RischIndefIntegration/RImpl"/>
  <monet:service-interface-description href="http://perseus
   .risc.uni-linz.ac.at:8080/axis/services/SymbolicIntegration?wsdl"/>
  <monet:service-binding>
      <monet:map operation="symbint:Integrator:indefInt"
        problem-reference="indefinite-integration" action="exec"/>
      <monet:message-construction
        message-name="symbint:IndefIntRequest"
        message-part="in0" io-ref="f"/>
  </monet:service-binding>
  <monet:service-metadata/>
  <monet:broker-interface>
      <monet:service-URI/>
  </monet:broker-interface>
</monet:service>
```

Example 2.

Find all problems under the classification concept "/GAMS/Symbolic Computation" with second input having type integer and order them according to their names in descending order.

```
select every problem
from /GAMS/Symbolic Computation
where //body/input[2]/signature//OMS[ @name = "Z" ]
orderby /problem/@name descending
```

The MSQL engine performed the query and returned the following MSDL documents as a result:

```
    <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<monet:problem name="sum"
     xmlns:monet="http://monet.nag.co.uk/monet/OpenMathDC">
    <monet:header/>
    <monet:body>
        <monet:input name="x_1">
            <monet:signature>
                <om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
                    <om:OMV name="IntegerRange"/>
                </om:OMOBJ>
            </monet:signature>
        </monet:input>
        <monet:input name="x_2">
            <monet:signature>
                <om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
                    <om:OMA>
                        <om:OMS name="mapsto" cd="sts"/>
                        <om:OMS name="Z" cd="setname1"/>
                        <om:OMV name="AbelianMonoid"/>
                    </om:OMA>
                </om:OMOBJ>
            </monet:signature>
        </monet:input>
        <monet:output name="y">
            <monet:signature>
                <om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
                    <om:OMV name="AbelianMonoid"/>
                </om:OMOBJ>
            </monet:signature>
        </monet:output>
        <monet:post-condition name="sum_post_std">
            <om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
                <om:OMA>
                    <om:OMS name="eq" cd="relation1"/>
                    <om:OMV name="y"/>
                    <om:OMA>
                        <om:OMS name="sum" cd="arith1"/>
                        <om:OMV name="x_1"/>
                        <om:OMV name="x_2"/>
                    </om:OMA>
                </om:OMA>
            </om:OMOBJ>
        </monet:post-condition>
    </monet:body>
</monet:problem>


<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<monet:problem name="product"
    xmlns:monet="http://monet.nag.co.uk/monet/OpenMathDC">
    <monet:header/>
    <monet:body>
        <monet:input name="x_1">
            <monet:signature>
```

```
                <om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
                    <om:OMV name="IntegerRange"/>
                </om:OMOBJ>
            </monet:signature>
        </monet:input>
        <monet:input name="x_2">
            <monet:signature>
                <om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
                    <om:OMA>
                        <om:OMS name="mapsto" cd="sts"/>
                        <om:OMS name="Z" cd="setname1"/>
                        <om:OMV name="AbelianMonoid"/>
                    </om:OMA>
                </om:OMOBJ>
            </monet:signature>
        </monet:input>
        <monet:output name="y">
            <monet:signature>
                <om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
                    <om:OMV name="AbelianMonoid"/>
                </om:OMOBJ>
            </monet:signature>
        </monet:output>
        <monet:post-condition name="product_post_std">
            <om:OMOBJ xmlns:om="http://www.openmath.org/OpenMath">
                <om:OMA>
                    <om:OMS name="eq" cd="relation1"/>
                    <om:OMV name="y"/>
                    <om:OMA>
                        <om:OMS name="product" cd="arith1"/>
                        <om:OMV name="x_1"/>
                        <om:OMV name="x_2"/>
                    </om:OMA>
                </om:OMA>
            </om:OMOBJ>
        </monet:post-condition>
    </monet:body>
</monet:problem>
```

5.5 Summary

In this chapter, we presented the design, the formal semantics, the architecture, and the implementation of the Mathematical Services Query Language (MSQL). MSQL is a light-weight, content-based, functional query language developed for querying the contents of mathematical documents formed in Mathematical Services Description Language (MSDL) where these documents are published in the mathematical registry. MSQL complements the metadata-based querying facility of the registry.

MSQL is designed such that it has the necessary constructs needed to interface to the registry to retrieve candidate MSDL documents, to compose query expressions that address the different information in an MSDL document, to filter these documents based on the composed query expressions, and to return the filtered MSDL documents as the results of the query.

The formal definition of MSQL is based on denotational semantics; a technique for rigorously specifying the semantics of a programming language. Denotational semantics specify semantic functions that map elements of MSQL syntactic domains to elements of its semantic domains. This gives a clear meaning to MSQL constructs and therefor provides a correct reference for implementation.

The implementation of MSQL is based on its denotational semantics. The core component of the implementation is the MSQL engine which receives queries, performs them, and returns MSDL documents as results. This functionality of the MSQL is exposed through the MSQL API.

MSDL is rich of semantic content, and in Chapter 6 we extend MSQL to accommodate semantic based query expressions that would require true reasoning. The query engine will contact a reasoner to reason about certain query expressions and return the corresponding decisions to the evaluator which uses these in processing MSQL queries.

Chapter 6

Service Discovery II: Semantic Extension of MSQL

The Mathematical Services Query Language (MSQL) supports queries on the syntactical structure of the Mathematical Services Description Language (MSDL) descriptions. However, mathematical objects are semantically rich and MSQL does not address the semantic layer of MSDL descriptions.

In this chapter we present an extension to MSQL that addresses the semantics of MSDL descriptions by queries made from predicate logic formulae. The MSQL engine is extended by a semantic evaluator to evaluate these formulae with the help of an automated reasoner.

The rest of this chapter describes the semantic extension of MSQL. It extends all concepts introduced in Chapter 5 related to MSQL such as its design, its formal semantics, its architecture and implementation. Section 6.1 introduces the semantic extension to MSQL. Section 6.2 describes the formal definition of the extension using denotational semantics. Section 6.3 describes the extension to the architecture and the implementation of the MSQL engine to incorporate semantic based querying. Section 6.4 presents some example queries performed by the two alternative installations of the MSQL engine.

6.1 A Semantic Extension to MSQL

In this section we present a semantic extension to MSQL. First we give an overview of semantics underlying MSDL descriptions. Then we present the extension of the language syntax to make use of this information.

6.1.1 MSDL and OpenMath

The Mathematical Services Description Language (MSDL) is capable of representing not only syntactic structures, but also mathematical conditions expressed in OpenMath [27] (see Section 2.5.1), an XML-based standard format for representing mathematical objects in a semantics-preserving way. The semantics of a mathematical object in OpenMath is specified by a reference to a Content Dictionary (CD). An OpenMath object such as the symbol `<OMS cd="sts" name="mapsto"/>`

is known by the *"sts"* CD in which it is defined and by *"mapsto"* name given to it in the CD.

As an example, we use the indefinite integration problem described in Figure 6.1.

```
 1 <problem name="indefinite-integration">
 2  <body>
 3   <input name="f">
 4    <signature>
 5     <OMOBJ>
 6      <OMA>
 7       <OMS cd="sts" name="mapsto"/>
 8       <OMS cd="setname1" name="R"/>
 9       <OMS cd="setname1" name="R"/>
10      </OMA>
11     </OMOBJ>
12    </signature>
13   </input>
14   <output name="i">
15     ...    (Same signature as input.)
16   </output>
17   <post-condition>
18    <OMOBJ>
19     <OMA>
20      <OMS cd="relation1" name="eq"/>
21      <OMV name="i"/>
22      <OMA>
23       <OMS cd="calculus1" name="indefint"/>
24       <OMV name="f"/>
25      </OMA>
26     </OMA>
27    </OMOBJ>
28   </post-condition>
29  </body>
30 </problem>
```

Figure 6.1: An MSDL Problem Description

This description consists of the following pieces of information:

- Input: $f : \mathbb{R} \to \mathbb{R}$ (lines 3 to 13) which expresses the type $\mathbb{R} \to \mathbb{R}$ of the input and gives it the local name f.
- Output: $i : \mathbb{R} \to \mathbb{R}$ which expresses the type $\mathbb{R} \to \mathbb{R}$ of the output and gives it the local name i.
- Post-condition: $i = \mathit{indefint}(f)$ (lines 17 to 28) which states that the output i equals the indefinite integral of the input f.

Such a semantic information can be used as a basis for discovering suitable services published in the mathematical registry. For example, suppose a client wants to solve a problem with the following specification:

- Input: a : $\mathbb{R} \to \mathbb{R}$

- Output: b : $\mathbb{R} \rightarrow \mathbb{R}$
- Post-condition: $diff(b) = a$ (which states that the differentiated output equals the input).

The client would thus like to find some service which solves a problem p such that

$$type(input_p) = \mathbb{R} \rightarrow \mathbb{R} \ \wedge \tag{6.1}$$

$$type(output_p) = \mathbb{R} \rightarrow \mathbb{R} \ \wedge \tag{6.2}$$

$$\forall\, a \in \mathbb{R} \rightarrow \mathbb{R},\, b \in \mathbb{R} \rightarrow \mathbb{R}\ (post_p(a, b) \Rightarrow diff(b) = a) \tag{6.3}$$

where formulae (6.1) and (6.2) state that the types of the input and output shall be $\mathbb{R} \rightarrow \mathbb{R}$ and the universally quantified subformula (6.3) states that the post-condition $post_p$ of the problem p implies that the differentiation of the output b equals the input a. The truth of this statement depends on knowledge available about the operation *diff*, e.g. a knowledge base may contain the formula $diff(indefint(a)) = a$ which semantically relates the operators *diff* and *indefint*. In the next section, we describe the MSQL extensions that maybe used for formulating such queries.

6.1.2 The MSQL Extension

To express predicate logic formulae including type matching (such as described in the previous section by equations (6.1), (6.2), and (6.3)), the MSQL grammar is extended by a set of rules as shown in Figure 6.2 (the complete grammar of MSQL is given in Appendix B.1).

By these rules, a new domain `<omObjExpr>` of OpenMath expressions is introduced for describing mathematical expressions (types, formulas, and terms). Using this domain, the domain `<msqlExpr>` of MSQL expressions is extended by two cases:

- The MSQL expression *"typematch(a,b)"* states that type a matches (i.e. equals or is a special version of) type b.
- The MSQL expression *"satisfy e"* states that the semantic interpretation of the predicate logic formula e (encoded as an OpenMath expression) yields *true*.

The rule for `<semanticExpr>` and its subrules define the grammar of predicate logic formulae based on the classification of OpenMath objects into *basic* objects and *compound* objects [27]. *Basic* objects include *Integers*, *Strings*, *Variables*, and *Symbols*. *Compound* objects include *Application*, *Attribution*, and *Binding*. The syntax is defined such that expressions are written in a readable prefix notation which is internally transformed into the corresponding (but hardly readable) OpenMath XML encoding. These expressions include:

MSQL Variable Expressions take the form

```
<var> ::= '$'<letter>
```

```
      <msqlQuery> ::= select ( every | some ) <entity>
                      ( from <classification> )?
                      ( where <msqlExpr> )?
                      ( orderby <msqlExpr )?;
                  ...
       <msqlExpr> ::= ... | <typeMatch> | <semanticExpr>;
      <typeMatch> ::= typematch (omObjExpr, omObjExpr);
   <semanticExpr> ::= satisfy ( <omObjExpr> );
      <omObjExpr> ::= <omApplication> | <omAttribution> | <omBinding>
                      | <omInt> | <omVar> | <omString> | <omSymbol>
                      | <var>;
  <omApplication> ::= oma '(' <omObjExpr> (, <omObjExpr> )* (
                       <varReplacement> )? ')';
  <omAttribution> ::= omattr '(' <omObjExpr>, ( <omObjExpr>
                      <omObjExpr> )(, ( <omObjExpr> <omObjExpr> ))*
                      ( <varReplacement> )? ')' ;
      <omBinding> ::= ombind '(' <omObjExpr> '[' <omBoundVariable>
                        (, omBoundVariable )* ']' <omObjExpr>
                        ( <varReplacement> )? ')';
<omBoundVariable> ::= omvar ':' ( <var> | <omVar> ) '@' '('<omObjExpr>,
                       <omObjExpr> ( <varReplacement> )? ')';
 <varReplacement> ::= '[' <omObjExpr> '/' <var> (, <omObjExpr> '/'
                       <var> )* ']';
          <omInt> ::= omi ':' <number>;
          <omVar> ::= omv ':' ( <letter> | <var> );
       <omString> ::= omstr ':' <letter> ;
       <omSymbol> ::= oms ':' <letter> :' <letter>;
            <var> ::= '$' <letter>;
                  ...
```

Figure 6.2: The MSQL Semantic Extension Grammar

When they are used within OpenMath expressions, they are usually bound by quantifiers in *Binding Expressions* (see below) using the `<omBoundVariables>` rule and are expanded to expressions making up the body of the quantified formula (e.g., see Use Case 1). They are used in *Attribution Expressions* (see below) to represent the attributed object, the attribute, and/or the value of the attribute. In *Variable Replacement Expressions* represented by the rule `<varReplacement>`, they are used as variables to be replaced by some other variable values. In *Variable Expressions* represented by the rule `<omVar>`, *MSQL Variable Expressions* are used to represent variable values read from MSDL descriptions and used in the construction of such *Variable Expressions*.

Integer Expressions represent OpenMath *Integer* objects. They take the form

```
<omInt> ::= omi ':' <number>
```

An expression such as

```
omi:9
```

is transformed to the OpenMath XML encoding

```
<OMI> 9 </OMI>
```

Variable Expressions represent OpenMath *Variable* objects. They take the form

```
<omVar> ::= omv ':' ( <letter> | <var> )
<var> ::= '$' <letter>
```

This allows us to represent a variable in two ways. The expression

```
omv:x
```

gives the name of the variable `x` directly and is encoded in OpenMath XML as

```
<OMV name=‘‘x’’/>
```

However, the expression

```
omv:$b
```

uses the MSQL variable `$b` which must be bounded by some MSQL quantifier to some variable name *i*. The expression is encoded in OpenMath XML as

```
<OMV name =‘‘i’’/>
```

String Expressions represent OpenMath *String* objects. They take the form

```
<omString> ::= omstr ':' <letter>
```

The expression

```
omstr:MSQL
```

is encoded in OpenMath XML as

```
<OMSTR>MSQL</OMSTR>
```

Symbol Expressions represent OpenMath *Symbol* objects. They take the form

```
<omSymbol> ::= oms ':' <letter> :' <letter>
```

The first `<letter>` represents the OpenMath Content Dictionary (CD) in which the *Symbol* occur and the second `<letter>` represents the name of the *Symbol*. The expression

```
oms:relation1:eq
```

represents a *Symbol* with the name "eq" (=) which occurs in the OpenMath "relation1" CD. It is encoded in OpenMath XML as

```
<OMS name="eq" cd="relation1"/>
```

Application Expressions represent OpenMath *Application* objects. An OpenMath *Application* constructs an OpenMath object from a sequence of OpenMath objects. The first object in the sequence is the *head* and the rest are the *arguments*. *Application* expressions take the form

```
<omApplication> ::= oma '(' <omObjExpr> (, <omObjExpr> )*
                            ( <varReplacement> )? ')';
```

The rule `<varReplacement>` (see below) takes care of any variable replacements in the expressions making up the *Application* object. For instance, the *Application* expression

```
oma(oms:relation1:eq, oma(oms:calculus1:diff, omv:b),
                                                omv:a)
```

is transformed to the OpenMath XML encoding

```
<OMA>
 <OMS name="eq" cd="relation1"/>
  <OMA>
    <OMS name="diff" cd="calculus1"/>
    <OMV name="b"/>
  </OMA>
 <OMV name="a"/>
</OMA>
```

Attribution Expressions represent OpenMath *Attribution* objects. An *Attribution* object is constructed from an object (attributed object) and a sequence of one or more pairs. The pairs are a *Symbol* object (attribute) and any OpenMath object (value of the attribute). They take the form

```
<omAttribution> ::= omattr '(' <omObjExpr>,
                       ( <omObjExpr> <omObjExpr> )
                       (, ( <omObjExpr> <omObjExpr> ))*
                       ( <varReplacement> )? ')'
```

For instance, the *Attribution* expression

```
omattr($a, oms:sts:type $ta)
```

attributes the object represented by variable `$a` with the attribute `type` and the attribute value represented by variable `$ta`. It is encoded in OpenMath XML as

```
1  <OMATTR>
2   <OMATP>
3     <OMS name="type" cd="sts"/>
4     <OMA>
5      <OMS name="mapsto" cd="sts"/>
6      <OMS name="R" cd="setname1"/>
7      <OMS name="R" cd="setname1"/>
```

```
 8     </OMA>
 9   </OMATP>
10   <OMV name="f"/>
11  </OMATTR>
```

assuming variable `$a` has the value `f` (line 10) read from a given MSDL description and variable `$ta` has the value $\mathbb{R} \to \mathbb{R}$ (lines 4 to 8) read from a given MSDL description.

Binding Expressions and Bound Variables represent OpenMath *Binding* objects. A *Binding* object is constructed from an OpenMath object (the binder), and from zero or more variables (the bound variables) followed by another OpenMath object (the body). They take the form

```
<omBinding> ::= ombind '(' <omObjExpr>
                           '[' <omBoundVariable>
                           (, omBoundVariable )* ']'
                           <omObjExpr>
                           ( <varReplacement> )? ')'
```

Binding expressions can represent quantified predicate logic formulae. Such a formula without variable replacement is used in the query of Use Case 1 below and a formula with variable replacement is used in the query of Use Case 2 below. The corresponding OpenMath XML encoding of this binding expression is shown in Figure 6.5.

Bound variables in *binding* expressions are represented by the rule

```
<omBoundVariable> ::= omvar ':' ( <var> | <omVar> ) '@'
                        '('<omObjExpr>, <omObjExpr>
                          ( <varReplacement> )? ')'
```

where `<var>` or `<omVar>` represent the bound variable which can be an *MSQL Variable Expression* or an OpenMath *Variable Expression* as explained above. The succeeding two expressions represented by `<omObjExpr>, <omObjExpr>` are used to denote the type of the bound variable. For example in Use Case 1, the expression

```
[omvar:$a@(oms:sts:type, $ta),
   omvar:$b@(oms:sts:type, $tb)]
```

represent two bound variables; variable $\$a$ with the type $\$ta$ and variable $\$b$ with the type $\$tb$. Both bound variables and their types are MSQL variables.

Variable Replacement Expressions are used for variable replacements in the expression they follow. They take the form

```
<varReplacement> ::= '[' <omObjExpr> '/' <var>
                         (, <omObjExpr> '/' <var> )* ']'
```

Every occurrence of the MSQL variable `<var>` in the expression which the replacement follows is replaced by the value of the expression `<omObjExpr>`. For example in Use Case 2, the variable replacement subexpression

```
[$a/$c, $b/$d]
```

replaces each occurrence of variable `$c` in the `satisfy` expression by variable `$a` and replaces each occurrence of variable `$d` by variable `$b`. The corresponding values of the replacing variables are substituted in the satisfy expression.

6.1.3 Use Cases

In the following, we give several uses cases to illustrate the application of the constructs introduced in the previous section.

Use Case 1

Find some service with problem p *such that the type checks (6.1) and (6.2) and the subformula (6.3) are satisfied.*

The request can be expressed by the following query:

```
select some service
from /GAMS/"Symbolic Computation"
where let $p:= doc(//problem/@href) in
          $a:= $p//input/@name,
          $b:= $p//output/@name,
         $ta:= $p//input/signature/OMOBJ,
         $tb:= $p//output/signature/OMOBJ,
       $post:= $p//post-condition/OMOBJ in
 (typematch(oma(oms:sts:mapsto(oms:setname1:R,
                       oms:setname1:R)), $ta)) and
 (typematch($tb, oma(oms:sts:mapsto(oms:setname1:R,
                            oms:setname1:R)))) and
 (satisfy(ombind(oms:quant1:forall
  [omvar:$a@(oms:sts:type, $ta),
   omvar:$b@(oms:sts:type, $tb)]
  oma(oms:logic1:implies, $post,
      oma(oms:relation1:eq,
          oma(oms:calculus1:diff, omv:$b), omv:$a)))))
```

Variable `$p` represents the problem description of the service retrieved from the registry by the *doc* function according to the problem *href* provided as part of the service description. Variables `$a` and `$b` represent the names of the input and the output of the problem. Variables `$ta` and `$tb` represent the types of the input and the output of the problem. Variable `$post` represents the post-condition of the problem.

The two `typematch` expressions correspond to formulae (6.1) and (6.2). They check if type $\mathbb{R} \to \mathbb{R}$ matches the type `$ta` of the input and if the type `$tb` of the output matches type $\mathbb{R} \to \mathbb{R}$.

The `satisfy` expression corresponds to the universally quantified formula (6.3).

As can be noticed in the type check subformulae (6.1) and (6.2) and the quantified subformula (6.3) (respectively their corresponding `typematch` and `satisfy` expressions), the type information $\mathbb{R} \to \mathbb{R}$ and the specification $\mathit{diff}(b) = a$ are

provided explicitly by the user. In Use Case 2, these pieces of information are read from a given MSDL description and are assigned to variables that represent them in the type check and the predicate logic formula (respectively in the corresponding `typematch` and `satisfy` expressions of the query).

Use Case 2

Find some service with problem p_1 such that its input, output, and postcondition and the input, the output, and the postcondition of the problem p_2 given in Figure 6.3 satisfy the following formula

$$type(input_{p2}) = type(input_{p1}) \ \wedge \tag{6.4}$$

$$type(output_{p1}) = type(output_{p2}) \ \wedge \tag{6.5}$$

$$\forall\, a \in type(input_{p2}),\, b \in type(output_{p2})\ (post_{p1}(a,b) \Rightarrow (post_{p2}(c,d)) \tag{6.6}$$

```
<monet:problem name="indefinite-integrationB">
  <monet:header></monet:header>
  <monet:body>
  <monet:input name="x"/>
   <monet:signature>
    <om:OMOBJ>
     <om:OMA>
      <om:OMS cd="sts" name="mapsto"/>
      <om:OMS cd="setname1" name="R"/>
      <om:OMS cd="setname1" name="R"/>
     </om:OMA>
    </om:OMOBJ>
   </monet:signature>
  </monet:input>
  <monet:output name="y"/>
     ...
  </monet:output>
  <monet:post-condition>
   <om:OMOBJ>
    <om:OMA>
     <om:OMS cd="relation1" name="eq"/>
     <om:OMA>
      <om:OMS cd="calculus1" name="diff"/>
      <om:OMV name="y"/>
     </om:OMA>
     <om:OMV name="x"/>
    </om:OMA>
   </om:OMOBJ>
  </monet:post-condition>
 </monet:body>
</monet:problem>
```

Input:
$x : \mathbb{R} \to \mathbb{R}$

Output:
$y : \mathbb{R} \to \mathbb{R}$

Post-condition:
$diff\,(y) = x$

Figure 6.3: A Given Problem Specification

The request can be expressed by the following query:

```
select some service
from /GAMS/"Symbolic Computation"
where
 let $p1:= doc(//problem/@href) in
      $a:= $p1//input/@name,
      $b:= $p1//output/@name,
     $ta:= $p1//input/signature/OMOBJ,
     $tb:= $p1//output/signature/OMOBJ,
  $post1:= $p1//post-condition/OMOBJ,
 let $p2 := localdoc(indefinite-integrationB.xml) in
      $c := $p2//input/@name,
      $d := $p2//output/@name,
     $tc := $p2//input/signature/om:OMOBJ,
     $td := $p2//output/signature/om:OMOBJ,
  $post2 := $p2//post-condition/om:OMOBJ in
 (typematch($tc, $ta)) and
 (typematch($tb, $td)) and
 (satisfy(ombind(oms:quant1:forall
  [omvar:$a@(oms:sts:type, $tc), omvar:$b@(oms:sts:type, $td)]
  oma(oms:logic1:implies, $post1, $post2)[$a/$c, $b/$d])))
```

Variable `$p1` represents the problem description of the service retrieved from the registry by the *doc* function according to the problem *href* provided as part of the service description. Variables `$a` and `$b` represent the names of the input and the output of the problem. Variables `$ta` and `$tb` represent the types of the input and the output of the problem. Variable `$post1` represents the post-condition of the problem.

Variable `$p2` represents the description of the second problem (i.e., the user specification) read from the file "indefinite-integrationB.xml" by the function *localdoc*. Variables `$c` and `$d` represent the names of the input and the output of this problem. Variables `$tc` and `$td` represent the types of the input and the output of this problem. Variable `$post2` represents the post-condition of this problem.

The two `typematch` expressions correspond to formulae (6.4) and (6.5). They check if type `$tc` of the input of the given (second) problem matches the type `$ta` of the input of the target (first) problem and if the type `$tb` of the output of the target problem matches type `$ty` of the output of the given (second) problem.

The `satisfy` expression corresponds to the universally quantified formula (6.6). It includes the variable replacement expression `[$a/$c, $b/$d]` which states that each occurrence of variable `$c` in the `satisfy` expression is replaced by variable `$a`. Similarly each occurrence of variable `$d` is replaced by variable `$b`. The corresponding values of the replacing variables are substituted in the expression. This maps the names of input/output variables to that of the target (first) problem which we are actually querying for.

In Section 6.3, we explain how these queries (in Use Case 1 and 2) are handled by the query engine.

6.2 Formal Semantics of the Extension

In Section 5.2 we presented a formal definition of MSQL using denotational semantics. In this section we extend that definition to include the definition of the semantic extension of MSQL.

6.2.1 Abstract Syntax

The abstract syntax consists of the extensions to the syntactic domains defined in Section 5.2.2 and the abstract rules of the semantic grammar shown in Figure 6.2.

The MSQL syntactic domains (see Section 5.2.2) are extended by two domains:

$E \in \mathit{Expression}$
...
$OME \in \mathit{OpenMathExpression}$
$OMEL \in \mathit{OpenMathExpressionList}$

The abstract syntax (see Section 5.2.2) is extended by the following rules which correspond to the grammar rules shown in Figure 6.2:

// An MSQL expression which evaluates to a value.
$E ::= V$ // Variable.
| ...
| $\mathsf{satisfy}(OME)$ // Satisfy an OpenMath expression.
| $\mathsf{typematch}(OME, OME)$ // Expressions type matching.

// OpenMath expressions.
$OME ::= \mathsf{oma}(OMEL)$ // OpenMath application.
| $\mathsf{omattr}(OME, OMEL)$ // OpenMath attribution.
| $\mathsf{ombind}(OME[OMEL]OME)$ // OpenMath binding.
| $\mathsf{oms} : N : N$ // OpenMath symbol.
| $\mathsf{omstr} : N$ // OpenMath string.
| $\mathsf{omv} : N$ // OpenMath variable.
| $\mathsf{omi} : N$ // OpenMath integer.
| $OME[OMEL]$ // OpenMath variable replacement.
| V // MSQL variable.

// OpenMath Expression List.
$OMEL ::= OME$
| $OMEL,\ OME$

6.2.2 Semantic Algebras

The semantic algebras defined in Section 5.2.3) are extended by the domains stated below.

I. **Node**
The *Node* domain models a document node as well as an OpenMath Object node. It uses a set of operations to access and decide on the elements contained in a node. The *name* operation returns the name of the node as a string. The *position* operation returns an integer value equivalent to the node's positions relative to its parent. The *children* operation returns the children of the given

node as a list of nodes. The *attributes* operation returns as a list the attributes of the given node. The operations *attrName* and *attrValue* return the name and the value of the given attribute respectively. The *Node* domain also uses a set of operations (those with their names prefixed with om) for constructing specific OpenMath object nodes.

$Domain\ n \in Node$
$Operations$
$name : Node \rightarrow String$
$position : Node \rightarrow Int$
$children : Node \rightarrow List$
$attributes : Node \rightarrow List$
$attrName : Node \rightarrow String$
$attrValue : Node \rightarrow Value$
$omApplication : Seq \rightarrow Node$
$omBinding : Node \times Seq \times Node \rightarrow Node$
$omAttribution : Node \times Seq \rightarrow Node$
$omSymbol : Str \times Str \rightarrow Node$
$omString : Str \rightarrow Node$
$omInteger : Str \rightarrow Node$
$omVariable : Str \rightarrow Node$
$replacement : Node \times Seq \rightarrow Node$

II. **OMValue**
The *OMValue* domain is a disjoint union of the *Node*, *Str*, and *Message* domains. It models the possible values encountered during the evaluation of an OME expression. Its "*in...*" operations map a given value to its respective sub-domain. Its "*is...*" operations check if a given value belongs to one of the three sub-domains.

$Domain\ omValue \in OMValue = Node + Str + Message$
$Operations$
$inNode : Node \rightarrow OMValue$
$inStr : Str \rightarrow OMValue$
$inMessage : Message \rightarrow OMValue$
$isNode : OMValue \rightarrow Bool$
$isStr : OMValue \rightarrow Bool$
$isMessage : OMValue \rightarrow Bool$

III. **Prover**
The *Prover* domain models the prover (the automated reasoner). The operation *pProve* takes a logical formula as an OpenMath node and a prover and returns a boolean value that describes the validity of the formula as determined by the prover (the result maybe an error if something has gone wrong). The *pTypeMatch* operation takes two types in the form of OpenMath and hands them to the prover which checks their compatibility. It returns the result as a boolean value, otherwise it returns an error message.

$Domain\ pr \in Prover$
$Operations$

$pProve : Node \times Prover \rightarrow Value$
$pTypeMatch : Node \times Node \times Prover \rightarrow Value$

6.2.3 Semantic Functions

The semantic functions defined in Section 5.2.4 are extended by the following functions which specify the meaning of the syntactic rules of the abstract syntax.

Expression Function

The function **E** maps an expression *Expression* to a value in the domain *Value*. Arguments to **E** are the declaration *Decl*, the current document node *Node*, the registry *Registry*, and the prover *Prover* .

$\mathbf{E} : Expression \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow Prover \rightarrow Value$

- The equation for logical satisfiability $\mathbf{E}[\![\mathsf{satisfy}(OME)]\!]$ tests if the logical OpenMath expression OME is logically valid by sending it to the prover which performs the operation *pProve* on it.

$$\begin{array}{l}\mathbf{E}[\![\mathsf{satisfy}(OME)]\!]\ d\ n\ r\ pr = \\ \quad \textbf{let}\ e = \mathbf{OME}[\![OME]\!]\ d\ n\ r\ pr \\ \quad\quad \textbf{in if}\ isNode(n) \\ \quad\quad\quad\quad \textbf{then}\ pProve(n, pr) \\ \quad\quad\quad\quad \textbf{else}\ inMessage(\mathsf{error})\end{array}$$

- The equation of $\mathbf{E}[\![\mathsf{typematch}(OME_1, OME_2)]\!]$ (the type matching) tests if the types represented by OME_1 and OME_2 are compatible.

$$\begin{array}{l}\mathbf{E}[\![\mathsf{typematch}(OME_1, OME_2)]\!]\ d\ n\ r\ pr = \\ \quad \textbf{let}\ e_1 = \mathbf{OME}[\![OME_1]\!]\ d\ n\ r\ pr \\ \quad\quad\ e_2 = \mathbf{OME}[\![OME_2]\!]\ d\ n\ r\ pr \\ \quad\quad \textbf{in cases}\ n_1\ \textbf{of} \\ \quad\quad\quad\quad isNode(n_1) \rightarrow\ \textbf{cases}\ n_2\ \textbf{of} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad isNode(n_2) \rightarrow \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad pTypeMatch(n_1, n_2, pr) \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad otherwise\ inMessage(\mathsf{error}) \\ \quad\quad\quad\quad otherwise\ inMessage(\mathsf{error})\end{array}$$

OpenMath Expression Node construction Function

Function **OME** maps an OpenMath expression *Function* to an OpenMath Node.

$\mathbf{OME} : OpenMathExpression \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow$
$Prover \rightarrow OMValue$

- The equation of $\mathbf{OME}[\![\mathsf{oma}(OMEL)]\!]$ (the OpenMath application expression) returns an OpenMath application node corresponding to the expression list $OMEL$. The *Node* domain operation *omApplication()* takes the sequence

of OpenMath nodes constructed by the evaluation of $\mathbf{OMEL}[\![OMEL]\!]$ function and constructs the corresponding OpenMath application node.

$$
\begin{array}{l}
\mathbf{OME}[\![\mathsf{oma}(OMEL)]\!]\ d\ n\ r\ pr = \\
\quad \mathbf{let}\ seq = \mathbf{OMEL}[\![OMEL]\!]\ d\ n\ r\ pr \\
\quad\ \mathbf{in\ cases}\ seq\ \mathbf{of} \\
\qquad\qquad not\ isEmpty(seq) \rightarrow omApplication(seq) \\
\qquad\qquad otherwise\ inMessage(\mathsf{error})
\end{array}
$$

- The equation of $\mathbf{OME}[\![\mathsf{omattr}(OME, OMEL)]\!]$ (the OpenMath attribution expression) results in an OpenMath attribution node. The expression OME is an OpenMath attribution object. The expression list $OMEL$ represents a list of (key, value) object pairs. $\mathbf{OME}[\![OME]\!]$ is evaluated to an OpenMath object node, While $\mathbf{OMEL}[\![OMEL]\!]$ is evaluated to a sequence of OpenMath object nodes. The *omAttribuation()* operation takes both, the OpenMath object node and the sequence of OpenMath object nodes and constructs an OpenMath attribution object node.

$$
\begin{array}{l}
\mathbf{OME}[\![\mathsf{omattr}(OME, OMEL)]\!]\ d\ n\ r\ pr = \\
\quad \mathbf{let}\ e = \mathbf{OME}[\![OME]\!]\ d\ n\ r\ pr \\
\qquad\ seq = \mathbf{OMEL}[\![OMEL]\!]\ d\ n\ r\ pr \\
\quad\ \mathbf{in\ cases}\ e\ \mathbf{of} \\
\qquad isNode(n) \rightarrow \mathbf{cases}\ seq\ \mathbf{of} \\
\qquad\qquad\qquad not\ isEmpty(seq) \rightarrow \\
\qquad\qquad\qquad\qquad\qquad omAttribution(n, seq) \\
\qquad\qquad\qquad otherwise\ inMessage(\mathsf{error}) \\
\qquad otherwise\ inMessage(\mathsf{error})
\end{array}
$$

- The equation of $\mathbf{OME}[\![\mathsf{ombind}(OME_1, OMEL, OME_2)]\!]$ (the binding expression) results in an OpenMath attribution node. The expression OME_1 is an OpenMath binder object. The expression list $OMEL$ represents a list of OpenMath objects corresponding to bound variables. The expression OME_2 is an OpenMath object representing the body of the binding expression. The *omBinding()* operation takes the binder node, the sequence of variable nodes, and the body node and constructs an OpenMath binding object node.

$$
\begin{array}{l}
\mathbf{OME}[\![\mathsf{ombind}(OME_1, OMEL, OME_2)]\!]\ d\ n\ r\ pr = \\
\quad \mathbf{let}\ e_1 = \mathbf{OME}[\![OME_1]\!]\ d\ n\ r\ pr \\
\qquad\ seq = \mathbf{OMEL}[\![OMEL]\!]\ d\ n\ r\ pr \\
\qquad\ e_2 = \mathbf{OME}[\![OME_2]\!]\ d\ n\ r\ pr \\
\quad\ \mathbf{in\ cases}\ e_1\ \mathbf{of} \\
\qquad isNode(n_1) \rightarrow \mathbf{cases}\ seq\ \mathbf{of} \\
\qquad\qquad\qquad not\ isEmpty(seq) \rightarrow \\
\qquad\qquad\qquad\qquad\qquad \mathbf{cases}\ e_2\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\ isNode(n_2) \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad omBinding(n_1, seq, n_2) \\
\qquad\qquad\qquad\qquad\qquad otherwise\ inMessage(\mathsf{error}) \\
\qquad\qquad\qquad otherwise\ inMessage(\mathsf{error}) \\
\qquad otherwise\ inMessage(\mathsf{error})
\end{array}
$$

- The equation of the OpenMath symbol expression $\mathbf{OME}[\![\mathsf{oms} : N_1 : N_2]\!]$ results in an OpenMath symbol node. The expressions N_1 and N_2 is evaluated to a string. The *omSymbol()* operation takes the two strings and constructs an OpenMath symbol object node.

$$\begin{array}{l}\mathbf{OME}[\![\mathsf{oms} : N_1 : N_2]\!]\ d\ n\ r\ pr = \\ \quad \mathbf{let}\ s_1 = \mathbf{N}[\![N_1]\!] \\ \quad \mathbf{let}\ s_2 = \mathbf{N}[\![N_2]\!] \\ \quad\quad \mathbf{in}\ omSymbol(s_1, s_2)\end{array}$$

- The equation of $\mathbf{OME}[\![\mathsf{omstr} : N]\!]$ (the OpenMath string expression) results in an OpenMath string node. The expression N is evaluated to a string. The *omString()* operation takes this string and constructs an OpenMath string object node.

$$\begin{array}{l}\mathbf{OME}[\![\mathsf{omstr} : N]\!]\ d\ n\ r\ pr = \\ \quad \mathbf{let}\ s = \mathbf{N}[\![N]\!] \\ \quad\quad \mathbf{in}\ omString(s)\end{array}$$

- The equation of $\mathbf{OME}[\![\mathsf{omv} : N]\!]$ (the OpenMath variable expression) results in an OpenMath variable node. The expression N is evaluated to a string. The *omVariable()* operation takes this string and constructs an OpenMath variable object node.

$$\begin{array}{l}\mathbf{OME}[\![\mathsf{omv} : N]\!]\ d\ n\ r\ pr = \\ \quad \mathbf{let}\ s = \mathbf{N}[\![N]\!] \\ \quad\quad \mathbf{in}\ omVariable(s)\end{array}$$

- The equation of $\mathbf{OME}[\![\mathsf{omi} : N]\!]$ (the OpenMath integer expression) results in an OpenMath integer node. The expression N is evaluated to a string. The *omInteger()* operation takes this string (a number in a string format) and constructs an OpenMath integer object node.

$$\begin{array}{l}\mathbf{OME}[\![\mathsf{omi} : N]\!]\ d\ n\ r\ pr = \\ \quad \mathbf{let}\ s = \mathbf{N}[\![N]\!] \\ \quad\quad \mathbf{in}\ omInteger(s)\end{array}$$

- The equation of $\mathbf{OME}[\![V]\!]$ (the MSQL variable expression as an OpenMath expression) results in the evaluation of the variable V using the MSQL function $\mathbf{E}[\![V]\!]$.

$$\begin{array}{l}\mathbf{OME}[\![V]\!]\ d\ n\ r\ pr = \\ \quad \mathbf{let}\ e = \mathbf{E}[\![V]\!]\ d\ n\ r\ pr \\ \quad\quad \mathbf{in\ cases}\ e\ \mathbf{of} \\ \qquad\quad isNode(n) \rightarrow inNode(n) \\ \qquad\quad isStr(s) \rightarrow inStr(s) \\ \qquad\quad otherwise\ inMessage(\mathsf{error})\end{array}$$

- The equation of $\mathbf{OME}[\![OME[OMEL]]\!]$ (the variable replacement expression) results in the replacement of every occurrence of the value of the second OME by the first OME for each pair of $OMEL$ expression list in expression OME. For example, if we have a list $OMEL$ of one pair, then $\mathbf{OME}[\![OME_1[OME_2/OME_3]]\!]$ results in the replacement of every occurrence of the value of expression OME_3 in expression OME_1 by expression OME_2.

$$
\begin{array}{l}
\mathbf{OME}[\![OME[OMEL]]\!]\ d\ n\ r\ pr = \\
\quad \textbf{let}\ seq = \mathbf{OMEL}[\![OMEL]\!]\ d\ n\ r\ pr \\
\qquad e = \mathbf{OME}[\![OME]\!]\ d\ n\ r\ pr \\
\quad \textbf{in cases}\ seq\ \textbf{of} \\
\qquad not\ isEmpty(seq) \rightarrow \textbf{cases}\ e\ \textbf{of} \\
\qquad\qquad\qquad\qquad isNode(n) \rightarrow replacement(n, seq) \\
\qquad\qquad\qquad\qquad otherwise\ inMessage(\mathsf{error}) \\
\qquad otherwise\ inMessage(\mathsf{error})
\end{array}
$$

OpenMath Expression List Function

Function **OMEL** maps an expression list to a sequence of Nodes.

$\mathbf{OMEL} : OpenMathExpressionList \rightarrow Decl \rightarrow Node \rightarrow Registry \rightarrow Seq$

- $$
\begin{array}{l}
\mathbf{OMEL}[\![OMEL,\ OME]\!]\ d\ n\ r = \\
\quad \textbf{let}\ seq = \mathbf{OME}[\![OMEL]\!]\ d\ n\ r \\
\qquad e = \mathbf{OME}[\![OME]\!]\ d\ n\ r \\
\quad \textbf{in}\ append(seq, e)
\end{array}
$$

- $$
\begin{array}{l}
\mathbf{OMEL}[\![OME]\!]\ d\ n\ r = \\
\quad \textbf{let}\ e = \mathbf{OME}[\![OME]\!]\ d\ n\ r \\
\quad \textbf{in}\ append(newSeq, e)
\end{array}
$$

6.3 Architecture and Implementation

The MSQL semantic extension has been incorporated into the MSQL engine architecture and implementation such that syntactic as well as semantic based queries can be performed.

6.3.1 MSQL Extended Architecture

The complete architecture containing the MSQL engine including the semantics extension is shown in Figure 6.4. It consists of the following components:

- The MSQL **Query Engine** which contains the MSQL query functionality. It consists of the following components:
  - The **Query Processor** which receives the query from the client, decomposes it into processable parts, and hands each part to the corresponding component.

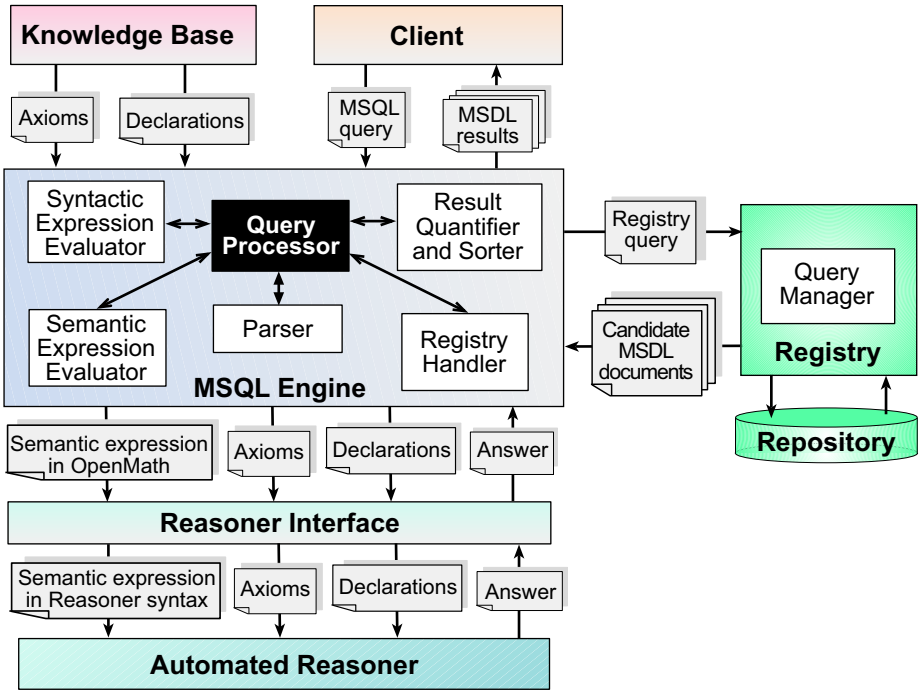


Figure 6.4: The MSQL Engine Architecture

- The **Parser** receives the query from the processor and parses it according to the MSQL syntax. If the query does not comply with the syntax, an error message is returned to the processor which forwards the message to the client.
- The **Registry Handler** receives from the processor the *entity* and *classificationConcept* parts of the query. It composes a registry query to retrieve *EVERY*/*SOME* description document of the given *entity* type classified under the given *classificationConcept*.
- The **Syntactic Expression Evaluator** receives from the Query Processor the syntactic expression part of the query and evaluates it against each description document retrieved from the registry. It returns to Query Processor those documents for which the expression evaluates to *true*.
- The **Semantic Expression Evaluator** receives from the Query Processor the semantic expression part of the query and evaluates it against each description document retrieved from the registry. It returns to the Query Processor those documents for which the expression evaluates to *true*. Unlike the Syntactic Expression Evaluator, the Semantic Expression Evaluator does not perform the whole evaluation by itself. It rather takes the semantic expression, converts it into OpenMath format (see Figure 6.5), retrieves from the Knowledge Base the axiom(s) and type declaration(s) needed to reason about the semantic expression and

sends all of them to the Reasoner Interface. As required by the Reasoner Interface, the axioms are represented in OpenMath format and the declarations are represented in OMDoc [97] format.

- The **Result Quantifier and Sorter** receives from the Query Processor SOME/EVERY document filtered by the two evaluators, orders (if needed) the documents according to the ORDERBY expression, and returns them as the query result to the Client.

- The **Registry** which stores a collection of published MSDL documents of different *entity* types and classifies them according to some registry-predefined classification schemes. Query requests to the registry are handled by the Query Manager of the registry.
- The **Reasoner Interface** receives from the Semantic Expression Evaluator the semantic expression part of the query in OpenMath, the axiom(s) in OpenMath, and the declaration(s) in OMDoc and converts each one to the format required by the Automated Reasoner and hands them to the reasoner. It gets the answer from the reasoner and sends it to the Semantic Expression Evaluator. The Reasoner Interface used is a component of the RISC ProofNavigator [114].
- The **Automated Reasoner** reasons about semantic expressions based on the axiom(s) and declaration(s) given and returns the answer to the Reasoner Interface. The Automated Reasoner currently used is the Cooperating Validity Checker Lite (CVCL) [18].
- The **Knowledge Base** holds declarations of OpenMath symbols that may be used in semantic queries together with axioms that describe the semantics of that symbols.

Performing the Semantic Query

Based on this architecture, we summarize the actions taken to perform the query in Use Case 1 (see Section 6.1.2):

- The Query Engine receives the query from the Client and hands it to the Query Processor
- The Query Processor asks the Parser to parse the query according to the MSQL syntax. If the query does not comply with the MSQL syntax, an error message is returned to the Client.
- The Query Processor decomposes the query to processable parts. It hands the registry-related part (the *service entity* and the *classificationConcept "/GAMS/Symbolic Computation"*) to the registry handler.
- The Registry Handler forms a registry query based on the *entity* and the *classificationConcept*, connects to the Registry and hands the registry query to the Query Manager of the Registry which performs the query and returns a set of candidate *service* documents to the Registry Handler.
- The Query Processor asks the Syntactic Expression Evaluator to evaluate the syntactic expression part on the current *service* document. The Syntactic Expression part consists of a *let* expression which has six assignment

subexpressions. The evaluations of these subexpressions assign values to variables $a, $b, $ta, $tb, and $post representing the input, the output, the input type, the output type, and the post-condition respectively. These variables are used in the semantic expression of the query.

- The Query Processor asks the Semantic Expression Evaluator to evaluate the semantics expression against the (same) current *service* document. The Semantic Evaluator performs the following steps:
  - It performs the type checking required by the two *typematch* expressions. If the result of the check is *true* it proceeds to the next step. Otherwise it returns *false* and the Query Processor proceeds to perform the query on the next candidate document.
  - It converts the `satisfy` expression to OpenMath format. The OpenMath representation of the satisfy expression is shown in Figure 6.5. The conversion also takes care of variable substitution (e.g., variable `$a` is substituted by the input name f).
  - It retrieves from the Knowledge Base the declarations of the symbols *diff* and *indefint* represented in OMDoc. The two symbols occur in the OpenMath representation of the `satisfy` formula after variable substitution. The declaration of the *diff* symbol is shown in Figure 6.6. The *indefint* symbol has a similar declaration.
  - It retrieves the axiom $\mathit{diff}(\mathit{indefint}(a)) = a$ from the Knowledge Base. This axiom is represented by the following quantified formula encoded in OpenMath XML (in the same way as the *satisfy* expression)

$$\forall f \in \mathbb{R} \to \mathbb{R}\ (\mathit{indefint}(\mathit{diff}(f)) = f)$$

  - It hands the satisfy expression (in OpenMath), the declarations (in OMDoc), and the axiom (in OpenMath) to the Reasoner Interface which converts each of them to the syntax required by the reasoner. The reasoner decides about the truth value of the expression based on the given axiom and declarations and returns the answer to the RISC ProofNavigator which in turn returns the answer to the Semantic Expression Evaluator.
- If the evaluation of the semantic expression yields *true*, the Query Processor returns the current *service* document to the Result Quantifier and Sorter which returns it to the Client as the ultimate result (because of the *some* clause) of the query. If the evaluation is *false* the Query Processor proceeds to process the query on the next candidate *service* document. If the evaluation is *false* for all candidate documents, then no document is returned as a result of the query.

Performing the query in Use Case 2 follows similar steps as the query in Use Case 1 with additional steps to read a specification from a given description and as consequence perform the required variable replacements. The *satisfy* expression is formed based on the problem description retrieved from the registry and the specification read from the given description (indefinite-integrationB.xml). To map input and output names to those of the target problem description retrieved from

```
 1 <OMOBJ>
 2  <OMBIND>
 3   <OMS name="forall" cd="quant1"/>
 4   <OMBVAR>
 5    <OMATTR>
 6      <OMATP>
 7        <OMS name="type" cd="sts"/>
 8        <OMA>
 9         <OMS name="mapsto" cd="sts"/>
10         <OMS name="R" cd="setname1"/>
11         <OMS name="R" cd="setname1"/>
12        </OMA>
13      </OMATP>
14      <OMV name="f"/>
15     </OMATTR>
16     <OMATTR>
17     ...
18      <OMV name="i"/>
19     </OMATTR>
20   </OMBVAR>
21   <OMA>
22    <OMS name="implies" cd="logic1"/>
23    <OMA>
24     <OMS name="eq" cd="relation1"/>
25     <OMV name="i"/>
26     <OMA>
27      <OMV name="indefint" cd="calculus1"/>
28      <OMV name="f"/>
29     </OMA>
30    </OMA>
31    <OMA>
32     <OMS name="eq" cd="relation1"/>
33     <OMA>
34      <OMV name="diff" cd="calculus1"/>
35      <OMV name="i"/>
36     </OMA>
37     <OMV name="f"/>
38    </OMA>
39   </OMA>
40  </OMBIND>
41 </OMOBJ>
```

Lines 5 to 15 represent the conversion of the binder expression

```
omvar:$a@(oms:sts:type, $ta)
```

with the variables $a and $b substituted by their values. It represents the declaration

$$f : \mathbb{R} \to \mathbb{R}$$

Lines 21 to 39 represent the conversion of the *satisfy* subexpression

```
oma(oms:logic1:implies, $post,
    oma(oms:relation1:eq,
        oma(oms:calculus1:diff,
            omv:$b), omv:$a))
```

with the variables $post, $a, and $b appropriately substituted by their values. It represents the implication

$$i = \mathit{indefint}(f) \Rightarrow \mathit{diff}(i) = f.$$

Figure 6.5: The *satisfy* Expression of Use Case 1

the registry, the needed variable replacements is performed using the subexpression `[$a/$c, $b/$d]` where $\$a$ denotes the input name f of the target description, $\$c$ denotes the input name x of the given description, $\$b$ denotes the output name i of the target description, and $\$d$ denotes the output name y of the given description; correspondingly, each occurrence of x is replaced by f and each occurrence of y is replaced by i.

The resulting OpenMath representation of the satisfy expression is the same as that of the query in Use Case 1 (see Figure 6.5).

6.3.2 Extending the Implementation

We have extended the implementation of the MSQL engine described in Section 5.3 to include the implementations of the corresponding functionality and components of the architectural extension shown in Figure 6.4. The implementation follows

```
<omdoc:omgroup>
 <omdoc:symbol kind="object" name="calculus1_diff">
  <omdoc:type system="simply_typed"
   xml:id="calculus1_diff_type">
   <om:OMA>
     <om:OMS cd="sts" name="mapsto"/>
     <om:OMA>
       <om:OMS cd="sts" name="mapsto"/>
       <om:OMS cd="setname1" name="R"/>
       <om:OMS cd="setname1" name="R"/>
     </om:OMA>
     <om:OMA>
       <om:OMS cd="sts" name="mapsto"/>
       <om:OMS cd="setname1" name="R"/>
       <om:OMS cd="setname1" name="R"/>
     </om:OMA>
    </om:OMA>
  </omdoc:type>
 </omdoc:symbol>
</omdoc:omgroup>
```

Figure 6.6: Declaration of variable *diff* in OMDoc

the same approach discussed in Section 5.3 by being based on the denotational semantics of the extension which formally defines the extended grammar of the language (see Section 6.2.

The implementation of the *Semantic Expression Evaluator* includes the corresponding functionality stated in the architecture. For instance, the *ASTtoOpenMath* class implements that part of the *Semantic Expression Evaluator* concerned with the conversion of the AST tree to OpenMath. It takes the AST of the *satisfy* expression and transforms it to the corresponding OpenMath XML representation.

The *ExpressionEvaluatorImpl* class has the *evaluateSatisfyExpr* method which implements that part of the *Semantic Expression Evaluator* concerned with accessing the Reasoner Interface with the semantic expression, the axioms, and the declarations as parameters and receives the result back from it. The *Semantic Expression Evaluator* accesses the Reasoner Interface either locally by means of the *ProverLocal* class or via a Web service by means of the *ProverServiceClient* (see Section 6.4.1).

The Reasoner Interface used is a component of the RISC ProofNavigator [114]. This component has the functionality to transform formulae (in OpenMath), axioms (in OpenMath), and declarations (in OMDoc) received from the *Semantic Expression Evaluator* to the corresponding format of the Automated Reasoner. The Automated Reasoner used is the Cooperating Validity Checker Lite (CVCL) [35].

The Knowledge Base is realized by a directory in the file system that includes axioms (in OpenMath) and declarations (in OMDOC) related to semantic queries. The *KnowledgeBaseReader* class implements the functionality needed by the *Semantic Expression Evaluator* to retrieve the axioms and declarations of the corresponding semantic query from the Knowledge Base.

This functionality of the *Semantic Expression Evaluator* is outlined by the following Java statements of the *evaluateSatisfyExpr* method of the *ExprEvalua-*

torImpl class:

```
OMObject omObject =
 ASTtoOpenMath.toOpenMath(satisfyExpr.getOMchild(),
                                          declaration);
 String semanticQuery =  OMObject.toString();
 String[] axioms = KnowledgeBaseReader.getAxioms();
 String[] declarations =
  KnowledgeBaseReader.getDeclarations();
 Prover prover = new Prover();
 boolean value =
  prover.prove(semanticQuery, axioms, declarations);
```

The complete API of the MSQL engine including its semantic extension is found in [13].

6.4 Querying in the Extended MSQL

In this section we demonstrate the extended API of the MSQL engine by two examples with respect to two different installations.

6.4.1 Sample Installations

We set up two installations which differ in how the Automated Reasoner is accessed.

- In the first installation, the MSQL engine accesses the Automated Reasoner through the Reasoner Interface available locally. The *ProveLocal* class includes the needed settings of the RISC ProofNavigator to invoke the CVCL running in a certain machine.
- In the second installation, the MSQL engine accesses the Automated Reasoner through the Reasoner Interface settings available as a Web service. We developed and deployed the *Prover* Web service (see Section 3.1.3) which includes the needed settings of the RISC ProofNavigator to invoke the CVCL running in a certain machine. The engine uses the *Prover* service through the *ProverServiceClient* class implemented local to the engine. It invokes the *Prover* service at its location sending to it the semantic expression, the axioms, and the declarations and receives from it the answer of the CVCL reasoner.

6.4.2 Using the extended API

The MSQL engine API functionality exposed to the user does not change with the extension. Only some internal functionality is changed (e.g. see the statements outlined in Section 6.3.2). There are two alternatives for using the API described in Section 5.4.2. In this section we outline one of these alternatives:

```
MsqlQuery msqlQuery = new MsqlQueryImpl();
MathBrokerConnection connection =
      msqlQuery.makeConnection(connectionProps);
```

```
ChildAST queryTree = msqlQuery.parseQuery(queryString);
Collection resultsCollection =
      msqlQuery.performQuery(queryTree, connection);
```

The *makeConnection* method initiates a connection to the registry according to the provided connection properties *connectionProps*. A received query (*queryString*) is parsed first using the *parseQuery* method resulting in the construction of the *queryTree* AST. The *performQuery* method takes care of performing the query on the registry-retrieved candidate documents returning the result(s) as a collection (*resultsCollection*) of documents. A complete client application including this outline is found in Appendix B.5.

6.4.3 Examples

We demonstrate the use of the MSQL API outlined above by the two use cases presented in Section 6.1.2 performed on each of the two installations.

Example 1

We have run the client application (see Appendix B.5) on the query of Use Case 1 (see Section 6.1) using the two installations. When the engine encounters the semantic expression, it reads the needed axioms and declarations from the specified location (the Knowledge Base). The engine, then, invokes the CVCL through the local settings of the RISC ProofNavigator in the first installation and through the *Prover* Web service settings of the RISC ProofNavigator in the second installation. In the two cases, the engine performed the query and returned as a result the MSDL document shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<monet:service name="RRISC"
   xmlns:monet="http://monet.nag.co.uk/monet/OpenMathDC">
  <monet:documentation>This is an implementation of the algorithm
    Risch. We use the mathb namespace to state expected performance
    of the concrete implementation wrt to its theoretical
    complexity measure.
  </monet:documentation>
  <monet:classification>
      <monet:problem href="http://risc.uni-linz.ac.at/mathb
        roker/RischIndefIntegration/indefinite-integration"/>
  </monet:classification>
  <monet:implementation href="http://risc.uni-linz.ac.at/ma
       thbroker/RischIndefIntegration/RImpl"/>
  <monet:service-interface-description href="http://perseus
   .risc.uni-linz.ac.at:8080/axis/services/SymbolicIntegration?wsdl"/>
  <monet:service-binding>
      <monet:map operation="symbint:Integrator:indefInt"
        problem-reference="indefinite-integration" action="exec"/>
      <monet:message-construction
        message-name="symbint:IndefIntRequest"
        message-part="in0" io-ref="f"/>
  </monet:service-binding>
  <monet:service-metadata/>
```

```
  <monet:broker-interface>
      <monet:service-URI/>
  </monet:broker-interface>
</monet:service>
```

Example 2

We have run the client application (see Appendix B.5) on the query of Use Case 2 (see Section 6.1) using the two installations. The query of Use Case 2 performs the same request as that of Use Case 1. But as mentioned earlier, it is formed based on the problem specification read from a given description rather than on the specification provided directly as a sub-expression in the query. As a consequence, the query includes the needed variable replacements. The resulting OpenMath representation of the satisfy expression is the same as that of the query in Use Case 1 (see Figure 6.5). The rest of the steps in the two installations are the same as in Example 1 and the same MSDL document is returned as a result.

6.5 Summary

The semantic extension of the Mathematical Services Query Language enables it to support semantic-based queries on the underlying semantical structures of mathematical service specifications. These semantical structures are represented in OpenMath; a standard for representing mathematical content. The extension adds a number of constructs to MSQL so that predicate logic formulae can be formed on the semantic information contained in the descriptions. The query engine of MSQL performs semantic-based queries with the help of an automated reasoner which takes predicate logic formulae, decides their validity, and returns the answer to the engine.

Chapter 7

Evaluation

In this chapter we give a brief evaluation of the framework by performing an experimental performance analysis and by comparing, according to some factors, the framework to some approaches in XML Querying and mathematical services.

7.1 Performance Analysis

In this section we present an experimental performance analysis of the framework. Particularly, we would like to find the effect of factors such as the framework configuration, the repository size, and the query type on the retrieval time of candidate documents from the repository of the registry and on the query evaluation time.

Our aim is not to measure the absolute retrieval and query evaluation time. Because this time would change considerably in an optimized version of this prototype of the framework. We aim to measure the relative increments/decrements in time when certain factors (such as the ones shown below) change. These increments/decrements can be expected to be the same in an optimized version of the framework.

7.1.1 Performance Factors

The analysis is carried considering the following factors:

1. **Framework Configuration**. We consider two configurations depending on the locations of the client application, the MSQL engine, and the registry:
   (a) *Single-Machine Configuration*. The client application, the MSQL engine, and the registry are located on the same machine.
   (b) *Network Configuration*. The client application and the MSQL engine are located on the same machine and the registry is located on another machine.
2. **Size of the Repository**. The number of MSDL documents classified under a given classification node are considered as representative repository size. We consider five sizes consisting of 1, 4, 16, 64, 256, and 1024 documents. We consider document structures such as those introduced in Chapter 3 and used

in Chapter 4, Chapter 5, and Chapter 6. An example is a problem description containing elements such as name, classification(s), input(s), output(s), precondition(s), and postcondition(s).

3. **Query Type**. We consider the following types of queries where */classificationScheme/classificationNode* represent the repository with different sizes:

   - **Query 1**: A simple query having the *some* construct and a simple *where* expression. It tests the simplest query that can be formed in MSQL.

     ```
     select some service
     from /classificationScheme/classificationNode
     where /service[@name = "name"]
     ```

   - **Query 2**: A query having an absolute path expression. It tests the effect of the *absolute path* expression on the evaluation time of the query.

     ```
     select some problem
     from /classificationScheme/classificationNode
     where
       /problem/body/post-condition/om:OMOBJ/
        om:OMA/om:OMA/om:OMS[1][@name = "eq"]
     ```

   - **Query 3**: A query having a relative path expression. It tests the effect of the *relative path* expression on the evaluation time of the query.

     ```
     select some problem
     from /classificationScheme/classificationNode
     where
       //om:OMA/om:OMS[1][@name = "eq"]
     ```

   - **Query 4**: A query having a single (un-nested) *every quantifier*. It tests the effect of the *quantified* expression on the evaluation time of the query.

     ```
     select every problem
     from /classificationScheme/classificationNode
     where
      every $p in /problem satisfies
         $p/om:OMOBJ/om:OMA/om:OMS[@cd = "sts"] and
         $p/om:OMOBJ/om:OMA/om:OMS[@name = "mapsto"]
     ```

   - **Query 5**: A query having a nested *every quantifier*. It tests the effect of the nested *quantified* expression on the evaluation time of the query.

     ```
     select every problem
     from /classificationScheme/classificationNode
     where
      every $p in /problem satisfies
     ```

```
every $s in $p//signature satisfies
  $s/om:OMOBJ/om:OMA[/om:OMS/@cd = "sts"]
```

- **Query 6**: A query having the *doc* function in its *let* expression. It tests the effect of the *(*doc) function which causes an additional retrieval of a specific document from the repository on the evaluation time of the query.

```
select every service
from /classificationScheme/classificationNode
where
 let $d := doc(//implementation/@href) in
 not //classification[empty(/problem)] and
 $d//hardware[contains(@name, "perseus")]
```

- **Query 7**: A semantic query having nested expression, *let* expression, *doc* function, and *satisfy* expression. The query reads the axioms and the variable declarations needed for the reasoning process, also it needs to send the query together with the axiom and the declarations to the reasoner through the reasoner interface (see Section 6.3.1). It tests the effect of multiple (complex) expressions including the *satisfy* expression on the evaluation time of the query.

```
select some service
from /classificationScheme/classificationNode
where let $p:= doc(//problem/@href) in
      let $a:= $p//input/@name,
          $b:= $p//output/@name,
         $ta:= $p//input/signature/OMOBJ,
         $tb:= $p//output/signature/OMOBJ,
       $post:= $p//post-condition/OMOBJ in
 (typematch(oma(oms:sts:mapsto(oms:setname1:R,
                      oms:setname1:R)), $ta)) and
 (typematch($tb, oma(oms:sts:mapsto(oms:setname1:R,
                           oms:setname1:R)))) and
 (satisfy(ombind(oms:quant1:forall
  [omvar:$a@(oms:sts:type, $ta),
   omvar:$b@(oms:sts:type, $tb)]
  oma(oms:logic1:implies, $post,
     oma(oms:relation1:eq,
       oma(oms:calculus1:diff, omv:$b), omv:$a)))))
```

7.1.2 Measurements

We measured the publishing time and the querying time. The measurements were repeated three times. The obtained time values are close to each other, therefore we drop the lowest and the highest values and take the remaining value.

	Repository Size in Number of Documents					
	1	4	16	64	256	1024
$T_{single\ machine}$	6.392	14.778	35.842	116.546	453.475	1747.341
$T_{network}$	7.782	16.068	38.966	113.95	431.324	1708.989

Table 7.1: Publishing Time (measured in seconds) for Different Repository Sizes on Single-Machine and Network Configurations

Publishing Time

The *publishing time* includes the time to connect to the registry, the time to check if the object to be published already exist in the registry (therefore needs to be updated), the time to extract metadata such as the name, the classification(s), and association(s) of the object from the description (document), the time to create the classification(s), the associations(s), and update them if they exist, and the time to store the document in the repository of the registry.

Querying Time

For the querying-related time, we measure the following types of time that are most relevant to the factors above.

- The *Total Time* (T_{total}) which is equal to the time starting from parsing the query to getting the result of the query.
- The *Retrieval Time* ($T_{retrieval}$) which is related to the size of the repository and the type of MSDL documents to be retrieved (in an MSQL query this is represented by the *entity* construct of the *select* clause and by the *from* clause). It is the time taken to retrieve the candidate document(s) measured from issuing the retrieval request (i.e. the registry query) to receiving the last document.
- The *evaluation time* ($T_{evaluation}$) which is related to the evaluation of the *where* expression on *every/some* retrieved document. It is equal to the difference between the *total time* and the *retrieval time*.

$$T_{evaluation} = T_{total} - T_{retrieval}$$

7.1.3 Publishing Time on Single-Machine and Network Configurations

In this section we measure the time to publish collections of 1, 4, 16, 64, 256, and 1024 MSDL documents stored in different files. Each collection is published under a given classification node in the registry so that they represent the different repository sizes used later in the query-related performance measurements.

Table 7.1 shows the time taken to publish the different numbers of MSDL documents on the *single-machine configuration* (a machine with 2400 MHz processor) and *network configuration* (two machines with 2400 MHz processors connected by a 100 Mbps network) of the framework.

The chart in Figure 7.1 depicts the *publishing time* shown in Table 7.1. Although it is expected that the *publishing time* on the *network configuration* is

longer than that on *single-machine configuration*, the two times are close to each other with the time on *single-machine configuration* a bit longer for the 256 and 1024 sizes. This apparently is because in the *single-machine configuration*, the client application, the MSQL machine and the registry are all installed on the same machine which imposes an extra overhead on the processor. Additionally the high speed of the network contributes to minimizing the time in the case of the *network configuration*.

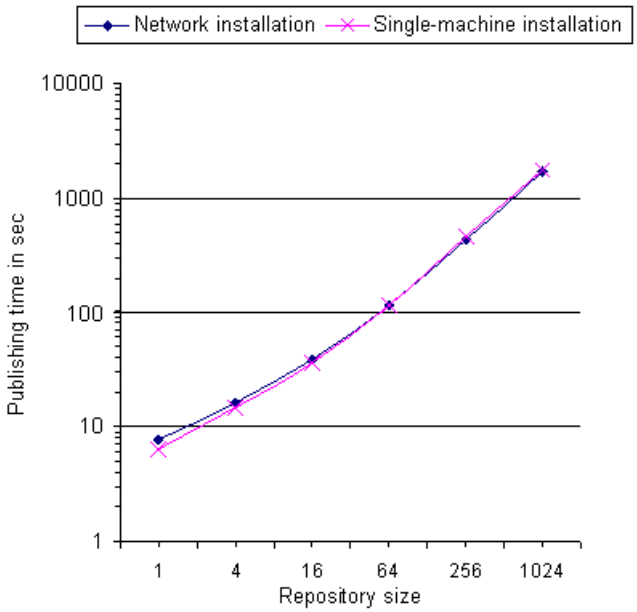


Figure 7.1: Publishing Time for Single-Machine and Network Configurations

This slightly large publishing time is due to the inefficiency of the ebXML registry reference implementation on which we built our mathematical registry.

The *publishing time* is not crucial factor in the performance of the framework since publishing is a demanding one time operation with maybe a small number of updates to the published objects.

7.1.4 Querying Time on a Single-Machine Configuration

We present the time results of performing queries 1 to 7 on the different repository sizes and considering the *single-machine configuration* of the framework where the client application, the MSQL engine and the registry are all running on the same machine with 2400 MHz processor. Table 7.2 shows the *total time*, the *retrieval time* and the *evaluation time* for each query.

The *retrieval time* and the *evaluation time* for queries 1 to 7 are depicted on the charts of Figure 7.2 and Figure 7.3. We depict a single *retrieval time* because all retrieval times have almost the same chart shape with values that are close to each other and we are more concerned with the *evaluation time*. The slightly large *retrieval time* is not caused by the MSQL engine since this time is spent at the registry. It is caused by the inefficiency of the ebXML registry implementation

		Repository Size in Number of Documents					
		1	4	16	64	256	1024
Query 1	T_{total}	2.072	2.536	4.390	11.437	40.423	178.111
	$T_{retrieval}$	1.993	2.517	4.361	11.385	40.289	177.603
	$T_{evaluation}$	0.079	0.019	0.029	0.052	0.134	0.508
Query 2	T_{total}	1.916	2.491	4.197	10.743	39.636	168.162
	$T_{retrieval}$	1.906	2.473	4.141	10.659	39.237	166.390
	$T_{evaluation}$	0.010	0.018	0.056	0.084	0.399	1.772
Query 3	T_{total}	1.910	2.395	4.146	10.678	39.186	163.519
	$T_{retrieval}$	1.902	2.372	4.141	10.671	39.176	163.404
	$T_{evaluation}$	0.008	0.023	0.005	0.007	0.010	0.115
Query 4	T_{total}	1.903	2.373	4.225	10.643	39.489	164.785
	$T_{retrieval}$	1.894	2.358	4.196	10.590	39.142	163.049
	$T_{evaluation}$	0.009	0.015	0.029	0.053	0.347	1.736
Query 5	T_{total}	1.931	2.367	4.160	10.677	39.568	162.746
	$T_{retrieval}$	1.924	2.353	4.122	10.609	39.025	162.691
	$T_{evaluation}$	0.007	0.014	0.038	0.068	0.543	0.055
Query 6	T_{total}	1.901	2.757	4.508	10.594	39.398	162.838
	$T_{retrieval}$	1.894	2.346	4.102	10.590	38.978	162.351
	$T_{evaluation}$	0.007	0.411	0.406	0.004	0.420	0.487
Query 7	T_{total}	2.994	3.357	6.463	16.287	52.545	185.260
	$T_{retrieval}$	2.572	2.759	5.096	12.699	42.437	165.395
	$T_{evaluation}$	0.422	0.598	1.367	3.588	10.108	19.865

Table 7.2: Total, Retrieval and Evaluation Time (measured in seconds) for Queries 1 to 7 on a Single-Machine Configuration

which servers as the basis of the mathematical registry.

The resulting time values shown on Table 7.2 and depicted on the Figure 7.2 and Figure 7.3 reveal the following observations:

- For queries 1 to 7, the *evaluation time* increases as the size of the repository increases with some exceptions where the *evaluation time* of the same query takes less time on some of the bigger repository sizes than on some of the smaller repository sizes. These exceptions are caused by the presence of the *some* construct where the query terminates at the first document that satisfies the expression being evaluated. For example *Query 3* has an *evaluation time* on, e.g., the 64 document repository size less than that of the 4 document repository size. The same is applied to Query 6 where its *evaluation time* on, e.g., the 64 document repository size is less than that of the 16 document repository size.
- Query 2 (with the *absolute path* expression) has a smaller *evaluation time* than Query 3 (with the *relative path* expression)
- Query 5 (with the nested *quantifiers*) has a bigger *evaluation time* than Query 4 (with the un-nested path expression) with some exceptions, e.g., on the 1024 documents repository size.
- Query 6 (with the *doc functions*) has bigger *evaluation time* than queries 1 to 5 with the exception of the 64 document size repository. This is because of the *doc* function which requires additional time for the additional document retrieval from the repository.

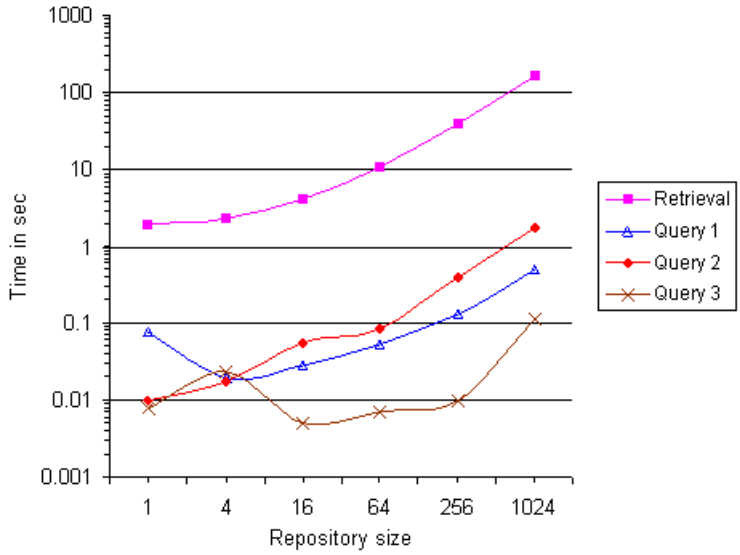


Figure 7.2: Retrieval and Evaluation Time for Queries 1 to 3 on a Single-Machine Configuration

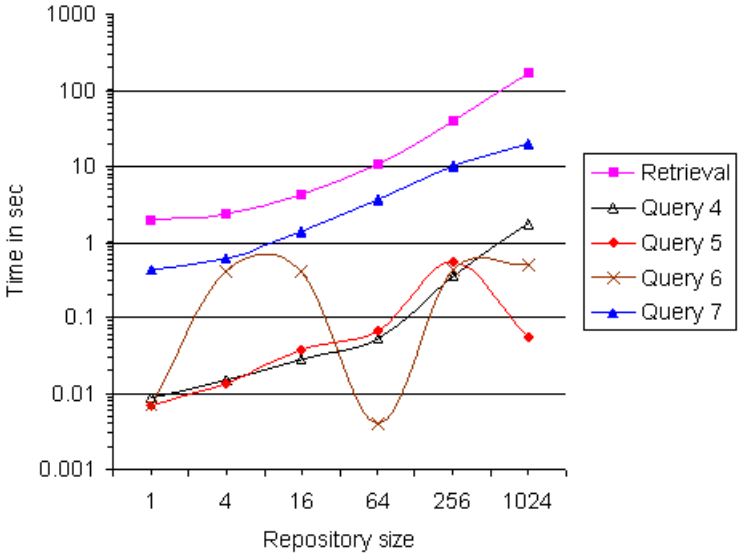


Figure 7.3: Retrieval and Evaluation Time for Queries 4 to 7 on a Single-Machine Configuration

		Repository Size in Number of Documents					
		1	4	16	64	256	1024
Query 1	T_{total}	2.024	2.496	4.435	11.387	40.679	164.811
	$T_{retrieval}$	1.955	2.479	4.408	11.331	40.335	164.400
	$T_{evaluation}$	0.069	0.017	0.027	0.056	0.344	0.411
Query 2	T_{total}	1.902	2.417	4.209	10.842	39.927	166.947
	$T_{retrieval}$	1.894	2.399	4.154	10.762	39.495	164.891
	$T_{evaluation}$	0.008	0.038	0.055	0.080	0.432	2.056
Query 3	T_{total}	1.906	2.368	4.209	10.698	39.752	163.170
	$T_{retrieval}$	1.900	2.346	4.125	10.692	39.745	162.820
	$T_{evaluation}$	0.006	0.022	0.084	0.006	0.007	0.350
Query 4	T_{total}	1.896	2.372	4.157	10.720	39.732	166.688
	$T_{retrieval}$	1.888	2.356	4.126	10.664	39.350	164.614
	$T_{evaluation}$	0.008	0.016	0.031	0.066	0.382	2.074
Query 5	T_{total}	1.895	2.370	4.176	10.727	39.871	163.837
	$T_{retrieval}$	1.888	2.351	4.123	10.656	39.310	163.776
	$T_{evaluation}$	0.007	0.019	0.053	0.071	0.561	0.061
Query 6	T_{total}	1.897	2.756	4.525	10.779	40.181	163.856
	$T_{retrieval}$	1.890	2.343	4.123	10.674	39.770	163.231
	$T_{evaluation}$	0.007	0.412	0.402	0.105	0.411	0.625
Query 7	T_{total}	3.490	3.916	6.214	17.111	57.157	210.524
	$T_{retrieval}$	2.571	3.398	5.125	12.731	42.696	167.209
	$T_{evaluation}$	0.818	0.518	1.089	4.380	14.461	43.315

Table 7.3: Total, Retrieval and Evaluation Time (measured in seconds) for Queries 1 to 7 on a Network Configuration

- Query 7 has the biggest *evaluation time* due to the *satisfy* expression which takes additional time spent in the reasoning process.

7.1.5 Querying Time on a Network Configuration

We present the time results of performing queries 1 to 7 on the different repository sizes and considering the *network configuration* of the framework where the client application and the MSQL engine run on one machine while the registry runs on another machine both with a 2400 MHz processor and connected by a 100 Mbps network. Table 7.3 shows the *total time*, the *retrieval time* and the *evaluation time* for each query.

The *retrieval time* and the *evaluation time* for queries 1 to 7 on the *Network configuration* of the framework are depicted on the charts of Figure 7.4 and Figure 7.5. We depict a single *retrieval time* because all retrieval times have almost the same chart shape with values that are close to each other and we are more concerned with the *evaluation time*.

The same observations mentioned about the resulting time values of the single-machine configuration (see Section 7.1.4) are applied here to the resulting time values shown on Table 7.3 and depicted on the Figure 7.4 and Figure 7.5.

An additional observation worth mentioning here is about the *retrieval time*. Although it is expected that the *retrieval time* on the *network configuration* is longer than that on *single-machine configuration*, the two times are close to each other with the time on *single-machine configuration* a bit longer for the 1024 docu-

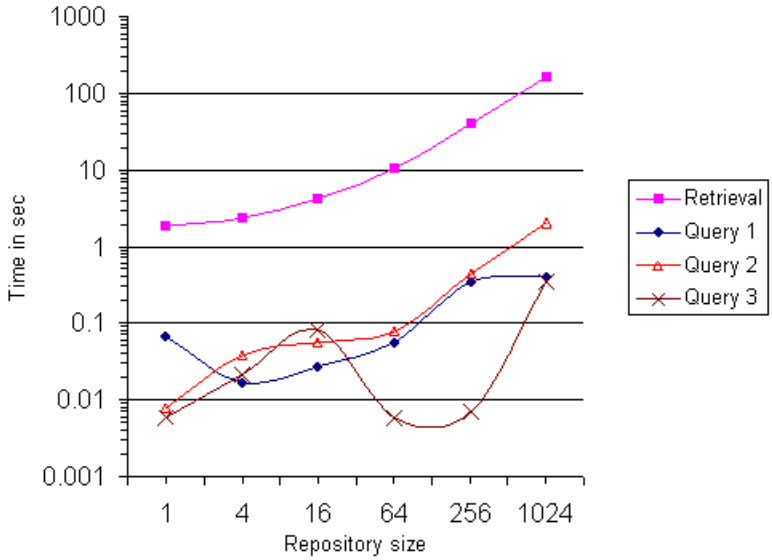


Figure 7.4: Retrieval and Evaluation Time for Queries 1 to 3 on a Network Configuration

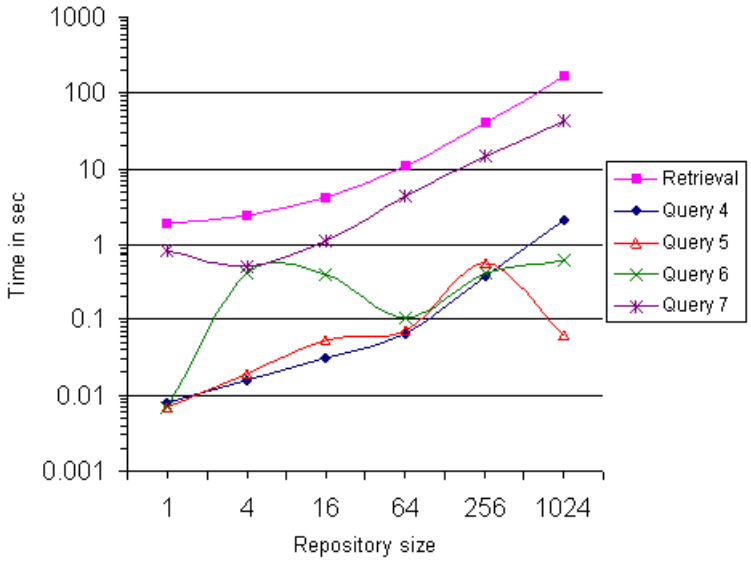


Figure 7.5: Retrieval and Evaluation Time for Queries 4 to 7 on a Network Configuration

ments repository size. This apparently is due to the fact that in the *single-machine configuration*, the client application, the MSQL machine and the registry are all installed on the same machine which imposes an extra overhead on the processor. Additionally the high speed of the network contributes to minimizing the time in the case of the *network configuration*.

7.1.6 Conclusion

Based on the obtained results of the publication time and the querying time for the two configurations, we conclude the following:

- The *publishing time* of the different repository sizes is much longer than the querying time including the retrieval and evaluation time. The *publishing time* is not a crucial factor in the performance since publishing is not expected to be performed as often as querying.
- Most of the *total time* is spent in the retrieval of the candidate documents from the repository.
- The *retrieval time* is an order of magnitude larger than the *evaluation time* due to the inefficiency of the ebXML registry reference implementation serving as the basis of our mathematical registry. An optimized version of the framework should include the query evaluation within the registry (or choose another basis for the registry implementation).
- As the size of the repository increases, the retrieval time respectively the total time increases. When the size of the repository increased by a factor of four, the retrieval time also increases by a factor of four.
- The type of entity affects the retrieval time since the type of each document in the repository is checked to see if it is the same as that specified in the *entity* construct of the *select* clause of the query.
- The query type, specifically the expression part, does not affect the retrieval time.
- The *evaluation time* is affected by the *every/some* construct of the *select* clause of the query. If *every* is used in the query, then the *where* expression of the query is executed on every document in the collection of the candidate documents. If *some* is used then the query terminates at the first document that satisfies the *where* expression. That is why, e.g., Query 6 takes less time with the 256 size repository than with the 4 or 16 size repository. Therefore the *evaluation time* is affected by the size of the repository depending on the *every/some* construct.
- The *evaluation time* is affected by the type of the query (i.e., by the *where* expression of the query). Queries with (nested) *quantifier* expressions, e.g. Query 4 and Query 5, often take more time than other types of queries. Queries with *doc* function, e.g. Query 6, which require additional document retrieval from the repository often take more time than other types of queries.
- Semantic queries, e.g. Query 7, take longer than syntactic queries because extra time is spent in converting the semantic expression to OpenMath, reading the axioms and variable declarations, using the reasoner interface, and reasoning about the semantic expression by the reasoner.

- In a *network configuration* an extra time is needed for the communication with the registry to retrieve the documents. This extra time is counted as part of the retrieval time. Comparing the retrieval time in the *single configuration* and the *network configuration* gives a small different of not more than 14 seconds for retrieving the contents of the 1024 document repository.

7.2 Comparison to other Approaches

In this section we briefly compare our framework to some related approaches. Particularly, we compare MSQL to the XQuery language according to factors such as purpose, expressions, target and resulting documents, and supported quires. We compare our publishing and discovery approach to that of the MONET project according to factors such as architecture, service description, and service discovery.

7.2.1 Comparison to XQuery

XQuery [22] (see Section 2.3.2) is a general purpose XML query language for querying collections of XML documents. We highlight some of the differences between MSQL and XQuery according to the following factors:

- **Purpose**: XQuery is a general purpose XML query and processing language that is meant to be used in any application area that stores, manipulates and retrieves XML data. MSQL, though can be considered (or adopted) as a general purpose query language, is meant to be a domain-specific query language that addresses the discovery needs of the presented mathematical framework. Although XQuery has a rich functionality, it cannot address some of our requirements such as dealing with classification schemes and types of objects stored in the registry.

- **Expressions**: MSQL (as light-weight) has a minimal set of expressions that are necessary to address the contents of the target MSDL documents. It can be said that these expressions are a subset of those of XQuery.

- **Target Documents**: XQuery is designed to be broadly capable of dealing with many sources of XML documents. MSQL, though can deal with such XML documents but not as broadly as XQuery, is designed to deal with the schema-based MSDL documents and the OpenMath-based mathematical representations within these documents (see Semantic Querying below).

- **Resulting Documents**: XQuery is capable of transforming target documents and generate XML structures as results. MSQL return MSDL documents without manipulating or transforming them. This is because the goal of MSQL is to return whole documents that can be further processed by other applications.

- **Semantic Querying**: XQuery does not address semantic content of its underlying documents nor supports semantic-based querying. MSQL has the constructs to address and represent sematic content using predicate logic and supports semantic-based querying over predicate logic using a reasoner.

The XQuery recommendation [22] has a number of existing implementations such as [50, 138, 61] that implements parts of the specified language. A possible performance comparison between XQuery in these implementations and MSQL would consider only the expression part since the query structures of the two languages are quite different. Based on our investigations, there are no performance measurements performed for these implementations that can serve as a basis for such a comparison.

7.2.2 Comparison to MONET

The MONET project [87] developed an architecture [121] (see Section 2.5.2) for brokering mathematical web services. We compare our framework to that of MONET according to the following factors:

- **Architecture**: The MONET architecture has the following components: the *server*, the *broker*, and the *client*. The server registers services with the *broker*. The *client* queries the *broker* for specific services. The *broker* executes a query by looking for available services and passing the client's request to them. Services perform the request and returns the results to the *client*.

  Our framework has the following components: the service *provider*, the *registry*, the *MSQL engine*, and the *client*. The service *provider* deploys a service to a Web server and publishes it to the *registry*, a *client* discovers the service either by directly querying the registry, or for more efficient results by using the syntactic- and semantic-based querying of MSQL. Based on the query result, the client access the service at the Web server.

- **Service Description**: MONET defines a set of ontologies written in OWL to model service descriptions. These ontologies are ontological conversions of MSDL descriptions. MONET uses two classes of ontologies: those describe models internal to MONET (e.g., problem and software ontologies) and those describe models external to MONET (e.g., classification ontologies). Individual ontologies of both classes are imported into one MONET ontology. When a service is submitted to the MONET broker, its description is presented in MSDL. This description is transformed to the OWL Abstract Syntax [19] by means of an XSLT stylesheet.

  In our case services are described directly in MSDL and when published in the registry, they are classified according to classification schemes imposed by the information model of the registry.

  Although in our case we submitted and used classification schemes in the registry to classify services, the use of ontologies for service classification in the registry is possible and can coexist with the conventional classification schemes (see the recommendation in Chapter 8).

- **Service Discovery**: Service matching in MONET is performed by submitting a query to the *Instance Store* (a component within the *broker* architecture used for matching queries to appropriate services) in the form of an OWL description. *Instance Store* answers the query by using a combination of Description Logic reasoning and database queries. MSQL queries (syntactic- and/or semantic-based) are formed on descriptions stored on the registry

according to given classifications. semantic-based queries uses the CVCL reasoner to reason about expressions expressed as predicate logic formulae.

The reasoning process in the case of MONET is based on a restricted form of first order logic which is more tractable for automated reasoning but strictly less expressive. In our semantic queries, we use full predicate logic which is a highly expressive language.

In an extension of MONET, a matching-based discovery approach [93] (see Section 2.5.3) performs matchmaking between representations of tasks (client requests) and capabilities (service descriptions) by applying a similarity measure calculated based on matching of capabilities or tasks. This matching process used in the discovery is ultimately based on the syntactic similarity traced between tasks and capabilities. In our case, the decision is based on logical implications between statements extracted from descriptions, which is strictly more general.

To our knowledge, no performance evaluation has been performed for the implemented MONET architecture and its service discovery components.

7.3 Summary

In this chapter we presented a brief evaluation of the framework based on an experimental performance analysis and a comparison with related approaches. The analysis considers the effect of factors such as the framework configuration, the size of the queried repository, and the type of query on the performance of the publishing and discovery in the framework. The comparison to XQuery is based on factors such as language purpose, expressions, target and resulting documents. The comparison to MONET approach is based on factors such as architecture, service description and discovery.

Chapter 8

Conclusions

We have developed a framework for publishing and discovering mathematical Web services. The framework consists of a registry for publishing mathematical service descriptions, a content-based query language for syntactically as well as semantically querying registry-published service descriptions.

The Registry framework is based on the ebXML registry standard and its reference implementation. We extended the ebXML registry information model to include the Mathematical Services Description Language (MSDL) information model and extended the ebXML *Registry Service* management interfaces to include mathematical object management interfaces. The implementation of these extensions on top of the ebXML Registry Reference implementation (ebXMLrr) provides a set of functionalities for processing, classifying, associating, publishing, and querying mathematical service descriptions expressed in MSDL.

Since the querying facilities of the registry do not support content-based querying, we have developed the content-based Mathematical Services Query Language (MSQL) to support queries at the syntactical structure of service descriptions. MSQL provides the functionality to interface to the mathematical registry and retrieve service descriptions on which queries are performed.

MSQL is designed as a light-weight query language having a minimal set of constructs necessary to interface to the registry and retrieve candidate MSDL documents (descriptions), to compose query expressions that address the description structures, to filter these descriptions , and to return the filtered descriptions as the results of the query.

The formal definition of MSQL is based on denotational semantics; a technique for rigorously specifying the semantics of a programming language. Denotational semantics specify semantic functions that map elements of MSQL syntactic domains to elements of its semantic domains. This gives a clear meaning to MSQL constructs and therefor provides a correct reference for implementation. The core component of the implementation is the MSQL engine which receives queries, performs them, and returns MSDL documents as results. This functionality of the MSQL is exposed through its API.

The MSQL engine returns whole MSDL documents as query results. This is so because one goal of the framework is to get such documents for further processing maybe by a software agent acting in behalf of a user (see below).

The Mathematical Services Query Language (MSQL) supports queries on the

syntactical structure of the Mathematical Services Description Language (MSDL) descriptions. However, mathematical objects are semantically rich and MSQL does not address the semantic layer of MSDL descriptions. The semantic extension of MSQL supports semantic-based queries on the underlying semantical structures of service descriptions. These semantical structures are represented in OpenMath; a standard for representing mathematical content. The extension adds a number of constructs to MSQL so that predicate logic formulae can be formed on the semantic information contained in the descriptions. The query engine of MSQL performs semantic-based queries with the help of an automated reasoner which takes predicate logic formulae, decides their validity, and returns the answer to the engine.

As for possible improvements and extensions of the framework, the syntax of MSQL and the presented query examples reveal an apparent difficulty in forming queries on target documents. Such a difficulty is alleviated only partially by the fact that MSQL has a relatively small number of constructs for forming queries and by the fact that its queries operate on documents that possess common structures imposed by a schema. Thus, a user-friendly tool for forming queries on target MSDL documents should be developed.

Another extension to the framework contains service compositions: when a client submits a service request, a broker agent determines suitable service compositions satisfying the client request and returns the description of a composition rather than that of a single service. To find the suitable candidate services, the agent might form MSQL queries based on information contained in the client request, send them to the MSQL engine, and make composition decisions based on the results returned by the MSQL engine.

Another promising extension is the development of the broker agent that receives a user's computational requirements, uses the framework to discover suitable services decides which service(s) would satisfy the user computational requirements, invokes it/them to do the computation, gets the results, and returns them to the user.

Finally, It is worth mentioning that during the runtime of the projects where this dissertation is carried out, ontology-based approaches to service description and discovery have gained popularity in the Web community. Thus alternative description and classification forms based on ontologies (possibly coexisting with our descriptions and classification schemes) may be investigated.

Bibliography

[1] ISO/IEC 14977:1996(E). Information Technology – Syntactic Metalanguage – Extended BNF. ISO/IEC, 1996. See http://www.cl.cam.ac.uk/ mgk25/iso-14977.pdf.

[2] The ActiveMath Learning Environment, September 2004. See http://www.activemath.org/.

[3] Algae RDF Query Language. W3C, June 2004. See http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[4] Annotea Project. W3C, January 2004. See http://www.w3.org/2001/Annotea/.

[5] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, Ferruccio Guidi, and Irene Schena. Mathematical Knowledge Management in HELM. *Ann. Math. Artif. Intell.*, 38(1-3):27–46, 2003.

[6] Andrea Asperti and Stefano Zacchiroli. Searching Mathematics on the Web: State of the Art and Future Developments. In *New Developments in Electronic Publishing of Mathematics*, Stockholm, Sweden, June 25 – 27 2004.

[7] Apache eXtensible Interaction System (axis). The Apache Web Services Project, March 2006. http://ws.apache.org/axis/.

[8] Franz Baader et al, editor. *The Description Logic Handbook — Theory, Implementation and Applications*. Cambridge University Press, Cambridge, UK, 2003.

[9] Grzegorz Bancerek and P. Rudnicki. Information Retrieval in MML. Proceedings of the Second International conference on Mathematical Knowledge Management, Bertinoro, Italy, February 2003. LNCS 2594, pp119-131.

[10] Rebhi Baraka. A Foundation for a Mathematical Web Services Query Language: A Survey on Relevant Query Languages and Tools. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2004. See http://apache.risc.uni-linz.ac.at/internals/ActivityDB/publications/download/risc_2036/04-18.ps.gz.

[11] Rebhi Baraka. MathBroker Registry API. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2004. See http://poseidon.risc.uni-linz.ac.at:8080/results/registry/MBregistryAPI/.

[12] Rebhi Baraka. Mathematical Services Query Language: Design, Formalization, and Implementation. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2005. See ftp://ftp.risc.uni-linz.ac.at/pub/techreports/.

[13] Rebhi Baraka. Mathematical Services Query Language (MSQL) API. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2005. See http://poseidon.risc.uni-linz.ac.at:8080/results/msql/doc/index.html.

[14] Rebhi Baraka, Olga Caprotti, and Wolfgang Schreiner. A Web Registry for Publishing and Discovering Mathematical Services. In *Proceedings of the IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE 2005), 29 March - 1 April 2005, Hong Kong, China*, pages 190–193. IEEE Computer Society.

[15] Rebhi Baraka, Olga Caprotti, and Wolfgang Schreiner. A Registry Service as a Foundation for Brokering Mathematical Services. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, February 2004. See ftp://ftp.risc.uni-linz.ac.at/pub/techreports/2004/04-13.ps.gz.

[16] Rebhi Baraka and Wolfgang Schreiner. Querying Registry-Published Mathematical Web Services. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA 2006), 18-20 April 2006, Vienna, Austria*, pages 767–772. IEEE Computer Society.

[17] Rebhi Baraka and Wolfgang Schreiner. Semantic Querying of Mathematical Web Service Descriptions. In *Proceedings of the Third International Workshop on Web Services and Formal Methods (WS-FM 2006), 8-9 September 2006, Vienna, Austria*. Springer. To appear.

[18] Clark W. Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker Category B. In *Proceedings of 16th International Conference on Computer Aided Verification*, Boston, MA, USA, July 13-17, 2004. Springer.

[19] Sean Bechhofer, Peter F. Patel-Schneider, and Daniele Turi. OWL Web Ontology Language Concrete Abstract Syntax. Technical report, The University of Manchester, UK, December 2003. See http://owl.man.ac.uk/2003/concrete/latest/.

[20] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. W3C Candidate Recommendation, World Wide Web Consortium, November 2005. http://www.w3.org/TR/xpath20/.

[21] Mathematical Web Services. W3C note, August 2003. See http://www.w3.org/Math/Documents/Notes/services.xml.

[22] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation, World Wide Web Consortium, November 2005. See http://www.w3.org/TR/xquery/.

[23] Joram Borenstein and Joshua Fox. Semantic Discovery for Web Services. *Web Services Journal*, 3(4), 2003.

[24] Business Process Execution Language for Web Services version (BPEL4WS), February 2005. See http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[25] Tim Bray et al. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, February 2004. See http://www.w3.org/TR/2004/REC-xml-20040204/.

[26] Dan Brichley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, February 2004. See http://www.w3.org/TR/rdf-schema/.

[27] Stephen Buswell, Olgo Caprotti, David Carlisle, Mike Dewar, Marc Gaëtano, and Michael Kohlhase. The OpenMath Standard (v 2.0). The OpenMath Society, June 2004. See http://www.openmath.org/cocoon/openmath/index.html.

[28] Olga Caprotti, Arjeh M. Cohen, and Manfred Riem. Java Phrasebooks for Computer Algebra and Automated Deduction. *SIGSAM Bulletin*, 34(2):33–37, 2000.

[29] Olga Caprotti and Wolfgang Schreiner. Mathematical Software as Web Services. In *Conference Poster. MathML International Conference*, Chicago, Illinois, June 28 – June 30, 2002.

[30] Olga Caprotti and Wolfgang Schreiner. Towards a Mathematical Service Description Language. In *International Congress of Mathematical Software ICMS 2002*, Bejing, China, August 17–19, 2002. World Scientific Publishing, Singapore.

[31] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An xml query language for heterogeneous data sources. In Dan Suciu and Gottfried Vossen, editors, *WebDB (Selected Papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.

[32] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft, April 2005. See http://www.w3.org/TR/xpath-datamodel/.

[33] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, June 2005. See http://www.w3.org/TR/xquery-semantics/.

[34] The Coq Proof Assistant, September 2004. See http://coq.inria.fr/.

[35] Cooperating Validity Checker Lite (CVCL). Leland Stanford Junior University and New York University, March 2006. See http://chicory.stanford.edu/CVCL.

[36] I. Dahn and A. Asperti. Mathematical Knowledge Management and Searchability. Deliverable D5.4, 2001. See http://monet.nag.co.uk/mkm/MKMNetTN-D5-4.pdf.

[37] DAML+OIL Web Ontology Language. W3C, March 2001. See http://www.w3.org/Submission/2001/12/.

[38] Mike Dewar, David Carlisle, and Olga Caprotti. Description Schemes for Mathematical Web Services. In *EuroWeb 2002: The Web and the Grid: From e-Science to e-Business*, Oxford, UK, December 2002. British Computer Society Electronic Workshops in Computing.

[39] Asuman Dogac, Yildiray Kabak, and Gokce Laleci. Enriching ebXML Registries with OWL Ontologies for Efficient Service Discovery. In *RIDE*, pages 69–76. IEEE Computer Society, 2004.

[40] Document Object Model (DOM). W3C, September 2004. See http://www.w3.org/DOM/.

[41] Andreas Duscher. An Execution Environment for Mathematical for Services based on WSRF and WS-BPEL. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, March 2006.

[42] ebXML Registry Information Model Schema, April 2002. See http://www.oasis-open.org/committees/regrep/documents/2.1/schema/.

[43] ebXML Registry Information Model v2.0. OASIS, December 2001. See http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebRIM.pdf.

[44] ebXML Registry Reference Implementation Project (ebxmlrr), April 2006. See http://ebxmlrr.sourceforge.net.

[45] ebXML Registry Services Specification v2.0. OASIS, April 2002. See http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebRS.pdf.

[46] ebXML Registry Services Specification v3.0. OASIS, April 2002. See http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebRS.pdf.

[47] Richard J. Fateman. Network Servers for Symbolic Mathematics. In *International Symposium on Symbolic and Algebraic Computation ISSAC 1997*, Maui, Hawaii, USA, July 21–23, 1997. ACM Press.

[48] Dieter Fensel and Christoph Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[49] Richard Fikes, Patrick Hayes, and Ian Horrocks. OWL–QL – A Language for Deductive Query Answering on the Semantic Web. Knowledge Systems Laboratory, 2003. See http://ksl.stanford.edu/KSL_Abstracts/KSL-03-14.html.

[50] Galax , March 2005. See http://www.galaxquery.org/.

[51] Guide to Available Mathematical Software. National Institute of Standards and Technology (NIST), May 2006. http://gams.nist.gov/.

[52] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4*, 2005. `See http://www.gap-system.org`.

[53] GAP Phrasebook. Research Institute for Applications of Computer Algebra (RIACA), Technical University of Eindhoven, Eindhoven, Holand, May 2006. See http://www.riaca.win.tue.nl/products/gap/.

[54] Google, February 2006. See http://www.google.com.

[55] Volker Haarslev and Ralf Moller. Description of the RACER system and its applications. In *Automated reasoning: First International Joint Conference, IJCAR 2001*, Siena, Itally, June 18–23, 2001. volume 2083 of Lecture Notes in Artificial Intelligence, New York, NY, USA, 2001. Springer Verlag Inc.

[56] UDDI and ebXML Registry: A Co-Existence Paradigm. XML.org, March 2003. http://lists.xml.org/archives/xml-dev/200304/pdf00000.pdf.

[57] HELM: Hypertextual Electronic Library of Mathematics, September 2004. See http://helm.cs.unibo.it/.

[58] James Hendler and Deborah L. McGuinness. The DARPA Agent Markup Language (DAML). *IEEE Intelligent Systems*, 15(6):67–73, November/December 2000.

[59] James A. Hendler and Deborah L. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6):72–73, 2000.

[60] Instance Store - Database Support for Reasoning over Individuals. The University of Manchester, 2002. See http://instancestore.man.ac.uk/instancestore.pdf.

[61] IPSI-XQ – The XQuery Demonstrator, September 2005. See http://www.ipsi.fraunhofer.de/oasys/projects/ipsi-xq/index_e.html.

[62] Database Language SQL Part 4: Persistent Stored Modules (SQL/PSM). ISO, December 1996.

[63] Java Architecture for XML Binding (JAXB). Sun microsystems, April 2004. http://java.sun.com/xml/jaxb/.

[64] Java API for XML Registries (JAXR) v0.9. Sun microsystems, April 2002. See http://java.sun.com/xml/jaxr/.

[65] Jena - A Semantic Web Framework for Java. Sourceforge, September 2004. See http://jena.sourceforge.net/.

[66] Journal of Formalized Mathematics. Institute of Computer Science, University of Bialystok. See http://mizar.org/JFM/.

[67] J/Link. Wolfram Research, February 2006. See http://www.wolfram.com/solutions/mathlink/jlink/.

[68] Michael Kay. XSL Transformations (XSLT). W3C Candidate Recommendation 3 Version 2.0, World Wide Web Consortium, November 2005.

[69] M. Kohlhase and H. Franke. MBase: Representing Knowledge and Content for the Integration of Mathematical Software Systems. Journal of Symbolic Computation, 32(4), pp 365-402, 2001.

[70] Maple. Maple Soft, February 2006. See http://www.maplesoft.com/.

[71] Massimo Marchiori. The Mathematical Semantic Web. In *Proceedings of the Second International Conference on Mathematical Knowledge Management (MKM 2003), 16-18 February 2003, Bertinoro, Italy*, pages 216–224. Springer.

[72] David L. Martin, Massimo Paolucci, Sheila A. McIlraith, Mark H. Burstein, Drew V. McDermott, Deborah L. McGuinness, Bijan Parsia, Terry R. Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia P. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In Jorge Cardoso and Amit P. Sheth, editors, *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2004.

[73] MathBroker — A Framework for Brokering Distributed Mathematical Services. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2004. See http://www.risc.uni-linz.ac.at/projects/basic/mathbroker.

[74] MathBroker II: Brokering Distributed Mathematical Services. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2006. See http://www.risc.uni-linz.ac.at/research/parallel/projects/mathbroker2/.

[75] Mathematica. Wolfram Research, February 2006. See http://www.wolfram.com/.

[76] Mathematica Phrasebook. Research Institute for Applications of Computer Algebra (RIACA), Technical University of Eindhoven, Eindhoven, Holand, May 2006. See http://www.riaca.win.tue.nl/products/mathematica/.

[77] Web Services Package . Wolfram Research, February 2006. See http://www.wolfram.com/solutions/mathlink/webservices/.

[78] MathLink. Wolfram Research, February 2006. See http://www.wolfram.com/solutions/mathlink/mathlink.html.

[79] Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation, February 2001. See http://www.w3.org/TR/MathML2.

[80] MathWeb, February 2006. See http://www.mathweb.org/.

[81] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, February 2004. See http://www.w3.org/TR/owl-features/.

[82] Sheila A. McIlraith and David L. Martin. Bringing Semantics to Web Services. *IEEE Intelligent Systems*, 18(1):90–93, 2003.

[83] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[84] METEOR-S: Semantic Web Services and Processes. LSDIS Lab University of Georgia, February 2006. See http://swp.semanticweb.org/.

[85] Mizar Project, September 2004. See http://mizar.org/project/.

[86] Mathematical Knowledge Management Network, November 2003. See http://monet.nag.co.uk/mkm/index.html.

[87] MONET — Mathematics on the Web. The MONET Consortium, April 2004. http://monet.nag.co.uk.

[88] Enrico Motta, John Domingue, Liliana Cabral, and Mauro Gaspari. IRS-II: A Framework and Infrastructure for Semantic Web Services. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 306–318. Springer, October 2003.

[89] Mathematics On the Web: Get it by Logic and Interfaces (MoWGLI), February 2006. See http://www.mowgli.cs.unibo.it/.

[90] Mathematical Services Description Language (MSDL). Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2004. http://poseidon.risc.uni-linz.ac.at:8080/mathbroker/results/xsd.html.

[91] Mathematical Services Description Language Library API. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2004. http://poseidon.risc.uni-linz.ac.at:8080/results/xsd/monet-based/api/index.html.

[92] Mathematical Services Description Language (MSDL) Schema. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2004. http://poseidon.risc.uni-linz.ac.at:8080/results/xsd/monet-based/mathbroker.xsd.

[93] William Naylor and Julian Padget. Semantic Matching for Mathematical Services. In *Proceedings of the Forth International conference on Mathematical Knowledge Management*, Bremen, Germany, 15 – 17 July, 2005. Springer.

[94] Netlib Repository, February 2006. See http://www.netlib.org/.

[95] NetSolve, February 2006. See http://icl.cs.utk.edu/netsolve/.

[96] Organization for the Advancement of Structured Information Standards (OASIS), February 2006. See http://www.oasis-open.org/home/index.php.

[97] OMDoc: A Standard for Open Mathematical Documents. MathWeb.org, September 2005. See http://www.mathweb.org/omdoc/.

[98] RIACA OpenMath Library. Research Institute for Applications of Computer Algebra (RIACA), Technical University of Eindhoven, Eindhoven, Holand, May 2006. See http://www.riaca.win.tue.nl/products/openmath/lib/.

[99] OWL Web Ontology Language Reference. W3C, February 2004. See http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[100] OWL-S: Semantic Markup for Web Services. W3C, November 2004. See http://www.w3.org/Submission/OWL-S/.

[101] OWL Web Ontology Language Guide. W3C, February 2004. See http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[102] A Conceptual Comparison between WSMO and OWL-S. WSMO, January 2005. See http://www.wsmo.org/2004/d4/d4.1/v0.1/20050106/.

[103] Terence Parr. ANTLR (ANother Tool for Language Recognition) Reference Manual, September 2005.

[104] PostgreSQL. The PostgreSQL Global Developer Group, May 2006. See http://www.postgresql.org.

[105] Resource Description Framework (RDF). W3C, September 2004. See http://www.w3.org/RDF/.

[106] RDF Gateway a Platform for Semantic Web. Intellidimension, September 2004. See http://www.intellidimension.com/.

[107] RDFQL. Intellidimension, September 2004. See http://www.intellidimension.com/.

[108] RDF Query Survey. W3C, April 2004. See http://www.w3.org/2001/11/13-RDF-Query-Rules/.

[109] RDF/XML Syntax Specification. W3C, February 2004. See http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[110] RDQL - A Query Language for RDF. W3C, January 2004. See http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[111] Arthur Ryman. Understanding Web Services. IBM, July 2003. See http://www-128.ibm.com/developerworks/websphere/library/techarticles.

[112] Cristina Schmidt and Manish Parashar. A Peer-to-Peer Approach to Web Service Discovery. *World Wide Web*, 7(2):211–229, 2004.

[113] David A. Schmidt. *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Boston, 1 edition edition, 1986.

[114] Wolfgang Schreiner. The RISC ProofNavigator. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, March 2006. See http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator/.

[115] Semantic Web. World Wide Web Consortium, March 2004. http://www.w3.org/2001/sw.

[116] Kaarthik Sivashanmugam, Kunal Verma, and Amit P. Sheth. Discovery of Web Services in a Federated Registry Environment. In *IEEE International Conference on Web Services (ICWS'04)*, pages 270–278, San Diego, California, USA, June 2004. IEEE Computer Society.

[117] Elena S. Smirnova, Clare M. So, and Stephen M. Watt. An Architecture for Distributed Mathematical Web Services. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Mathematical Knowledge Management, Third International Conference (MKM 2004), Bialowieza, Poland, September 19-21, 2004, Proceedings*, volume 3119 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2004.

[118] Simple Object Access Protocol (SOAP), June 2003. See http://www.w3.org/TR/soap/.

[119] Structured Query Language. FIPS, June 1993. See http://www.itl.nist.gov/fipspubs/fip127-2.htm.

[120] RDF Squish Query Language. ILRT, February 2001. See http://ilrt.org/discovery/2001/02/squish/.

[121] Buswell Stephen, Caprotti Olga, and Dewar Mike. MONET Architecture Overview. The MONET Consortium, 2002. See http://monet.nag.co.uk/cocoon/monet/publicdocs/monet-overview.pdf.

[122] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Series in Computer Science. MIT Press, Cambridge MA, 1977.

[123] SOAP with Attachments. W3C Note, December 2000. See http://www.w3.org/TR/SOAP-attachments.

[124] SymbolicNet. Institute for Computational Mathematics (ICM), Kent State University, February 2006. See http://www.symbolicnet.org/.

[125] Apache Tomcat. The Apache Jakarta Project, April 2004. http://jakarta.apache.org/tomcat/.

[126] UDDI Version 2.04 API Specification. OASIS, July 2002. http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf.

[127] Kunal Verma, Kaarthik Sivashanmugam, Amit Sheth, Abhijit Patil, Swapna Oundhakar, and John Miller. METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Journal of Information Technology and Management*, 6(1):17–39, 2005.

[128] World Web Consortium (W3C), February 2006. See http://www.w3.org/.

[129] Paul S. Wang. Design and Protocol for Internet Accessible Mathematical Computation. In *International Symposium on Symbolic and Algebraic Computation ISSAC 1999*, Vancouver, Canada, July 28-31, 1999. ACM Press.

[130] webMathematica. Wolfram Research, February 2006. See http://www.wolfram.com/products/webmathematica.

[131] Web Services Architecture. W3C, February 2004. See http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[132] Web Service Choreography Interface (WSCI), August 2002. See http://www.w3.org/TR/wsci/.

[133] Web Services Conversation Language (WSCL), March 2002. See http://www.w3.org/TR/2002/NOTE-wscl10-20020314/.

[134] Web Services Description Language (WSDL), January 2006. See http://www.w3.org/TR/wsdl20/.

[135] Web Service Modeling Language (WSML). W3C, June 2005. See http://www.w3.org/Submission/WSML/.

[136] Web Service Modeling Ontology (WSMO). W3C, June 2005. See http://www.w3.org/Submission/WSMO/.

[137] XML Schema, February 2006. See http://www.w3.org/XML/Schema.

[138] XQEngine , March 2005. See http://xqengine.sourceforge.net/.

Appendix A

Registry Examples

A.1 An XSLT Stylesheet for GAMS Transformation

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="xml"/>
 <xsl:output indent="yes"/>
 <xsl:template match="/">
   <xsl:element name="PredefinedConcepts">
    <xsl:element name="JAXRClassificationScheme">
     <xsl:attribute name="id">
      <xsl:text>urn:uuid:gams-id</xsl:text>
     </xsl:attribute>
    <xsl:attribute name="name">
     <xsl:text>gams:1997</xsl:text>
    </xsl:attribute>
   </xsl:element>
   <xsl:apply-templates/>
  </xsl:element>
 </xsl:template>
 <xsl:template match="//gamslist" name="nested">
  <xsl:for-each select="gamslist">
   <xsl:element name="ClassificationNode">
    <xsl:attribute name="id">
     <xsl:text>urn:uuid:gams-id/</xsl:text>
     <xsl:value-of select="gamscode"/>
    </xsl:attribute>
    <xsl:attribute name="code">
     <xsl:value-of select="gamscode"/>
    </xsl:attribute>
   <xsl:element name="Name">
         <xsl:element name="LocalizedString">
          <xsl:attribute name="charset">UTF-8</xsl:attribute>
          <xsl:attribute name="value">
           <xsl:value-of select="title"/>
           </xsl:attribute>
```

```
          </xsl:element>
      </xsl:element>
     <xsl:if test="count(child::gamslist) > 0">
           <xsl:call-template name="nested"/>
        </xsl:if>
    </xsl:element>
   </xsl:for-each>
 </xsl:template>
</xsl:stylesheet>
```

A.2 A Sample MSDL Service Description

```
<monet:definitions
 targetnamespace=
  "http://risc.uni-linz.ac.at/mathbroker/RischIndefIntegration"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:mathb="http://risc.uni-linz.ac.at/mathbroker/ns"
 xmlns:monet="http://monet.nag.co.uk/monet/OpenMathDC"
 xmlns:om="http://www.openmath.org/OpenMath"
 xmlns:symbint="http://perseus.risc.uni-linz.ac.at:8080/
                axis/services/SymbolicIntegration"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemalocation="http://monet.nag.co.uk/monet/OpenMathDC/
                 home/olga/cvs/perseus/monet-based-xsd/xsd/monetOM_DC.xsd">
  <!--  $Id: risch.xml,v 1.1 2004/04/23 10:49:52 rbaraka Exp $ -->

  <mathb:machine_hardware address="193.170.37.69"
   name="perseus.risc.uni-linz.ac.at">
    <mathb:CPU name="Intel Celeron"></mathb:CPU>
    <mathb:CPU_speed mhz="733"></mathb:CPU_speed>
    <mathb:RAMsize mb="256"></mathb:RAMsize>
    <mathb:disksize gb="40"></mathb:disksize>
    <mathb:OS href="http://www.suse.de"></mathb:OS>
  </mathb:machine_hardware>

  <monet:problem name="indefinite-integration">
    <monet:header></monet:header>
    <monet:body>
      <monet:input name="f">
        <monet:signature>
          <om:OMOBJ>
            <om:OMA>
              <om:OMS cd="sts" name="mapsto"></om:OMS>
              <om:OMS cd="setname1" name="R"></om:OMS>
              <om:OMS cd="setname1" name="R"></om:OMS>
              <om:OMS cd="setname1" name="R"></om:OMS>
            </om:OMA>
          </om:OMOBJ>
        </monet:signature>
      </monet:input>
      <monet:output name="i">
        <monet:signature>
```

```
42            <om:OMOBJ>
43              <om:OMA>
44                <om:OMS cd="sts" name="mapsto"></om:OMS>
45                <om:OMS cd="setname1" name="R"></om:OMS>
46                <om:OMS cd="setname1" name="R"></om:OMS>
47                <om:OMS cd="setname1" name="R"></om:OMS>
48              </om:OMA>
49            </om:OMOBJ>
50          </monet:signature>
51        </monet:output>
52        <monet:post-condition>
53          <om:OMOBJ>
54            <om:OMA>
55              <om:OMS cd="relation1" name="eq"></om:OMS>
56              <om:OMV name="i"></om:OMV>
57              <om:OMA>
58                <om:OMS cd="calculus1" name="indefint"></om:OMS>
59                <om:OMV name="f"></om:OMV>
60              </om:OMA>
61            </om:OMA>
62          </om:OMOBJ>
63        </monet:post-condition>
64      </monet:body>
65    </monet:problem>
66    <monet:algorithm name="RischAlg">
67      <monet:documentation>This is the metadata for the algorithm
68        Risch. The namespace is the target namespace of this document.
69        </monet:documentation>
70      <monet:bibliography href="http://www.emis.de/cgi-bin/zmen/ZMATH/en/
71                                    quick.html?type=xml&an=0184.06702">
72        <!-- more dublin core -->
73        <monet:documentation> Dublin Core Data </monet:documentation>
74        <dc:creator>Risch,R.H.</dc:creator>
75        <dc:title>The Problem of Integration in Finite Terms</dc:title>
76        <dc:source> Trans. A.M.S. 139 pp.167 - 189</dc:source>
77        <dc:publisher>AMS</dc:publisher>
78        <dc:date>1969</dc:date>
79      </monet:bibliography>
80    </monet:algorithm>
81      <monet:implementation name="RImpl">
82      <mathb:efficiency_factor wrt="S200Spec">
83        <mathb:speed>1.1</mathb:speed>
84        <mathb:throughput>0.7</mathb:throughput>
85      </mathb:efficiency_factor>
86        <monet:software href="http://www.wolfram.com"></monet:software>
87        <monet:software href="http://riaca.win.tue.nl/software/ROML">
88        </monet:software>
89        <monet:hardware href="http://risc.uni-linz.ac.at/mathbroker/
90                          RischIndefIntegration/perseus.risc.uni-linz.ac.at">
91        </monet:hardware>
92        <monet:algorithm href="http://risc.uni-linz.ac.at/
93                          mathbroker/RischIndefIntegration/RischAlg">
94        </monet:algorithm>
95      </monet:implementation>
```

```
96    <monet:service name="RRISC">
97        <monet:documentation>This is an implementation of the algorithm
98          Risch. We use the mathb namespace to state expected performance
99          of the  concrete implementation wrt to its theoretical
100         complexity measure.</monet:documentation>
101     <monet:classification>
102       <monet:problem href="http://risc.uni-linz.ac.at/mathbroker/
103                       RischIndefIntegration/indefinite-integration">
104       </monet:problem>
105     </monet:classification>
106     <monet:implementation href="http://risc.uni-linz.ac.at/mathbroker/
107                           RischIndefIntegration/RImpl">
108     </monet:implementation>
109     <monet:service-interface-description
110       href="http://perseus.risc.uni-linz.ac.at:8080/axis/
111             services/SymbolicIntegration?wsdl">
112     </monet:service-interface-description>
113     <monet:service-binding>
114       <monet:map action="exec" operation="symbint:Integrator:indefInt"
115        problem-reference="indefinite-integration"></monet:map>
116       <monet:message-construction io-ref="f"
117       message-name="symbint:IndefIntRequest" message-part="in0">
118       </monet:message-construction>
119     </monet:service-binding>
120     <monet:service-metadata></monet:service-metadata>
121     <monet:broker-interface>
122       <monet:service-URI></monet:service-URI>
123     </monet:broker-interface>
124   </monet:service>
125 </monet:definitions>
```

A.3 Publish and Query Registry Examples

A.3.1 Publish Example

```
1  import java.io.*;
2  import java.util.*;
3  import java.io.InputStream.*;
4  import java.net.PasswordAuthentication;
5  import java.net.URL;
6  import javax.activation.*;
7  import javax.xml.parsers.*;
8  import javax.xml.registry.*;
9  import javax.xml.registry.infomodel.*;
10 import javax.xml.registry.BulkResponse;
11 import javax.xml.registry.ConnectionFactory;
12 import javax.xml.registry.infomodel.*;
13 import com.sun.xml.registry.ebxml.ConnectionFactoryImpl;
14 import at.ac.uni_linz.risc.mathbroker.registry.infomodel.*;
15 import at.ac.uni_linz.risc.mathbroker.registry.infomodel.impl.*;
16 import javax.xml.bind.*;
17 import com.sun.msv.grammar.*;
18 import org.w3c.dom.*;
```

```
19 import javax.xml.parsers.*;
20 import org.xml.sax.InputSource;
21 import org.xml.sax.SAXException;
22 import monet.openmath.lang.*;
23 import monet.openmath.lang.impl.*;
24 import org.openmath.lang.*;
25 import nl.tue.win.riaca.openmath.io.OMXMLReader;
26 import nl.tue.win.riaca.openmath.lang.OMObject;
27 import org.xml.sax.XMLReader;
28 import at.ac.uni_linz.risc.mathbroker.lang.*;
29 import at.ac.uni_linz.risc.mathbroker.lang.impl.*;
30 import at.ac.uni_linz.risc.mathbroker.lang.ObjectFactory;
31 import at.ac.uni_linz.risc.mathbroker.registry.infomodel.*;
32 import com.sun.xml.registry.ebxml.*;
33 import org.oasis.ebxml.registry.bindings.rim.ExtrinsicObjectType;
34
35 /**
36  * The MathBrokerPublish class consists of a main method, a
37  * makeConnection method, and a publish method.
38  * It takes an xml file with mathbroker element defenitions,
39  * extracts each element to a separate xml file, and
40  * creates a Mathbroker object for it and loads the
41  * corresponding repository item file to the Mathbroker
42  * registry.
43  */
44
45 public class MathBrokerPublish {
46  MathBrokerRegistryService mrs = null;
47  MathBrokerConnection connection=null;
48  MathBrokerFocusedQueryManager fqm=null;
49  MathBrokerLifeCycleManager mlcm=null;
50  Marshaller marshaller =null;
51  String uri=null;
52  String username = "user1";
53  String password = "testuser1";
54
55  public MathBrokerPublish(){
56  }
57
58  public static void main( String[] args ) throws Exception {
59   String queryPublishUrl =
60    "http://perseus.risc.uni-linz.ac.at:8080/ebxmlrr/registry/soap";
61   MathBrokerPublish mathbrokerPublish = new MathBrokerPublish();
62   mathbrokerPublish.makeConnection(queryPublishUrl);
63   mathbrokerPublish.publish(args);
64  }
65
66  /**
67   * Establishes a connection to a registry.
68   * @param queryUrl  the URL of the query registry
69   */
70  public void makeConnection(String queryPublishUrl) {
71   /*
72    * Define connection configuration properties.
```

```
73     */
74
75    // To publish, you need both the query URL and the publish URL
76    Properties props = new Properties();
77    props.setProperty("javax.xml.registry.queryManagerURL",
78                      queryPublishUrl);
79    props.setProperty("javax.xml.registry.lifeCycleManagerURL",
80                      queryPublishUrl);
81    props.setProperty("javax.xml.registry.factoryClass",
82                  "com.sun.xml.registry.ebxml.ConnectionFactoryImpl" );
83    try {
84     // Create the connection, passing it the configuration properties
85     ConnectionFactory factory =
86       MathBrokerConnectionFactoryImpl.newInstance();
87     System.out.println("NewInstance is " + factory);
88     factory.setProperties( props );
89     MathBrokerConnection connection =
90            (MathBrokerConnection)factory.createConnection();
91     System.out.println( "---Created connection to registry---" );
92     mrs = connection.getMathBrokerRegistryService();
93     mlcm = mrs.getMathBrokerLifeCycleManager();
94     fqm = mrs.getMathBrokerFocusedQueryManager();
95     System.out.println("Got Mathbroker registry service and  manager");
96     // Get authorization from the registry
97     PasswordAuthentication passwdAuth =
98        new PasswordAuthentication( username, password.toCharArray() );
99     Set creds = new HashSet();
100    creds.add( passwdAuth );
101    connection.setCredentials( creds );
102    System.out.println( "Established security credentials" );
103   }catch (Exception e) {
104     e.printStackTrace();
105     if (connection != null) {
106      try {
107       connection.close();
108      } catch (JAXRException jaxre) {}
109     }
110   }
111  }
112
113  public void publish(String [] args) {
114  try {
115   String fileName = args[0];
116   System.out.println("Filename reads: "+ fileName);
117   File repositoryItemFile = new File (fileName);
118   javax.activation.DataHandler repositoryItem =
119           new DataHandler(new FileDataSource(repositoryItemFile));
120         mlcm.publishMathBrokerObject(fqm, repositoryItem);
121   }catch (JAXBException je) {
122           je.printStackTrace();
123   }catch (JAXRException jre) {
124           jre.printStackTrace();
125   }catch (IOException ioe) {
126           ioe.printStackTrace();
```

```
127    }
128   }
129  }
130
```

A.3.2 Query Example

```
1   import java.io.*;
2   import java.net.*;
3   import java.util.*;
4   import javax.xml.registry.*;
5   import javax.xml.registry.infomodel.*;
6   import javax.activation.DataHandler;
7   import com.sun.xml.registry.ebxml.*;
8   import com.sun.xml.registry.ebxml.infomodel.*;
9   import at.ac.uni_linz.risc.mathbroker.registry.infomodel.*;
10  import at.ac.uni_linz.risc.mathbroker.registry.infomodel.impl.*;
11  import javax.xml.bind.JAXBContext;
12  import com.sun.msv.grammar.*;
13
14  /**
15   * The MathBrokerQuery class consists of a main
16   * method, a makeConnection method a getManagers method,
17   * a makeSelection method,  and executeQuery method. It
18   * searches a registry for information about
19   * Mathbroker Object(s).
20   */
21
22  public class MathBrokerQuery {
23   public MathBrokerRegistryService rs;
24   public MathBrokerLifeCycleManager mlcm;
25   public MathBrokerFocusedQueryManager fqm;
26   MathBrokerDeclarativeQueryManager dqm;
27   MathBrokerConnection connection=null;
28   RegistryObject response=null;
29
30   public MathBrokerQuery() {
31    }
32
33   public static void main(String[] args) {
34    String queryURL =
35     "http://perseus.risc.uni-linz.ac.at:8080/ebxmlrr/registry/soap";
36     MathBrokerQuery mbQuery = new MathBrokerQuery();
37     mbQuery.makeConnection(queryURL);
38         mbQuery.makeSelection();
39     }
40
41   /**
42    * Establishes a connection to a registry.
43    * @param queryUrl the URL of the query registry
44    */
45   public void makeConnection(String queryUrl) {
46   /*
47    * Define connection configuration properties.
```

```
48    */
49     Properties props = new Properties();
50     props.setProperty("javax.xml.registry.queryManagerURL", queryUrl);
51     props.setProperty("javax.xml.registry.factoryClass",
52     "com.sun.xml.registry.ebxml.ConnectionFactoryImpl");
53
54    try {
55     // Create the connection, passing it the configuration properties
56     ConnectionFactory factory =
57             MathBrokerConnectionFactoryImpl.newInstance();
58     factory.setProperties(props);
59     connection = (MathBrokerConnection)factory.createConnection();
60     System.out.println("---Created connection to registry---");
61     rs = connection.getMathBrokerRegistryService();
62     mlcm = rs.getMathBrokerLifeCycleManager();
63     fqm = rs.getMathBrokerFocusedQueryManager();
64     System.out.println("Got Mathbroker registry service and  managers");
65    } catch (Exception e) {
66       e.printStackTrace();
67       if (connection != null) {
68        try {
69              connection.close();
70            } catch (JAXRException jaxre) {}
71       }
72     }
73   }
74
75   public void makeSelection() {
76    try {
77     String argument = null;
78     BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
79     System.out.println("Enter selection: 1 to query by Id,
80      2 to query by Name, or 3 to query by Classification ");
81     String str =in.readLine();
82     StringTokenizer st =new StringTokenizer(str);
83     int selection=Integer.parseInt(st.nextToken());
84     switch (selection){
85      case 1 : System.out.println("Enter id.");
86               argument = in.readLine();
87               fqm.executeQueryById(argument);
88               break;
89      case 2 : System.out.println("Enter name. You may use % wildcard.");
90               argument = in.readLine();
91               fqm.executeQueryByName(argument);
92               break;
93      case 3 : System.out.println("Enter classification.
94                                   You may use % wildcard.");
95           argument = in.readLine();
96           fqm.executeQueryByClassification(argument);
97               break;
98      default: System.out.println("Selections are 1 to 3 only");
99     }
100   }catch (IOException e){
101           e.printStackTrace();
```

```
102    }catch (JAXRException jaxre){
103            jaxre.printStackTrace();
104    }catch (javax.xml.bind.JAXBException jaxbe){
105            jaxbe.printStackTrace();
106     }
107   }
108  }
```

Appendix B

MSQL Grammar and Use Cases

B.1 The MSQL Grammar

The grammar of MSQL is a typical Extended Backus-Naur Form (EBNF) [1]. It includes the parser grammar followed by the lexer grammar. It uses the following notation:

- `::=` for rule definition. LHS is a non-terminal. RHS is a sequence of terminals and/or non-terminals with choice and repetition defined by:
- `?` for zero or one of the preceding element.
- `*` for zero or more of the preceding element.
- `+` for one or more of the preceding element.
- `()` for grouping.
- `<>` for non-terminal element.
- `CAPITAL LETTERS` for lexical element.
- `small letters` for key word.

B.1.1 MSQL Parser

```
msqlQuery ::= select
              ( every | some )
              <entity>
              ( from <classificationConcept> )?
              ( <whereClause> )?
              ( <orderbyExpression> )?

entity ::= service
           | implementation
           | algorithm
           | problem
```

```
          | machine

classificationConcept ::= ( SLASH ( URN )?
                            ( NUMBER )?
                            ( LETTER | STRING ) )*

whereClause ::= where <msqlExpr>

msqlExpr ::= <notExpr>
             | <quantifiedExpr>
             | <ifExpr>
             | <letExpr>
             | <semanticExpr>
             | <typeMatchExpr>

notExpr ::= ( NOT )?
            <orExpr>

orExpr ::= <andExpr>
           ( OR <andExpr> )*

andExpr ::= <comparisonExpr>
            ( <AND> <comparisonExpr> )*

comparisonExpr ::= <arithmeticExpr>
                   ( ( EQUALS | NE | LT | LTE | GT | GTE )
                   <arithmeticExpr> )*

arithmeticExpr ::= <additiveExpr>
                   ( ( PLUS | MINUS ) <additiveExpr> )*

additiveExpr ::= <pathExpr> ( STAR <pathExpr> )*

pathExpr ::= <regularExpr> | <stepDel> <regularExpr>

regularExpr ::= <stepExpr>
                ( <stepDel> <stepExpr> )*

stepExpr ::= <step> ( <predicate> )*
             | <primaryExpr> ( <predicate> )*

primaryExpr ::= <var>
                | <literal>
                | <msqlFunction>
                | DOT
                | LPAR <msqlExpr> RPAR

var ::= DOLLAR <qName>
```

```
literal ::= <numericLiteral> | <stringLiteral>
            | <booleanLiteral>

numericLiteral ::= NUMBER

stringLiteral ::= STRING

booleanLiteral ::= true | false

stepDel ::= SLASH | SLASHSLASH

step ::= <nameTest> | <attributeStep>

attributeStep ::= AT <nameTest>

nameTest ::= LETTER ( COLON LETTER )?

qName ::= LETTER ( COLON LETTER )?

predicate ::= LB <msqlExpr> RB

orderbyExpression ::= orderby <msqlExpr>
                      ( COMMA <msqlExpr> )*
                      ( <sort> )?

sort ::= ascending | descending

msqlFunction ::= <fname> LPAR ( <exprList> )? RPAR

fname ::= count | contains | empty
          | position | doc | doclocal
          | exists

exprList ::= <msqlExpr> ( COMMA <msqlExpr> )*

quantifiedExpr ::= <quantifier> <var> in <msqlExpr>
                   ( COMMA <var> in <msqlExpr> )*
                   satisfies <msqlExpr>

quantifier ::= some | every

ifExpr ::= if <msqlExpr> then <msqlExpr> else <msqlExpr>

letExpr ::= let <letAssignmentClause>
            ( COMMA <letAssignmentClause> )*
            in <msqlExpr>

letAssignmentClause ::= <var> ASSIGNMENT <msqlExpr>
```

```
semanticExpr ::=  satisfy LPAR <omObjExpr> RPAR

typeMatchExpr ::= typematch LPAR <omObjExpr>
                  COMMA <omObjExpr> RPAR

omObjExpr ::= <omApplication>
              | <omAttribution>
              | <omBinding>
              | <omInt>
              | <omVar>
              | <omString>
              | <omSymbol>
              | <var>

omApplication ::= oma LPAR <omObjExpr>
                  ( COMMA <omObjExpr> )*
                  ( <varReplacement> )?
                  RPAR

omAttribution ::= omattr LPAR <omObjExpr> COMMA
                  ( <omObjExpr> <omObjExpr> )
                  ( COMMA ( <omObjExpr> <omObjExpr> ) )*
                  ( <varReplacement> )?
                  RPAR

omBinding ::= ombind LPAR <omObjExpr> LB <omBoundVariable>
              ( COMMA <omBoundVariable> )*
              RB <omObjExpr>
              ( <varReplacement> )?
              RPAR

omBoundVariable ::= omvar COLON
                    ( <var> | <omVar> )
                    AT LPAR <omObjExpr> COMMA <omObjExpr>
                    ( <varReplacement> )?
                    RPAR

varReplacement ::= LB <omObjExpr> SLASH <var>
                   ( COMMA <omObjExpr> SLASH <var> )*
                   RB

omInt ::= omi COLON NUMBER

omVar ::= omv COLON ( LETTER | <var> )

omString ::= omstr COLON LETTER

omSymbol ::= oms COLON LETTER COLON LETTER
```

B.1.2 MSQL Lexer

```
SLASH ::= '/'

SLASHSLASH ::= '//'

AT ::= '@'

DOLLAR ::= '$'

DOT ::= '.'

LB ::= '['

RB ::= ']'

LPAR ::= '('

RPAR ::= ')'

COMMA ::= ','

COLON ::= ':'

ASSIGNMENT ::= ':='

EQUALS ::= '='

LT ::= '<'

LTE ::= '<='

GT ::= '>'

GTE ::= '>='

NE ::= '!='

PLUS ::= '+'

MINUS ::= '-'

STAR ::= '*'

URN ::= 'urn:uuid:'

DOUBLE_QUOTE_STRING ::= '"' ( ~( '"' ) )* '"'

STRING ::= DOUBLE_QUOTE_STRING
```

```
DIGIT ::= ( '0'..'9' )

LETTER ::= ( 'a'..'z' | 'A'..'Z' | '_' )
    ( 'a'..'z' | 'A'..'Z' | '-' | '_' | '0'..'9' | '.' )*

NUMBER ::= ( DIGIT )+ ( DOT ( DIGIT )+ )?
```

B.2 The MSQL Grammar in ANTLR Syntax

```
header {
    package at.ac.uni_linz.risc.mathbroker.msql.msqlParser;
    import at.ac.uni_linz.risc.mathbroker.msql.treeWalker.*;
}
class MsqlParser extends Parser;
options {
    k = 2;             // two token lookahead
    exportVocab=Msql;  // Call the vocabulary "msql"
    buildAST = true;   // build the Abstract Syntax Tree
}
tokens
{
 SELECT = "select" <AST=SelectNode>;
 FROM = "from" <AST=FromNode>;
 RETURN = "return" <AST=ReturnNode>;
 WHERE = "where" <AST=WhereNode>;
 SERVICE = "service" <AST=ServiceNode>;
 IMPLEMENTATION = "implementation" <AST=ImplementationNode>;
 ALGORITHM = "algorithm" <AST=AlgorithmNode>;
 PROBLEM = "problem" <AST=ProblemNode>;
 MACHINE = "machine" <AST=MachineNode>;
 COUNT = "count" <AST=CountNode>;
 CONTAINS = "contains" <AST=ContainsNode>;
 EMPTY = "empty" <AST=EmptyNode>;
 POSITION = "position" <AST=PositionNode>;
 DOC = "doc" <AST=DocNode>;
 DOCLOCAL = "localdoc" <AST=LocalDocNode>;
 EXISTS = "exists" <AST=ExistsNode>;
 ORDERBY = "orderby"<AST=OrderbyNode>;
 ASCENDING = "ascending" <AST=AscendingNode>;
 DESCENDING = "descending" <AST=DescendingNode>;
 SOME = "some" <AST=SomeNode>;
 EVERY = "every" <AST=EveryNode>;
 IN = "in" <AST=InNode>;
 IF = "if" <AST=IfExpr>;
 THEN = "then" <AST=ThenNode>;
 ELSE = "else" <AST=ElseNode>;
 LET = "let" <AST=LetExpr>;
 SATISFIES = "satisfies" <AST=SatisfiesNode>;
 AND = "and" <AST=AndExpr>;
 OR = "or" <AST=OrExpr>;
 NOT = "not"<AST=NotExpr>;
 RPATHEXPR;
 ARITHMETICEXPR;
 APATHEXPR;
 STEPEXPR;
 REGULAREXPR;
 STEP;
 OMSYMBOL;
 EQUALS<AST=EqualsNode> ;
 NE<AST=NENode> ;
 LT<AST=LTNode> ;
```

```
53  LTE<AST=LTENode> ;
54  GT<AST=GTNode> ;
55  GTE<AST=GTENode> ;
56  PLUS<AST=PlusExpr> ;
57  MINUS<AST=MinusExpr> ;
58  STAR<AST=MultExpr> ;
59  SLASH<AST=SlashNode> ;
60  SLASHSLASH<AST=SlashSlashNode> ;
61  }
62
63  msqlQuery : SELECT^ (EVERY | SOME) msdlEntity (FROM classification)?
64      (whereClause)? (orderbyExpression)?
65      ;
66  msdlEntity : SERVICE | IMPLEMENTATION | ALGORITHM | PROBLEM | MACHINE
67      ;
68  classification : (SLASH<AST=ClassificationSlashNode>
69        (URN<AST=UrnNode>)?
70        (NUMBER<AST=NumberNode>)?
71          (LETTER<AST=LetterNode>
72          | "problem"<AST=ProblemStep>
73          | "service"<AST=ServiceStep>
74          | "algorithm"<AST=AlgorithmStep>
75          | "implementation"<AST=ImplementationStep>
76          | STRING<AST=StringNode> ))*
77          ;
78  whereClause : WHERE^ msqlExpr
79      ;
80  msqlExpr : notExpr | quantifiedExpr | ifExpr | letExpr
81             | semanticExpr | typeMatchExpr
82      ;
83  notExpr : (NOT^)? orExpr
84      ;
85  orExpr : andExpr (OR^ andExpr)*
86      ;
87  andExpr : comparisonExpr (AND^ comparisonExpr)*
88      ;
89  comparisonExpr : arithmeticExpr ( (EQUALS^ | NE^ | LT^ | LTE^
90                                     | GT^ | GTE^) arithmeticExpr)*
91      ;
92  arithmeticExpr : additiveExpr ((PLUS^ | MINUS^) additiveExpr)*
93      ;
94  additiveExpr : pathExpr ( STAR^ pathExpr)*
95      ;
96  pathExpr : regularExpr {#pathExpr =
97           #([RPATHEXPR, "pathExpr", "RelativePathExpr"], #pathExpr); }
98           | stepDel regularExpr
99           { #pathExpr = #([APATHEXPR, "pathExpr", "AbsolutePathExpr"],
100                           #pathExpr); }
101     ;
102 regularExpr : stepExpr (stepDel stepExpr)*
103     { #regularExpr = #([REGULAREXPR, "regularExpr", "RegularExpr"],
104                        #regularExpr); }
105     ;
106 stepExpr : step (predicate)*
```

```
107             { #stepExpr = #([STEP, "stepExpr", "Step"], #stepExpr); }
108           | primaryExpr (predicate)* { #stepExpr = #([STEPEXPR, "stepExpr",
109             "PrimaryExpr"], #stepExpr); }
110     ;
111 primaryExpr : var
112     | literal
113     | msqlFunction
114     | DOT<AST=DotNode>
115     | LPAR^<AST=LPARNode> msqlExpr RPAR!
116     ;
117 var : DOLLAR^<AST=VarExpr> qName
118     ;
119 literal : numericLiteral | stringLiteral | booleanLiteral
120     ;
121 numericLiteral : NUMBER<AST=NumberNode>
122     ;
123 stringLiteral : STRING<AST=StringNode>
124     ;
125 booleanLiteral : "true"<AST=TrueNode>
126     | "false"<AST=FalseNode>
127     ;
128 stepDel : SLASH | SLASHSLASH
129     ;
130 step : nameTest | attributeStep
131     | "problem"<AST=ProblemStep>
132     | "service"<AST=ServiceStep>
133     | "algorithm"<AST=AlgorithmStep>
134     | "implementation"<AST=ImplementationStep>
135     ;
136 attributeStep : AT^<AST=AtNode> nameTest
137     ;
138 nameTest : LETTER<AST=LetterNode>
139  (COLON<AST=COLONLETTERNode>
140  LETTER<AST=LetterNode>)?
141  ;
142 qName: LETTER<AST=LetterNode>
143     (COLON<AST=COLONLETTERNode>
144      LETTER<AST=LetterNode>)?
145     ;
146 predicate : LB<AST=LBracketNode> msqlExpr
147     RB<AST=RBracketNode>
148     ;
149 orderbyExpression : ORDERBY^ msqlExpr
150     (COMMA<AST=CommaNode> msqlExpr)*
151         (sort)?
152         ;
153 sort : ASCENDING | DESCENDING
154     ;
155 msqlFunction : fname LPAR! (exprList)?
156     RPAR<AST=RPARNode>
157      ;
158 fname : COUNT | CONTAINS | EMPTY | POSITION | DOC | DOCLOCAL| EXISTS
159     ;
160 exprList : msqlExpr
```

```
161     (COMMA<AST=CommaNode> msqlExpr)*
162         ;
163 quantifiedExpr : quantifier var IN<AST=InNode> msqlExpr
164      (COMMA<AST=CommaNode> var
165      IN<AST=InNode> msqlExpr)*
166      SATISFIES<AST=SatisfiesNode> msqlExpr
167      ;
168 quantifier : SOMEˆ<AST=SomeQuantifiedExpr>
169     | EVERYˆ<AST=EveryQuantifiedExpr>
170     ;
171 ifExpr : IFˆ msqlExpr THEN msqlExpr ELSE msqlExpr
172     ;
173 letExpr : LETˆ letAssignmentClause
174     (COMMA<AST=CommaNode> letAssignmentClause)*
175         IN<AST=InNode> msqlExpr
176         ;
177 letAssignmentClause : var ASSIGNMENTˆ<AST=AssignmentNode>
178      msqlExpr
179       ;
180 //Semantic extension of MSQL starts here.
181 semanticExpr : "satisfy"ˆ<AST=SatisfyNode>
182                         LPAR! omObjExpr RPAR!
183     ;
184 typeMatchExpr : "typematch"ˆ<AST=TypeMatchNode>
185                             LPAR! omObjExpr COMMA! omObjExpr RPAR!
186     ;
187 omObjExpr : omApplication | omAttribution | omBinding | omInt
188             | omVar | omString | omSymbol | var
189      ;
190 omApplication : "oma"ˆ<AST=OMAnode>
191                 LPAR! omObjExpr (COMMA<AST=CommaNode>
192                  omObjExpr)* (varReplacement)? RPAR<AST=RPARNode>
193      ;
194 omAttribution : "omattr"ˆ<AST=OMATTRnode>
195                         LPAR! omObjExpr COMMA! (omObjExpr omObjExpr)
196                         (COMMA<AST=CommaNode>  (omObjExpr omObjExpr))*
197                         (varReplacement)? RPAR!
198      ;
199 omBinding : "ombind"ˆ<AST=OMBINDnode>
200               LPAR! omObjExpr LB<AST=LBracketNode>
201                omBoundVariable (COMMA! omBoundVariable)*
202                RB<AST=RBracketNode>
203                omObjExpr (varReplacement)? RPAR<AST=RPARNode>
204    ;
205 omBoundVariable : "omvar"ˆ<AST=OMVARnode>
206                COLON!<AST=ColonNode>
207                   (var | omVar) AT<AST=AtNode>
208                   LPAR! omObjExpr COMMA! omObjExpr (varReplacement)?
209                   RPAR<AST=RPARNode>
210    ;
211 varReplacement : LBˆ<AST=OMVARreplacement>
212                   omObjExpr SLASH! var
213                   (COMMA<AST=CommaNode>
214                  omObjExpr SLASH!  var)* RB!
```

```
215     ;
216  omInt : "omi"^<AST=OMInode>
217           COLON<AST=ColonNode>
218           NUMBER<AST=NumberNode>
219     ;
220  omVar : "omv"^<AST=OMVnode>
221          COLON<AST=ColonNode>
222          (LETTER<AST=LetterNode> | var)
223     ;
224  omString : "omstr"^<AST=OMSTRINGnode>
225             COLON<AST=ColonNode>
226             LETTER<AST=LetterNode>
227     ;
228  omSymbol : "oms"^<AST=OMsymbol>
229             COLON<AST=ColonNode>
230             LETTER<AST=LetterNode>
231             COLON<AST=ColonNode>
232             LETTER<AST=LetterNode>
233    ;
234
235  //================== Lexer =====================
236
237  class MsqlLexer extends Lexer;
238
239  options {
240      testLiterals=false;  //automatically test for literals
241      k=4;
242      charVocabulary='\3'..'\377' ; // allow ascii
243      importVocab=Msql;
244     //filter = true;
245  }
246
247  SLASH : '/' ;
248
249  SLASHSLASH : '/''/' ;
250
251  AT : '@';
252
253  DOLLAR : '$' ;
254
255  DOT : '.';
256
257  LB : '[';
258
259  RB : ']';
260
261  LPAR : '(';
262
263  RPAR : ')';
264
265  COMMA : ',';
266
267  COLON : ':';
268
```

```
269 ASSIGNMENT : ":=";
270
271 EQUALS : '=' ;
272
273 LT : '<' ;
274
275 LTE : "<=" ;
276
277 GT : '>' ;
278
279 GTE : ">=" ;
280
281 NE : "!=" ;
282
283 PLUS : '+' ;
284
285 MINUS : '-' ;
286
287 STAR : '*' ;
288
289 URN: "urn:uuid:" ;
290
291 protected
292 DOUBLE_QUOTE_STRING  : '"'! (~('"'))* '"'! ;
293
294 STRING: DOUBLE_QUOTE_STRING ;
295
296 protected
297 DIGIT : ('0'..'9') ;
298
299 LETTER options {testLiterals=true;} :
300    ('a'..'z'|'A'..'Z'|'_')
301    ('a'..'z'|'A'..'Z'|'-'|'_'|'0'..'9'|'.')* ;
302
303 NUMBER : (DIGIT)+ (DOT (DIGIT)+)? ;
304
305 WS : ( ' '
306        // | '\r' '\n'
307           | '\r'
308         | '\n'
309         | '\t'
310         )+
311         {$setType(Token.SKIP);}
312       ;
313 EXIT : ';' { System.out.println();
314       System.exit(0);};
```

B.3 MSQL Syntactic Use Cases

The following use cases demonstrate the various functionalities of MSQL. They demonstrate clauses, expressions and alternative ways to express the same query in MSQL. They were submitted by the client application (see Appendix B.5) in a batch to the MSQL engine and were performed and returned MSDL documents as their results.

1. Find a problem with the name "lcm".

```
select some problem
where /problem[@name = "lcm"]
```

2. Find all services in the default classification node.

```
select every service
```

3. Find a problem in /GAMS/Arithmetic, error analysis/Integer with "integration" as part of its name.

```
select some problem
from /GAMS/"Arithmetic, error analysis"/Integer
where
 /problem[contains(@name , "integration")]
```

4. Find all problems in /GAMS/Arithmetic, error analysis/Integer that have no precondition" and arrange them by name in ascending order.

```
select every problem
from /urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/A/A1
where
  //body[empty(/pre-condition)]
orderby /problem/@name ascending
```

5. Find all problems in /GAMS/Arithmetic, error analysis/Integer that have a postcondition $y = gcd(x_1, x_2)$.

```
select every problem
from /urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/A/A1
where
 //body/post-condition//om:OMA/om:OMS[1]
      [@name = "eq"] and
 //body/post-condition//om:OMA/om:OMA[1]/om:OMS[1]
     [@name = "gcd"] and
 //body/post-condition//om:OMA/om:OMA[1]
     [count(/om:OMV) = 1]
```

6. Find all problems in /GAMS/Linear Algebra/Singular value decomposition that have 2 inputs.

```
select every problem
from /GAMS/"Linear Algebra"/"Singular value decomposition"
where
 //body[count(/input) = 2]
```

7. Find all problems in /GAMS/"Linear Algebra"/"Singular value decomposition" with first input having type integer.

```
select every problem
from /urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/D/D6
where
 //body/input[1]/signature/om:OMOBJ/om:OMS[1]
             [@name = "Z" and @cd="setname1"]
```

8. Find all problems in /GAMS/Arithmetic, error analysis/Integer with an input $x : Z \to Z$

```
select every problem
from /urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/A/A1
where
 //body/input/signature//om:OMA/om:OMS[1]
                   [@name = "mapsto" and @cd="sts"] and
 //body/input/signature//om:OMA[count(/om:OMS) = 3] and
 //body/input/signature//om:OMA/om:OMS[2][
                       @name="Z" and @cd="setname1"] and
 //body/input/signature//om:OMA/om:OMS[3][
                       @name="Z" and @cd="setname1"]
```

9. Find all services in /GAMS/Arithmetic, error analysis/Integer that solve the problem "indefinite-integration"

```
select every service
from /urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/A/A1
 where
   //classification/problem[@href = "http://risc.uni-
    linz.ac.at/mathbroker/RischIndefIntegration
                              #indefinite-integration"]
```

10. Find all services that solves the problem (whose names contain) "indefinite-integration".

```
select every service
where
doc(//classification/problem/@href)//problem[
    contains(@name, "indefinite-integration")]
```

11. (*Alternative to the previous query*). Find all services that solves the problem (whose names contain) "indefinite-integration".

```
select every service
where
let $d := doc(//classification/problem/@href) in
$d/problem[contains(@name, "integration")]
```

12. Find all implementations in /Mathbroker/Entities/implementation that run on the machine "koyote".

```
select every implementation
where
  not (empty(//hardware)) and
 //hardware[@name, "koyote"]
```

13. Find all services in /MathBroker/Entities/service that run on the machine named "perseus".

```
select every service
where
  let $d := doc(//implementation/@href) in
  not //classification[empty(/problem)] and
  $d//hardware[contains(@name, "perseus")]
```

14. Find all problems in /GAMS/Arithmetic, error analysis/Integer that have the precondition $\exists S(Singeneric\_alg\_cats.semigroup) \Rightarrow$ $(algebraic\_cats.semigroup\_setS = SemiGroup)$.

```
select every problem
from /urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/A/A1
where
 //body/pre-condition/om:OMOBJ/om:OMBIND/om:OMS[
                                  @name = "exists"] and
 //body/pre-condition/om:OMOBJ/om:OMBIND/om:OMA[
    /om:OMS/@name = "implies"]/om:OMA[1][
     /om:OMS[1]/@name = "in"][/om:OMS[2]
                      /@name = "semigroup"] and
 //body/pre-condition/om:OMOBJ/om:OMBIND/om:OMA[
   om:OMA[2]/om:OMS[1]/@name = "eq"][/om:OMA[3]
    /om:OMS/@name = "semigroup_set"]/om:OMV[1][
                               @name = "SemiGroup"]
```

15. Find all problems in /GAMS/Arithmetic, error analysis/Integer with at least one input and arrange them in descending order.

```
select every problem
from /urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/A/A1
```

```
where
 //body[count(/input) >= 1]
orderby /problem/@name descending
```

16. Find all problems in /GAMS/Arithmetic, error analysis/Integer in which both "sts" content dictionary and "mapsto" symbol occur in the same signature.

```
select every problem
from /urn:uuid:56e73807-5d2f-43c8-925a-ec6341b29dcc/A/A1
where
   some $s in
     $p//signature satisfies
     $s/om:OMOBJ/om:OMA/om:OMS[@cd = "sts"] and
     $s/om:OMOBJ/om:OMA/om:OMS[@name = "mapsto"]
```

B.4 MSQL Semantic Use Cases

The following use cases demonstrate the semantic extension of MSQL. They were submitted by the client application (see Appendix B.5) in a batch to the MSQL engine. The engine performed the queries and returned MSDL documents as their results.

1. Find some service with problem p such that the following formula is satisfied.

$$\begin{aligned} type(input_p) = \mathbb{R} \to \mathbb{R} \ \wedge \\ type(output_p) = \mathbb{R} \to \mathbb{R} \ \wedge \\ \forall\, a \in \mathbb{R} \to \mathbb{R},\, b \in \mathbb{R} \to \mathbb{R}\ (post_p(a,b) \Rightarrow diff(b) = a) \end{aligned}$$

```
select some service
from /GAMS/"Symbolic Computation"
where let $p:= doc(//problem/@href) in
          $a:= $p//input/@name,
          $b:= $p//output/@name,
         $ta:= $p//input/signature/OMOBJ,
         $tb:= $p//output/signature/OMOBJ,
       $post:= $p//post-condition/OMOBJ in
 (typematch(oma(oms:sts:mapsto(oms:setname1:R,
                       oms:setname1:R)), $ta)) and
 (typematch($tb, oma(oms:sts:mapsto(oms:setname1:R,
                           oms:setname1:R)))) and
 (satisfy(ombind(oms:quant1:forall
  [omvar:$a@(oms:sts:type, $ta),
   omvar:$b@(oms:sts:type, $tb)]
  oma(oms:logic1:implies, $post,
      oma(oms:relation1:eq,
          oma(oms:calculus1:diff, omv:$b), omv:$a)))))
```

2. (A variation of the previous Use Case) Find some problem p such that the above formula is satisfied.

```
select some problem
from /GAMS/"Symbolic Computation"
where let $a:= //input/@name,
          $b:= //output/@name,
         $ta:= //input/signature/OMOBJ,
         $tb:= //output/signature/OMOBJ,
       $post:= //post-condition/OMOBJ in
 (typematch(oma(oms:sts:mapsto(oms:setname1:R,
                       oms:setname1:R)), $ta)) and
 (typematch($tb, oma(oms:sts:mapsto(oms:setname1:R,
                           oms:setname1:R)))) and
 (satisfy(ombind(oms:quant1:forall
  [omvar:$a@(oms:sts:type, $ta),
   omvar:$b@(oms:sts:type, $tb)]
  oma(oms:logic1:implies, $post,
```

```
            oma(oms:relation1:eq,
                oma(oms:calculus1:diff, omv:$b), omv:$a)))))
```

3. Find some service with problem p_1 such that its input, output, and postcondition and the input, the output, and the postcondition of the problem p_2 given below satisfy the following formula.

$$\begin{aligned} type(input_{p2}) = type(input_{p1}) \ \wedge \\ type(output_{p1}) = type(output_{p2}) \ \wedge \\ \forall\, a \in type(input_{p2}),\, b \in type(output_{p2})\ (post_{p1}(a,b) \Rightarrow (post_{p2}(c,d)) \end{aligned}$$

```
 <monet:problem
name="indefinite-integrationB">
  <monet:header></monet:header>
  <monet:body>
  <monet:input name="x"/>
   <monet:signature>
    <om:OMOBJ>
     <om:OMA>
      <om:OMS cd="sts" name="mapsto"/>
      <om:OMS cd="setname1" name="R"/>
      <om:OMS cd="setname1" name="R"/>
     </om:OMA>
    </om:OMOBJ>
   </monet:signature>
  </monet:input>
  <monet:output name="y"/>
     ... (Same as input.)
  </monet:output>
  <monet:post-condition>
   <om:OMOBJ>
    <om:OMA>
     <om:OMS cd="relation1" name="eq"/>
     <om:OMA>
      <om:OMS cd="calculus1" name="diff"/>
      <om:OMV name="y"/>
     </om:OMA>
     <om:OMV name="x"/>
    </om:OMA>
   </om:OMOBJ>
  </monet:post-condition>
 </monet:body>
</monet:problem>

select some service
from /GAMS/"Symbolic Computation"
where
 let $p1:= doc(//problem/@href) in
      $a:= $p1//input/@name,
      $b:= $p1//output/@name,
```

```
    $ta:= $p1//input/signature/OMOBJ,
    $tb:= $p1//output/signature/OMOBJ,
 $post1:= $p1//post-condition/OMOBJ,
let $p2 := localdoc(indefinite-integrationB.xml) in
     $c := $p2//input/@name,
     $d := $p2//output/@name,
    $tc := $p2//input/signature/om:OMOBJ,
    $td := $p2//output/signature/om:OMOBJ,
 $post2 := $p2//post-condition/om:OMOBJ,
in
(typematch($tc, $ta)) and
(typematch($tb, $td)) and
(satisfy(ombind(oms:quant1:forall
 [omvar:$a@(oms:sts:type, $tc),
  omvar:$b@(oms:sts:type, $td)]
oma(oms:logic1:implies, $post1, $post2)[$a/$c, $b/$d])))
```

4. (A variation of the previous Use Case) Find some problem p_1 such that its input, output, and postcondition and the input, the output, and the postcondition of the given problem p_2 satisfy the given formula (see previous Use Case).

```
select some problem
from /GAMS/"Symbolic Computation"
where
 let  $a:= //input/@name,
      $b:= //output/@name,
     $ta:= //input/signature/OMOBJ,
     $tb:= //output/signature/OMOBJ,
  $post1:= //post-condition/OMOBJ,
 let $p2 := localdoc(indefinite-integrationB.xml) in
      $c := $p2//input/@name,
      $d := $p2//output/@name,
     $tc := $p2//input/signature/om:OMOBJ,
     $td := $p2//output/signature/om:OMOBJ,
  $post2 := $p2//post-condition/om:OMOBJ,
 in
 (typematch($tc, $ta)) and
 (typematch($tb, $td)) and
 (satisfy(ombind(oms:quant1:forall
  [omvar:$a@(oms:sts:type, $tc),
   omvar:$b@(oms:sts:type, $td)]
 oma(oms:logic1:implies, $post1, $post2)[$a/$c, $b/$d])))
```

B.5 A Client Application Using MSQL API

```
1  import java.io.*;
2  import java.util.*;
3  import at.ac.uni_linz.risc.mathbroker.msql.treeWalker.ChildAST;
4  import at.ac.uni_linz.risc.mathbroker.registry.infomodel.*;
5  import javax.xml.bind.JAXBException;
6  import javax.xml.parsers.DocumentBuilder;
7  import javax.xml.parsers.DocumentBuilderFactory;
8  import javax.xml.registry.JAXRException;
9  import org.w3c.dom.Document;
10 import org.w3c.dom.Element;
11 import org.w3c.dom.NodeList;
12 import org.w3c.dom.Text;
13
14 /**
15  * This class demonstrates the usage of the
16  * Mathematical Services Query Language (MSQL) API.
17  * It uses the API in the follwoing manner:
18  * It reads a set of MSQL queries from an xml file,
19  * Executes each query against the MSDL documents fetched from the registry.
20  * Displays those documents satisfying each query.
21  *
22  * @author Rebhi Baraka
23  */
24 public class MsqlApiDemo {
25  public static void main(String[] args){
26   MsqlApiDemo msqlApiDemo = new MsqlApiDemo();
27   String registryURL =
28    "http://koyote.risc.uni-linz.ac.at:8080/omar/registry/soap";
29   Properties connectionProps = new Properties();
30   MsqlQueryEvaluator msqlQueryEvaluatorImpl = new MsqlQueryEvaluatorImpl();
31   MathBrokerConnection registryConnection;
32   String userInput;
33   System.out.println("MSQL API demo");
34   try {
35    FileInputStream fileInputStream =
36     new FileInputStream("documents/case.xml");
37    InputStream queryInputStream = fileInputStream;
38    MsqlQuery msqlBase = new MsqlQueryImpl();
39    //make connection to the MathBroker registry.
40     System.out.println("Connecting to MathBroker registry... \n");
41     connectionProps.put("javax.xml.registry.queryManagerURL", registryURL);
42     registryConnection = msqlBase.makeConnection(connectionProps);
43
44    //Used to build a dom model for the xml file containing the query.
45    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
46    factory.setNamespaceAware(true);
47    DocumentBuilder builder = factory.newDocumentBuilder();
48
49    //xml document containing MSQL use cases.
50    Document queryDoc = (Document)builder.parse(queryInputStream);
51    NodeList cases = queryDoc.getElementsByTagName("query");
52    for (int i = 0; i < cases.getLength(); i++){
```

```
53      Element element = (Element)cases.item(i);
54      Text text = (Text)element.getFirstChild();
55      String queryString = text.getData();
56      System.out.println();
57      System.out.println("Parsing the Query... \n");
58      ChildAST queryTree = msqlBase.parseQuery(queryString);
59      msqlBase.printQuery(queryTree);
60      System.out.println("Retreiving MSDL documents from the registry
61                          and performing the query on them... \n");
62      Collection resultsCollection =
63        msqlBase.performQuery(queryTree, registryConnection);
64      System.out.println("Printing MSDL documents satisfying the query...");
65      msqlApiDemo.resultIterator(resultsCollection);
66      System.out.println("****** End of results for this query ******");
67     }
68    }catch (Exception e) {
69          e.printStackTrace();
70     }
71   }
72   /**
73    * This method simply iterates over the documents satisfying the query
74    * and prints them according to the user preference;
75    * either one by one or all at once.
76    * @param results The collection of satisfying documents to be printed out.
77    */
78   public void resultIterator(Collection results) throws IOException,
79    JAXRException, JAXBException {
80    BufferedReader input =
81      new BufferedReader(new InputStreamReader(System.in));
82    String userInput="";
83    System.out.println("  press ENTER to see next resulting document or type
84           'all' to print out all resulting documents for the current query");
85    Iterator resultIter = results.iterator();
86    try {
87    userInput = input.readLine();
88    if (userInput.equalsIgnoreCase("this"))
89        while (resultIter.hasNext()){
90         Object obj = resultIter.next();
91         if (obj instanceof MathBrokerObject){
92            MathBrokerObject mathBrokerObject = (MathBrokerObject)obj;
93            System.out.println("-------------------------------------");
94            mathBrokerObject.print();
95         }
96       }
97    else if (userInput.equalsIgnoreCase("all"))
98      while (resultIter.hasNext()){
99       MathBrokerObject mathBrokerObject =
100         (MathBrokerObject)resultIter.next();
101      System.out.println("-------------------------------------");
102      mathBrokerObject.print();
103     }
104   else
105     while (resultIter.hasNext()){
106      System.out.println("-------------------------------------");
```

```
107       MathBrokerObject mathBrokerObject =
108          (MathBrokerObject)resultIter.next();
109       mathBrokerObject.print();
110       userInput = input.readLine();
111      }
112    }catch (Exception e) {
113      System.err.println();
114     }
115  }
116 }
```

Index

Curriculum Vitae

Rebhi Soliman Baraka

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Phone: +43 732 2468 9967 Fax: +43 732 2468 9930
rbaraka@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/rbaraka

Personal Information

First name: Rebhi
Middle initial: Soliman
Family name: Baraka
Date of birth: Feb. 20, 1967
Place of birth: Gaza Strip, Palestine
Nationality: Palestinian
Marital status: Married
Children: 3 Girls and 2 Boys

Education

Ph.D. in Computer Science, Johannes Kepler University, 2006 (Expected).

M.Sc. in Computer Science, De La Salle University, Philippines, 1996.

B.S. in Electronics and Communications Engineering, University of the East, Philippines, 1991.

Work Experience

Research Assistant, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2002–2006.

Instructor, The Islamic University of Gaza, Gaza, Palestine, 1997–2001.

Instructor, College of Education, Gaza, Palestine, 1997.

Instructor, College of Science and Technology, Khanyounis, Palestine, 1997.

Publications and Technical Reports

Rebhi Baraka and Wolfgang Schreiner. *Semantic Querying of Mathematical Web Service Descriptions*. Third International Workshop on Web Services and Formal Methods (WS-FM 2006), September 8-9, 2006. Vienna, Austria.

Rebhi Baraka and Wolfgang Schreiner. *Querying Registry-Published Mathematical Web Services.* IEEE 20th International Conference on Advanced Information Networking and Applications (AINA 2006). April 18-20, 2006. Vienna, Austria.

Rebhi Baraka. *Mathematical Services Query Language: Design, Formalization, and Implementation.* Project Report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2005.

Rebhi Baraka, Olga Caprotti, and Wolfgang Schreiner. *A Web Registry for Publishing and Discovering Mathematical Services.* IEEE Conference on e-Technology, e-Commerce, and e-Service (EEE-05), March 29 - April 1, 2005. Hong Kong.

Rebhi Baraka. *A Foundation for a Mathematical Web Services Query Language: A Survey on Relevant Query Languages and Tools.* Technical Report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2004.

Rebhi Baraka, Olga Caprotti, and Wolfgang Schreiner. *Publishing and Discovering Mathematical Service Descriptions: A Web Registry Approach.* Technical Report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2004.

Rebhi Baraka, Olga Caprotti, and Wolfgang Schreiner. *A Registry Service as a Foundation for Brokering Mathematical Services.* Technical Report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, February 2004.

Attended Conferences and Workshops

IEEE AINA 2006 IEEE 20th International Conference on Advanced Information Networking and Applications, Vienna, April 18-20, 2006

IEEE EEE 05 IEEE Conference on e-Technology, e-Commerce, and e-Service, Hong Kong, March 29 - April 1, 2005.

AISC 2004 Seventh International Conference on Artificial Intelligence and Symbolic Computation, Hagenberg, Austria, September 22-24, 2004.

Mathematical Web Services, Project workshop. RISC-Linz, Johannes Kepler University, November 11-12, 2002, Linz, Austria.

LMCS 2002 Logic, Mathematics and Computer Science: Interactions, October 20-22, 2002, Hagenberg, Austria.

ADG 2002 Fourth International Workshop on Automated Deduction in Geometry, Hagenberg, Austria, September 04-06, 2002.

MKM 2001 First International Workshop on Mathematical Knowledge Management, Hagenberg, Austria, September 24-26, 2001.

Building Interactive Media for Technical Education and Training Using Multimedia Software Tools, The Islamic University of Gaza, December 13-17, 1998, Palestine.

First IUG's Mathematics and Computer Conference , The Islamic University of Gaza, February 28, 1998, Palestine

Summer Workshop for Advanced Computing, De La Salle University, May 22-26, 1995, Philippines.

Stimulating Computing Research in the Academe, Industry, and Government, De La Salle University, November 25-27, 1993, Philippines.

Refereeing

MKM2005 Fourth International Conference on Mathematical Knowledge Management, Bremen, Germany, July 15-17, 2005.

IJITWE International Journal of Information Technology and Web Engineering, Inaugural Issue: January 2006.

Software Packages

MathBroker Registry Provider.

Mathematical Services Query Language (MSQL).

Talks

A Framework for Publishing and Discovering Mathematical Web Services. RISC forum, RISC-Linz, Johannes Kepler University, Linz, Austria. June 26, 2006.

Querying Registry-Published Mathematical Web Services. IEEE 20th International Conference on Advanced Information Networking and Applications (AINA 2006). April 18-20, 2006. Vienna, Austria.

A Web Registry for Publishing and Discovering Mathematical Services. IEEE Conference on e-Technology, e-Commerce, and e-Service. Hong Kong Baptist University, Hong Kong, March 29 - April 1, 2005.

Brokering Mathematical Services Through a Web Registry. RISC forum, RISC-Linz, Johannes Kepler University, Linz, Austria, May 14, 2004.

XML Registries. MathBroker Workshop on Mathematical Web Services, RISC-Linz, Johannes Kepler University, Linz, Austria, November 11-12, 2002.