RZ 3582 (# 99592) 02/07/2005
Computer Science 86 pages

Research Report

Thesis Report
Building and Querying a Repository of BPEL Process Specifications

Jussi Vanhatalo

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Almaden • Austin • Beijing • Delhi • Haifa • T.J. Watson • Tokyo • Zurich

HELSINKI UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering

Software Business and Engineering Institute

Building and Querying a Repository of BPEL Process Specifications

by Jussi Vanhatalo

Thesis submitted in partial fulfillment of the requirements

for the degree of Master of Science in Technology

Zurich, September 29, 2004

Supervisor: Professor Casper Lassenius, M.Sc. (Tech.)

Instructor: Jana Koehler, Ph.D. (Dr. rer.nat.)

Abstract of the Master's Thesis

Author:	Jussi Vanhatalo
Title of the Thesis:	Building and Querying a Repository of BPEL Process Specifications
Date:	September 29, 2004
Pages:	86
Department:	Department of Computer Science and Engineering
Professorship:	T-76 Software Engineering
Field of Study (Major):	Software Engineering and Business
Supervisor:	Professor Casper Lassenius, M.Sc. (Tech.)
Instructor:	Jana Koehler, Ph.D. (Dr. rer.nat.)

There are no suitable repositories available satisfying the existing requirements of the domain around the *Business Process Execution Language for Web Services* (BPEL4WS, BPEL for short). This Master's Thesis explores an object-oriented approach for providing data retrieval and querying services to other software. The BPEL repository is built on top of the *Eclipse* platform and it uses the *Eclipse Modeling Framework* (EMF) for handling the data persistence of objects as XML files.

Instead of using the native querying language of a database system, an object-oriented approach is utilized. The data is queried in EMF object representation. The Eclipse Modeling Framework takes care of converting the data from one form to another and transforming the data models to another.

The *Object Constraint Language* is used as an object-oriented querying language for the repository data. The power of an object-oriented approach is that only the object representation of data must be known when the queries are formulated. Programs use this object representation internally and thus, it is straightforward to construct queries for this data representation. It is not necessary to know how the data is stored, for example in a relational database, or in an XML database.

The querying performance of the BPEL repository scales linearly compared to the repository size, because all the queried files are loaded completely in a main memory, when the query is executed. No indexing mechanisms or other query speed-ups familiar from database systems are utilized. However, the querying mechanism scales well enough for the purpose it was created. Future work could explore how only the necessary parts of the files are loaded into objects using for example a lazy XML loading mechanism. Another option is to map the object-oriented queries to a query language provided by a database system.

Keywords: Repository, object-oriented, query, BPEL, OCL

Diplomityön tiivistelmä

Tekijä:	Jussi Vanhatalo
Työn nimi:	Tietokannan ja hakumenetelmän toteutus BPEL prosessikuvauksille
Päivämäärä:	29.9.2004
Sivumäärä:	86
Osasto:	Tietotekniikan osasto
Professuuri:	T-76 Ohjelmistotuotanto ja -liiketoiminta
Pääaine:	Ohjelmistotuotanto ja -liiketoiminta
Työn valvoja:	Professori Casper Lassenius, DI
Työn ohjaaja:	Jana Koehler, TkT (Dr. rer.nat.)

Business Process Execution Language for Web Services (BPEL4WS, lyhyesti BPEL) on suunniteltu bisnesprosessien kuvauskieleksi. Näille tiedostoille sopivaa dokumenttikirjastoa ei ole olemassa. Tämä diplomityö tutkii oliosuuntautunutta lähestymistapaa tarjoten tiedonhakupalveluja muille ohjelmistoille. BPEL-dokumenttikirjasto on suunniteltu *Eclipse* alustalle ja se hyödyntää *Eclipse*n mallinnusmenetelmää (*Eclipse Modeling Framework*, EMF) hoitamaan tiedon muuntamisen olioesitystavasta XML-tiedostoiksi.

Dokumenttikirjasto käyttää oliosuuntautunutta lähestymistapaa tiedonhakuun, sen sijaan että haut suoritettaisiin suoraan tietokannan tukemalla kielellä. Haettava tieto on mallinnettu EMF-olioina. EMF hoitaa tiedon muuntamisen kuvauskielestä toiselle ja tietomallien muuntamisen mallinnuskielestä toiselle.

Oliorajoituskieltä (*Object Constraint Language*, OCL) on käytetty oliomallisen tiedon hakemiseen. Oliosuuntautuneen tavan etuna on se, että ainoastaan tiedon oliomalli täytyy olla tuttu ohjelmistokehittäjälle, kun hän laatii kyselyjä. Ohjelmat käyttävät oliomalleja sisäisesti ja siksi on suoraviivaista laatia kyselyjä käyttäen samaa tietomallia. Siten ei ole tarpeellisesta tietää tiedon tallentamissyntaksia, jota käytetään tiedon tallentamiseen relaatio- tai XML-tietokantoihin.

Hakumenetelmä käyttää prosessointi- ja muistiresursseja lineaarisesti verrattuna tietokirjaston kokoon, koska kaikki haussa tarvittavat tiedostot ladataan kokonaisuudessaan keskusmuistiin, kun haku prosessoidaan. Dokumenttien sisältöjä ei ole indeksoitu, eikä muita tietokantoista tuttuja haun nopeuttamismekanismeja ole käytetty. Kuitenkin hakumenetelmä skaalautuu tarpeeksi hyvin sen käyttötarkoitukseen. Tulevaisuudessa voitaisiin tutkia, kuinka tietokirjastoa voitaisiin parantaa lataamalla vain hauille olennainen tieto dokumenteista käyttämällä esimerkiksi laiskaa XML-latausmenetelmää. Toisena vaihtoehtona olisi suorittaa haut oliohakukielellä suoraan tietokannassa.

Avainsanat:	Dokumenttikirjasto, oliosuuntautunut, hakumenetelmä, BPEL, OCL

Résumé de Thèse

Auteur :	Jussi Vanhatalo
Titre de la Thèse :	Construction et interrogation d'une bibliothèque pour les spécifications des processus BPEL
Date :	29 septembre, 2004
Pages :	86
Département :	Department of Computer Science and Engineering
Professeur :	T-76 Software Engineering
Option :	Software Engineering and Business
Superviseur :	Professeur Casper Lassenius, M.Sc. (Tech)
Encadrant:	Jana Koehler, Ph.D. (Dr. rer.nat.)

Il n'y pas actuellement de bibliothèque adaptée disponible satisfaisant les contraintes existantes dans le domaine des langages d'exécution pour les processus d'affaires pour les services Internet (*Business Process Execution Language for Web Services*, BPEL4WS, BPEL en résumé). Ce travail de diplôme présente une approche orientée objet qui a pour but de fournir la récupération de données et des services de requêtes pour d'autre logiciel. La bibliothèque BPEL est construite au sommet de la plateforme *Eclipse* et utilise le cadre de la modélisation Eclipse (*Eclipse Modeling Framework*, EMF) pour gérer la persistance des données de type objet tels que les fichiers XML.

Au lieu d'utiliser le langage de requêtes d'un système de bases de données, une approche orientée objet est utilisée. Les données sont interrogées dans une représentation EMF objet. Le cadre de la modélisation EMF prend correctement en compte la conversion de données et de modèles d'un format à un autre.

Le langage objet de contrainte (*Object Constraint Language*, OCL) est utilisé en tant que langage orienté objet de requêtes pour les données de la bibliothèque. L'intérêt d'une approche orientée objet est que seule une représentation objet des données ne doit être connu au moment où les requêtes sont formulées. Les programmes utilisent cette représentation objet de manière interne et par conséquent, il est simple de construire des requêtes pour cette représentation de données. Il n'est pas nécessaire de savoir comment les données sont stockées, par exemple dans une base de données relationnelle ou dans une base de données XML.

Les performances des requêtes d'une bibliothèque BPEL augmentent linéairement par rapport à la taille de celle-ci, étant donné que la totalité des fichiers de requêtes sont complètement chargés par la mémoire principale, au moment où la requête est exécutée. Les indexages, ou encore les mécanismes permettant d'accélérer les requêtes ne sont pas utilisés. Cependant, le mécanisme de requêtes se comporte suffisamment bien pour le but dans lequel il a été créé. De futurs travaux pourront explorer comment seules des parties nécessaires de fichiers ont besoin d'être chargées en tant qu'objet en utilisant par exemple un mécanisme de chargement XML. Une autre possibilité est de faire passer les requêtes orientées objet dans un langage de requête fourni par un système de base de données.

Mots-clés :	Bibliothèque, orienté objet, requête, BPEL, OCL

Acknowledgements

First, I would like to thank Ed Merks for the technical help given to me during my project. He helped me to solve the technical challenges related to Eclipse Modeling Framework. In addition, he helped me to overcome my problems with the Eclipse platform from another continent, even though it is not in his area of responsibility.

Jana Koehler gave exceptional academic guidance with her strong experience, as the industrial supervisor. She contributed to the technological decisions as well. Her theoretical approach and advanced skills in writing academic publications was useful support to my work.

My academic supervisors, Professor Casper Lassenius from the Helsinki University of Technology, and Professor Benoit Huet from the Institute Eurecom, gave their valuable instructions through my Master's Thesis.

John Novotnack contributed to the repository project by designing example BPEL processes. While finalizing the report I was fortunate to have his support as a native English speaker. Michael Wahler instructed me with his Object Constraint Language knowledge, among other support.

In addition, I would like to thank the other BPIA project members, Rainer Hauser, Shane Sendall and Jochen Kuester for providing me a fruitful environment for developing ideas.

I want to thank the researchers in the CHAMPS partner projects, especially Biplav Srivastava, Alexander Keller, Aaron Brown and Joseph Hellerstein, for the co-operation on three continents.

I express my deep gratitude to Maija, Jaakko and Rita for always being there for me no matter what has been going-on.

All my friends around the world, especially in Finland and Switzerland, have kept my free-time interesting and thus, my working motivation high. In addition, Johan Roman provided me his help as a native French speaker.

Last but definitely not least, I am grateful to my loving parents, who have supported me on my path all the way to the academic studies.

Table of Contents

Abstract of the Master's Thesis 2

Diplomityön tiivistelmä 3

Résumé de Thèse 4

Acknowledgements 5

Table of Contents 6

Abbreviations 9

1 Introduction 10

1.1 Motivation 10

1.2 Goal 11

1.3 Scope 12

1.4 Structure of the Thesis 12

2 Requirements for a BPEL Repository 13

2.1 Functional Requirements 13

2.1.1 BPEL File Support 13

2.1.2 Support of Related Standard File Types 15

2.1.2.1 WSDL Public Interface Files 15

2.1.2.2 XSD Message Type Files 16

2.1.3 Support of Arbitrary XML Metadata Files 16

2.1.3.1 WS-Policy Metadata Files 16

2.1.3.2 Multiple Sources for Creating Repository XML Schemas 16

2.1.4 Data Persistence 16

2.1.5 Querying Capabilities 17

2.1.6 Graphical User Interface 17

2.2 Non-Functional Requirements 17

2.3 Usage Scenario – Change Management with Planning and Scheduling 18

2.3.1 Project Specific XML Metadata 19

3 State of the Art 20

3.1 Generic Data Storages 20

3.2 XML File Support 21

3.3 Links amongst XML Documents 21

3.4 Querying Capabilities 21

3.5 Loading a BPEL File as an Object 22

4 Repository Design 23

4.1 Implementation as an Eclipse Plug-in 23

4.1.1 Implementation Support Provided by the Eclipse Platform 23

4.2 Architecture 24

4.2.1 Repository API Plug-in 25

4.2.1.1 Repository Logics Component 25

4.2.1.2 Query Engine Adapter Component ... 25
4.2.1.3 Data Handler Component ... 26
4.2.2 User Interface Plug-in ... 27
4.2.2.1 Graphical User Interface Component ... 27
4.2.2.2 Query Wizard and Result View Component ... 28
4.2.3 External Software Using the API Services ... 29
4.3 Data Structure ... 30
4.3.1 Tree-Structured Organization Hierarchy ... 30
4.3.2 Descriptor File for each Organization ... 31
4.3.2.1 Example Contents of the Descriptor File ... 32
4.3.2.2 Automatic Generation of the Descriptor Files ... 32
4.4 Data Persistence ... 33
4.4.1 Eclipse Modeling Framework ... 33
4.4.2 Extensibility to Service Data Objects ... 34
4.5 Query Mechanism ... 36
4.5.1 Query Parameters ... 36
4.5.2 Query Result Table ... 38
4.5.3 Querying Algorithm ... 39
4.5.4 Query Performance Optimization Approaches ... 40
4.5.4.1 Minimizing the Number of Loaded Files ... 40
4.5.4.2 Minimizing Memory Requirements Using Iteration ... 41
4.5.4.3 References to the Organizations and Files in Hashtables ... 41

5 Repository Performance Measurements ... 42

5.1 Profiling Environment ... 42
5.1.1 Repository Contents Used in the Performance Tests ... 43
5.2 Repository Start-Up Performance Test Cases ... 43
5.2.1 Repository Start-Up Time as a Function of the Repository Size ... 44
5.2.2 Repository Start-Up Time with Different Repository Contents ... 46
5.2.3 Repository Start-Up Memory Usage ... 47
5.3 Querying Performance Test Cases ... 49
5.3.1 Scalability of the Querying Mechanism ... 50
5.3.1.1 Results of the Query Processing Time Tests ... 52
5.3.1.2 Results of the Main Memory Usage Tests ... 54
5.3.2 Scalability of the Querying Mechanism with Fixed Scope ... 56
5.3.3 Query Performance Depending on the Result Types ... 59

6 Qualitative Analysis of the Repository Design ... 63

6.1 Comparison to Other Repositories ... 63
6.1.1 Object Representation of the Data ... 63
6.1.2 Support of Object-Oriented Querying ... 64
6.1.3 Performance of the Querying Mechanism ... 65
6.1.4 Links amongst the XML Documents ... 65
6.1.5 Summary of the Repository Comparison ... 66
6.2 Querying Capabilities ... 66
6.3 Extensibility ... 67

7 Synthesis of the Repository Work ... 69

7.1 Object-Oriented Querying Capabilities 69
7.2 General Analysis of the Repository Performance 70
7.3 Suitability to the Intended Purpose 71
7.4 Future Work 72
8 Conclusions 74
References 76
Appendix I Dictionary of the Repository Terms 80
Appendix II Example WSDL File 82
Appendix III UML Diagrams of the Repository API 84

Abbreviations

API	Application Program Interface
BPIA	Business Process Integration and Automation project
BPEL	Business Process Execution Language for Web Services (BPEL4WS) [ACD+03]
CHAMPS	Change Management with Planning and Scheduling project [KB04], [KHW+04]
CPU	Central Processing Unit
CVS	Concurrent Versions System [Ced04]
DB2	IBM DB2 Universal Database [Ibm04]
DOM	Document Object Model [HHW04], [W3c04]
EMF	Eclipse Modeling Framework [BSM+03], [Ecl04b]
GUI	Graphical User Interface
HUT	Helsinki University of Technology
IBM	International Business Machines Corporation
MDA	Model Driven Architecture
MOF	Meta-Object Facility [Omg03c]
OASIS	Organization for the Advancement of Structured Information Standards
OCL	Object Constraint Language [WK03]
OMG	Object Management Group, Inc.
QVT	MOF 2.0 Query / Views / Transformations [Omg02]
SDO	Service Data Objects [IB04]
SQL	Structured Query Language
SWT	Standard Widget Toolkit [CR04]
UML	Unified Modeling Language [Omg03a]
URI	Uniform Resource Identifier [BFI+98], [Ecl04b]
XMI	XML Metadata Interchange [Omg03b]
XML	Extensible Markup Language [BPS+04]
XQuery	XML Query Language [BCF+04]
XPath	XML Path Language [BBC+04b]
XSD	XML Schema definition language [TBM+01]
WSDL	Web Service Definition Language [CCM+01]

1 Introduction

This Master's Thesis is a part of the Business Process Integration and Automation (BPIA) project at the IBM Zurich Research Laboratory. The project explores the ability of high-level business process models to be transformed into executable applications, such as Web services. There have been a number of XML notations for executable business processes designed by individual software companies including IBM (WSFL [Ley01]) and Microsoft (XLANG [Tha01]). There is on-going work towards a standard for a language for executable business processes, which is under consideration by the Organization for the Advancement of Structured Information Standards (OASIS). At the moment of writing, version 1.1 of the Business Process Execution Language for Web Services (BPEL4WS, or BPEL for short) [ACD+03] is anticipated to be accepted in the near future. It has already become a defacto standard for Web service composition [MSS+04].

The purpose of my internship is to build a repository for a collection of BPEL processes. The repository supports the retrieval of the BPEL files and provides a powerful querying mechanism to find a process with searched properties or metadata.

1.1 Motivation

Despite many on-going research and development projects around the BPEL specification, there is a rather limited set of publicly available BPEL files. A file collection would provide example processes while building software or formulating theory related to the BPEL specification. Thus, there is an interest to have a collection of BPEL files.

As BPEL uses an XML notation, it does not provide very convenient access to the object model for object-oriented programming languages such as Java. In addition to the XML notation, it would be efficient to provide an object model of the BPEL schema for software that handles BPEL files as Java objects. This would allow a developer to use the object model directly as a basis of the application, instead of building a separate

mechanism for each application to convert XML files to objects. A repository where BPEL files are stored could be a reasonable choice to offer this functionality.

As a part of the Eclipse open-source project [Ecl04a], the Eclipse Modeling Framework (EMF) [Ecl04b] provides data persistence mechanism for objects, which can be stored as XML files. In the BPIA project, we are interested in evaluating the ability of EMF to support our design goals.

Since we have an interest to collect a large number of files in the BPEL repository, there should be an efficient way to find these files, search information from them, or retrieve a file with some wished data structure or data. There are specialized languages such as XPath [BBC+04b] and XQuery [BCF+04] for querying XML data. A disadvantage of these languages is that the structure of the XML Schema (XSD) [TBM+01] must be known to formulate the queries. Since we are interested in offering an object model for XML files, we are interested in providing a query mechanism on this object model. It is a more intuitive model for humans to use as a basis of queries than any XML based document syntax, such as XSD. This will provide an advantage for application programmers, who can use the services of the repository to build their software using only the object model and ignoring the corresponding XML syntax.

There is a standard language called Object Constraint Language (OCL) [WK03], which is a part of the broadly used Unified Modeling Language (UML). We were interested to explore how OCL suits for our querying purposes.

BPEL files are often related to other XML files, which provide metadata or a public interface for a BPEL file. However, the mechanism to link these files and their locations is not provided by the standard XML Schemas. Thus, it is necessary to provide storage for this information, if a large collection of files is stored in the same repository.

There is an IBM research project, which is looking for a BPEL repository implementation to be used under its software components. The requirements from the Change Management with Planning and Scheduling (CHAMPS) [KB04], [KHW+04] project provide an example use case of how a BPEL repository can be used in the near future.

1.2 Goal

The goal of my Master's Thesis is to build an open-source BPEL repository. It is targeted to support research projects, which collect, harvest, and exchange BPEL files. In addition to BPEL files it supports other XML files, which are related to BPEL files. It also provides a way to link these files together.

The repository provides an Application Programming Interface (API) for other software to be used as a basis of other solutions. In addition, a graphical user interface (GUI) is implemented for demonstration purposes.

The repository API is generic such that it can be used for multiple purposes. It is designed in an extensible way, so if some necessary functionality is missing from it, open source code can be easily extended from other projects using the repository as a basis of their applications.

The API provides an object model of the BPEL schema and other related files, which can be used by other programs. Retrieving and searching files and their contents from the repository are possible by querying the object model.

1.3 Scope

The scope of my Master's Thesis is to build a BPEL specific repository. It is not meant to be a generic XML database, even though it can support multiple XML file types. Rather it is planned from the BPEL specification point of view, and how other file types relate to BPEL files.

The BPEL repository provides an API for querying and retrieving XML files from the repository. The underlying file storage mechanism, such as a file system or a database is not designed by this project, but the storage system is used under the repository API.

The BPEL repository API provides a querying mechanism for the files, but the query engine itself may be designed by other projects, if a suitable query engine is available.

The goal of the project is rather to build a research prototype of a BPEL repository to be a publicly available example than to build a commercial quality product to be sold by IBM.

1.4 Structure of the Thesis

This Introduction chapter provides the motivations behind the BPEL repository work. The next chapter "2 Requirements for a BPEL Repository" goes into the details for the requirements for the BPEL repository by explaining both the functional and non-functional requirements for the solution. In chapter "3 State of the Art", the existing alternatives for the BPEL repository are examined.

The chapter "4 Repository Design" presents the detailed design for the repository. Its performance is profiled and analyzed in chapter "5 Repository Performance Measurements". The next chapter "6 Qualitative Analysis of the Repository Design" discusses the design decisions behind the repository architecture and compares it to other alternatives. The analysis chapters are followed by chapter "7 Synthesis of the Repository Work", which discusses the results from the quantitative and the qualitative analyses. It also takes an outlook on future work. Finally, the last chapter "8 Conclusions" gives an overview of the explored topics. In addition there is a short dictionary of repository terms in Appendix I.

2 Requirements for a BPEL Repository

The requirements for the BPEL repository are divided into functional and non-functional requirements. First, the functional requirements are explained. Then, the non-functional requirements for the repository are introduced. Finally, the CHAMPS partner project is presented in the last section 2.3.

2.1 Functional Requirements

The repository is designed for BPEL files and other files related to them. It is not meant to be a generic multipurpose XML repository, but it is built to support BPEL-centric environments.

2.1.1 BPEL File Support

The Business Process Execution Language for Web Services defines a notation for a set of communicating executable business processes. It can also be used to describe abstract processes. [ACD+03]

As an example of a simple BPEL document, a Hello World process [Nov04] is illustrated on the next page. A BPEL document consists typically of four different parts. In the first part, the attributes of the *process* XML element declare the namespaces from the referred XML files. The second part consisting of the *partnerLinks* element refers to the public interface of the BPEL process. The public interface is stored in a separate file using the Web Services Definition Language (WSDL).

The *variables* element is the third part of a BPEL file. It introduces the message types that are exchanged between this BPEL process and its partner processes. Finally, the fourth part describes the orchestration-logic as a process activity tree. In this example, a *sequence* is the root element of the activity tree. It declares that the child activities are processed sequentially. The activity for parallel computing is a *flow*. Other

possible activities are *receive*, *reply*, *invoke*, *assign*, *throw*, *terminate*, *wait*, *empty*, *switch*, *while*, *pick*, *scope*, and *compensate*. [ACD+03]

The Hello World process takes a name encoded as a string from a partner process as an attribute and concatenates it with the string *Hello*. The concatenated string is returned as an output. The WSDL public interface for this process is presented in Appendix II.

```
<?xml version="1.0"?>
<!-- SyncHelloWorld BPEL Process -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     NAMESPACE DECLARATIONS (Part 1)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<process name="SyncHelloWorld"
   targetNamespace="http://zurich.ibm.com/bpia/bpel"
   suppressJoinFailure="yes"
   xmlns:tns="http://zurich.ibm.com/bpia/bpel"
   xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
   xmlns:bpelx="http://schemas.collaxa.com/bpel/extension">
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     PARTNERLINKS (Part 2)
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <partnerLinks>
    <partnerLink name="client" partnerLinkType="tns:SyncHelloWorld"
       myRole="SyncHelloWorldProvider"/>
  </partnerLinks>
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     VARIABLES (Part 3)
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <variables>
    <variable name="input"
       messageType="tns:SyncHelloWorldRequestMessage"/>
    <variable name="output"
       messageType="tns:SyncHelloWorldResponseMessage"/>
  </variables>
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     ORCHESTRATION LOGIC (Part 4)
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <sequence name="main">
    <!-- **********************************************************-->
    <!-- Receive a name from the user, append "Hello " to the front-->
    <!-- of the string, and return the final result to the user.   -->
    <!-- **********************************************************-->
    <receive name="receiveInput" partnerLink="client"
       portType="tns:SyncHelloWorld" operation="process"
       variable="input" createInstance="yes"/>
      <assign name="createReturnStr">
        <copy>
          <from expression="concat( 'Hello ', bpws:getVariableData(
            'input', 'payload', '/SyncHelloWorldRequest/name'))"/>
          <to variable="output" part="payload"
             query="/tns:SyncHelloWorldResponse/tns:helloString"/>
        </copy>
      </assign>
    <reply name="replyOutput" partnerLink="client"
       portType="tns:SyncHelloWorld" operation="process"
       variable="output"/>
  </sequence>
</process>
```

2.1.2 Support of Related Standard File Types

A BPEL file describes only the operational logic of a business process, but there is additional information stored separately in other files. The repository must support these file types, because they are needed when a BPEL process is executed, or when a BPEL process is searched based on metadata related to the process. Figure 2.1 illustrates the different file types that can be related to a BPEL file.

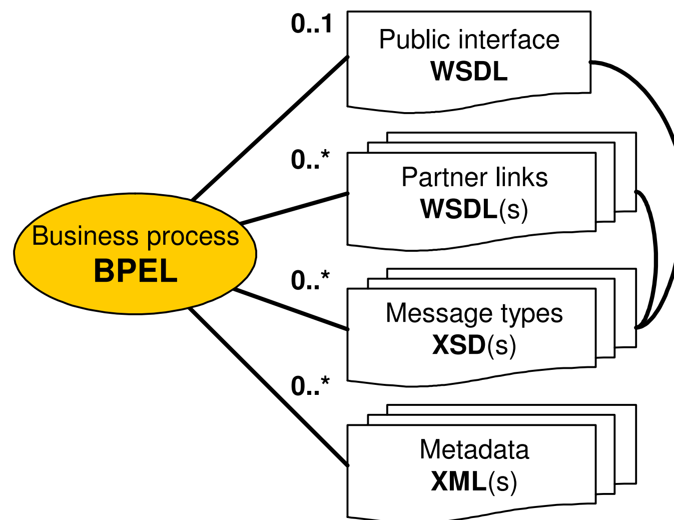

Figure 2.1: A BPEL file is related to its WSDL public interface, the public interfaces of its partner processes, the message types exchanged by the processes and metadata related to the process. Message types are also referred from WSDL files and they can be integrated to the WSDL files.

2.1.2.1 WSDL Public Interface Files

The Web Services Definition Language (WSDL) [CCM+01] defines a standard notation for Web service interfaces. Typically, this file is used to provide a public interface for a BPEL process.

The BPEL and WSDL files do not contain links to the locations where other related files are stored. However, when a BPEL file is executed as a process it needs to be linked to other files, such as its public interface defined as a WSDL file. In order to enable copying a file from one location to another without changing its contents, these links cannot be contained in the files. Nevertheless, applications that make use of these files are dependent on linkages. Therefore, it is helpful for application development that the repository, which provides access to the XML files, takes care of the linkages. The purpose of keeping the file contents independent of location is to make it possible to execute them from different locations.

An example WSDL file is presented in Appendix II. It is the public interface for the Hello World BPEL process. A message type schema is included in the WSDL file.

2.1.2.2 XSD Message Type Files

The XML Schema (XSD) [TBM+01] files are used to describe the types used by the messages exchanged amongst Web services. These schemas are often included in a WSDL file, such as in Appendix II.

2.1.3 Support of Arbitrary XML Metadata Files

It should be possible to attach arbitrary XML files as metadata to a BPEL file. Keeping the metadata in independent files of a BPEL file allows relating arbitrary metadata to BPEL files without extending the standard BPEL XML schema. The metadata is not added unstructured into a BPEL file, which would make reading the metadata more difficult. Independent metadata files allow BPEL files to be deployed in multiple environments, where metadata is separated from the process description.

2.1.3.1 WS-Policy Metadata Files

One example of metadata related to BPEL is WS-Policy [BBC+04a], [BCH+03], a standard for expressing metadata related to Web services. However, in addition or instead of using WS-Policy files, it is possible to link other XML files to a BPEL file in the repository.

2.1.3.2 Multiple Sources for Creating Repository XML Schemas

In addition to using XML metadata files with standard XML schemas, it is possible to add links from a BPEL file to arbitrary metadata files. These files can be defined by their XML schemas, but also other ways of relating metadata to a BPEL file are possible in order to support multi-purpose use of the repository.

Often applications themselves contain information in their data structure, which must be stored in files, while the application is not running. This information has its form of presentation in the object model of the application. It would be convenient to allow storing this data directly from objects to an XML file. Therefore, the repository should enable creating XML metadata files, which get their structure from the class models of the Java objects. Ideally, this structure will be created either from the classes themselves or their UML models.

2.1.4 Data Persistence

The repository offers a data persistence functionality for the software using its services. The repository API provides an object model for BPEL files and other related files. Any client software could directly use these object models as their data structure and they do not have to know how these objects are stored in XML files. The repository hides the details behind the data persistence on the objects and thus, the client software does not to be concerned with them.

There are several kinds of data storage alternatives, such as a file system and different kinds of database systems. These systems have diverse advantages over each other. For example, a file system is an easy solution for a file storage, while databases

often offer more efficient and scalable querying mechanism. Ideally, the repository could use any of these, so a suitable system could be selected depending on the purpose of use. The repository allows any method of data storage and offers a common interface to the client software, which separates them from the data storage details. The repository architecture should be designed in a way that the data storage can be exchanged.

2.1.5 Querying Capabilities

The repository contents can be queried based on their object model representation. The purpose of this requirement is to allow the client software to handle the BPEL and other XML file contents as Java objects, instead of requiring the developers to be aware of the syntax, which is used to store the Java objects as XML. Handling the objects is also a more natural approach to data manipulation. It supports an easy way of navigation by utilizing associations when navigating through elements of the data.

It should be possible to reduce the number of files used in a search by storing the files in a structured way. Reducing the scope of the query also decreases the number of files used in a search by applying the query only to the relevant set of data.

It is possible to retrieve a file, which is selected based on the information that is stored in another file related to the file. For example, a BPEL file can be retrieved by querying metadata linked to the BPEL file.

2.1.6 Graphical User Interface

Even though the repository API is the main contribution of this project, a graphical user interface (GUI) can be implemented in order to demonstrate how the repository functionality works. This helps other software developers to integrate the BPEL repository to their software.

Providing a graphical user interface to the repository enables human-users to manipulate BPEL file collections. This further increases the advantages of using the BPEL repository in other research projects. It also serves as a starting point for applications extending the BPEL repository.

2.2 Non-Functional Requirements

The repository should be designed to be scalable for large numbers of XML files. It is necessary to know the repository scaling behavior.

The repository is a generic solution for storing and retrieving BPEL files. This enables it to be used for multiple purposes. It must be extensible in order to be adopted by client software for required purposes. In order to make it possible to extend the repository for different purposes, its code will be released as open-source, allowing any developer to examine the repository design and modify it based on their requirements.

The repository uses common standards or other widely used solutions, in order to provide support for the standards and to encourage using them. For example, there are several proprietary extensions to the BPEL specification, but the repository is based on

the standard and does not force anyone to use unwanted proprietary extensions of the BPEL specification. Of course, it is an advantage if the BPEL extensions could be handled as well with the repository API, but it is not the main goal of the project. However, since the repository is to be extensible, developers wishing for using the repository with any standard extensions could implement required changes to the design.

2.3 Usage Scenario – Change Management with Planning and Scheduling

There is an IBM research project for Change Management with Planning and Scheduling (CHAMPS) [KB04], [KHW+04]. The BPEL repository API is a part of the research prototype for the change management solution for an IT system illustrated in Figure 2.2. Although developed with this application in mind, the repository is built as a general solution, and not only fulfills the requirements from the partner project.

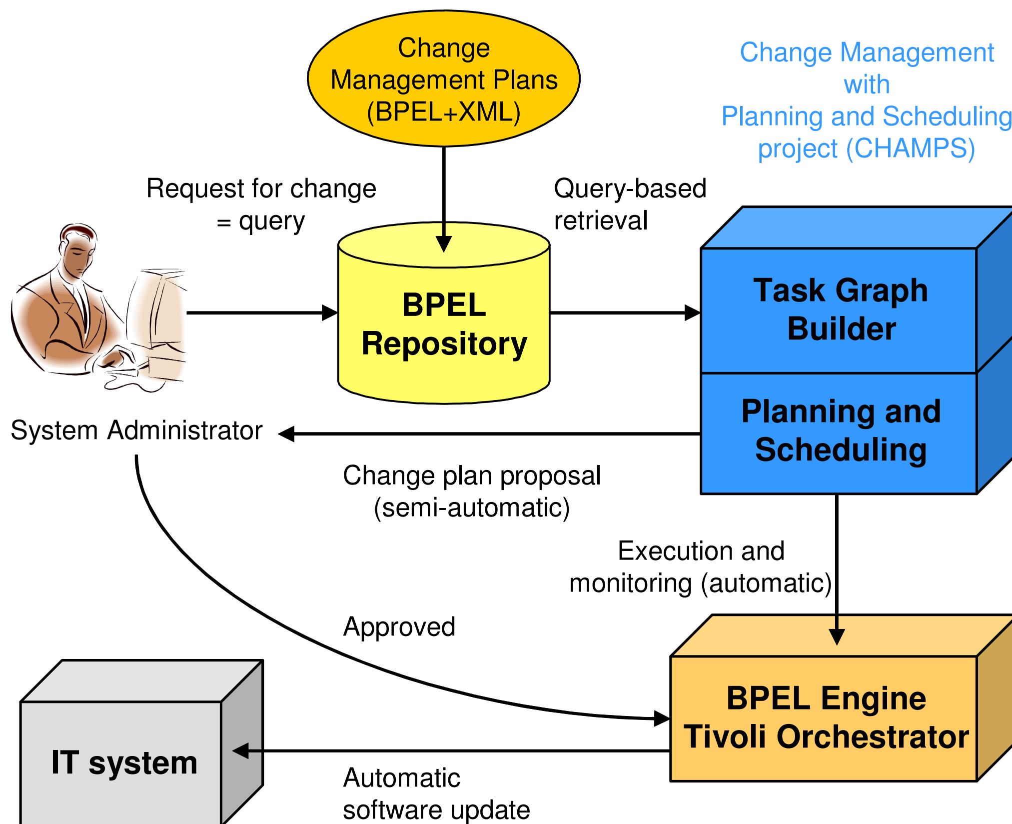

Figure 2.2: The Change Management with Planning and Scheduling (CHAMPS) system. Change management plans and metadata are stored into the BPEL repository. BPEL files are retrieved from the repository based on queries to the file contents.

The BPEL repository is used to store BPEL files, which contain the workflows of the change management plans. In addition, metadata associated with the plans is stored in

the repository. The XML file content querying mechanism is an important contribution of the repository to this usage scenario.

Based on the system administrator's request for a change, the query retrieves related plans from the BPEL repository. The BPEL file metadata is used as a search criteria. The Task Graph Builder component creates a task graph based on the retrieved plans. The Planning and Scheduling component builds a change plan proposal from the task graph.

There are two ways to deploy the change plan. In the semi-automatic way, the system administrator needs to approve the change plan before it is executed. In the other option, the plan is deployed automatically after it is completed. The BPEL Engine possibly used in the Tivoli Orchestrator could execute the BPEL processes in the order of the change plan. These BPEL processes update the IT system.

2.3.1 Project Specific XML Metadata

In this project, a BPEL file is related to project specific XML metadata, which is designed as a UML class diagram represented in Figure 2.3. This is an example case of the need to create a structure for XML files from a Java class model.

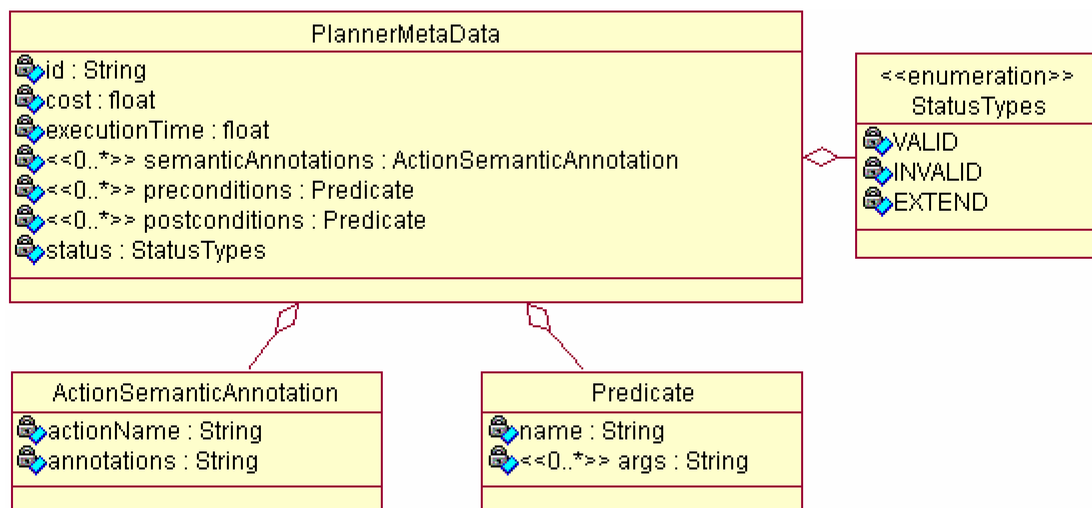

Figure 2.3: The UML class diagram of the planner metadata for the CHAMPS project.

The planner metadata annotations are similar to the essential language elements of the Planning Domain Definition Language (PDDL) [EH04]. The metadata contains a number of attributes including preconditions and postconditions of the BPEL process.

3 State of the Art

There are no publicly available BPEL repositories that could be compared to the approach in this project. There are only generic XML file storages available, where BPEL files can be stored. Applications that use BPEL files as objects do not have a BPEL repository underneath as a separate layer, but instead each application takes care of the data persistence individually.

The solution built in this project is compared to a number of these other more generic storage systems, which do not support all the required BPEL specific functionalities. These storages include:

- File systems (e.g. in Windows XP)
- Version control systems (e.g. CVS [Ced04])
- XML databases (e.g. Natix [FHK+02])
- Relational databases (e.g. DB2 [Ibm04])

The example implementations of the storage types are shown above. These choices are used in the comparisons if not otherwise stated.

3.1 Generic Data Storages

All the above introduced data storages are built for generic use. They support storing various kinds of data. File systems, version control systems and XML databases are built for keeping data organized in files. XML databases provide extra services for handling XML files, unlike normal file systems and version control systems, which do not have specific services for XML files. None of these handle BPEL or WSDL files any differently than other XML files. They are built for storing data rather than providing special services for specific types of files.

Relational databases are built for efficient data storage as their performance in querying and retrieving information outperforms the others. However, they require a

different kind of data organization using a relational representation with tuples. Data is organized in tables, which can have indexes to make the queries more efficient. Thus, they do not have as such native XML file support.

3.2 XML File Support

There are three conceptually different methods for storing XML data: unstructured, shredded and structured [McC04]. The unstructured approach saves an entire XML document as a single unit, an approach used by file systems and version control systems. Relational databases support this approach by allowing the XML document to be stored as a single CLOB data type (Character Large Object). Querying the unstructured XML document is inefficient, as the whole document must be parsed for the query. The structure of the XML document cannot be used to load only the parts of the document relevant to the query.

The conventional approach to enable efficient queries in relational databases of XML data is to shred it into the relational tables. However, with this approach the hierarchical relationships of the XML document are lost. For this reason exporting a file as it was imported is not possible.

Some relational databases have recently started to provide a structured XML storage, also known as native XML support. The structured approach allows efficient queries and preserves the hierarchy of the data. Commercial relational databases, such as IBM DB2 8.1, Oracle 10g, and Sybase ASE 12.5.1, support structured XML storage. However, at the moment only Oracle supports efficient content query with XQuery to XML data.

XML databases, such as Natix from the University of Mannheim, are built to provide structured XML storage. Similarly to Oracle, Natix supports XQuery. Since Oracle 10g is a commercial product, Natix is more suitable as a basis of open-source research prototypes.

3.3 Links amongst XML Documents

None of the compared data storages provide links directly to the locations where related files are stored, but the links can be implemented on top of them. Each application using these data storages implements this functionality itself and most probably differently than in other applications.

An example of this approach is in the Oracle-Collaxa BPEL Designer [Ora04], where linkage information is stored in a separate file and the application is responsible for keeping this data up-to-date. However, these linkages belong to the basic functionalities provided by the BPEL repository.

3.4 Querying Capabilities

In the context of BPEL files, it is important to have an efficient querying mechanism to search file content. File systems and version control systems do not provide

sophisticated querying mechanisms. Instead, file search functionality can be used on the data stored in these systems. This makes it possible to search for a file containing a substring, but any querying mechanisms taking advantage of the structured content of XML files may significantly outperform any file search. In addition, the simple substring search is prone to errors, since the user is most often interested in finding the string in a specific context rather than anywhere in the file. Therefore, a file search can lead to misinterpretations of the data.

XML repositories provide native querying of XML data. Two example query languages XPath [BBC+04b] and XQuery [BCF+04] have the capability of taking advantage of XML document structure. Therefore, XML repositories have efficient querying services for XML documents. Indexing mechanisms over the contents of the documents provide further improvements to querying performance.

Relational databases are specialized in representing data in a way that is efficient for storing, retrieving, and querying data. For example, indexes are often more straightforward to build supporting a broad set of queries than in native XML repositories. Indexes are even better suited for tuples than the unstructured XML document representation. However, storing and indexing structured XML content is more difficult with relational databases, because they are not especially designed for this purpose. Therefore, only Oracle 10g is the only relational database supporting XQuery, which is needed for sophisticated XML queries. For example, it is not possible to combine XML data and produce XML as a result with a relational query language alone [McC04].

3.5 Loading a BPEL File as an Object

The examined data storage systems are not BPEL-specific, so they do not provide a mechanism for converting XML files to objects in some object-oriented programming language, such as Java. As a BPEL object model is not available in these systems, they cannot take care of loading a BPEL file to a BPEL object. For this approach, a repository provides a way to generate an object class model from XML schemas and a default implementation for the generated model.

None of the compared repositories provides an object representation for XML files, so applications which use these repositories implement the functionality themselves if it is necessary. A BPEL repository must provide this functionality, so each application will not have to take care of data serialization individually. In fact, this allows client software to use only the object model. Then the applications do not need to take into consideration the XML representation of the files at all.

4 Repository Design

The repository is designed and implemented as a part of my Master's Thesis. The following sections describe the details of the design decisions and the repository implementation.

4.1 Implementation as an Eclipse Plug-in

In order to make the repository easy to be used as a component in other software, the implementation is designed to be an Eclipse plug-in. Eclipse is an open-source platform for developing software [Ecl04a]. It provides a powerful way to integrate software components. Each component is a plug-in to the platform. Other software can be integrated into existing software components by extending these plug-ins with a new plug-in. Each plug-in must declare other plug-ins it requires, and the extension points it provides for other plug-ins.

The power of the Eclipse architecture is that it can be extended with arbitrary new plug-ins. Eclipse plug-ins are loaded in memory only when they are needed, so the starting time and memory requirements can be minimized. It is still possible to provide a large range of functionality in the same application, and different companies or organizations can implement this functionality independently of each other.

The Eclipse platform provides a powerful support for extensibility, which is one of the main requirements for the repository API. There is a large number of on-going development projects based on the Eclipse platform, which makes it more likely that the repository API will be used as a basis of some other software in the future.

4.1.1 Implementation Support Provided by the Eclipse Platform

In addition to the component integration support that the Eclipse platform supports, it has other powerful features, which increase the interest to use it as the underlying software platform.

The Eclipse project provides the Standard Widget Toolkit (SWT) and JFace libraries for building graphical user interfaces. The libraries provide widgets that are efficient to execute and easy to implement. The repository user interface is implemented on top of the basis that the Eclipse workbench provides. The usability of the application increases, since the look and feel follows the guidelines for any Eclipse application. This further improves the extensibility and usage of the repository as a basis of other software. [GB03]

The Eclipse Modeling Framework (EMF) provides data persistence services to other software. It helps to manage documents as object resources. The resources can be stored in a file system. This topic is further discussed in section 4.4.

4.2 Architecture

The repository implementation is divided into several components, which provide services for each other and hide the details of the solution. This component based approach increases the manageability of the software, since each component concentrates on its specific well-defined functionality. This approach follows the generally accepted design patterns of information hiding and encapsulation [GHJ+95]. The component structure is presented in Figure 4.1.

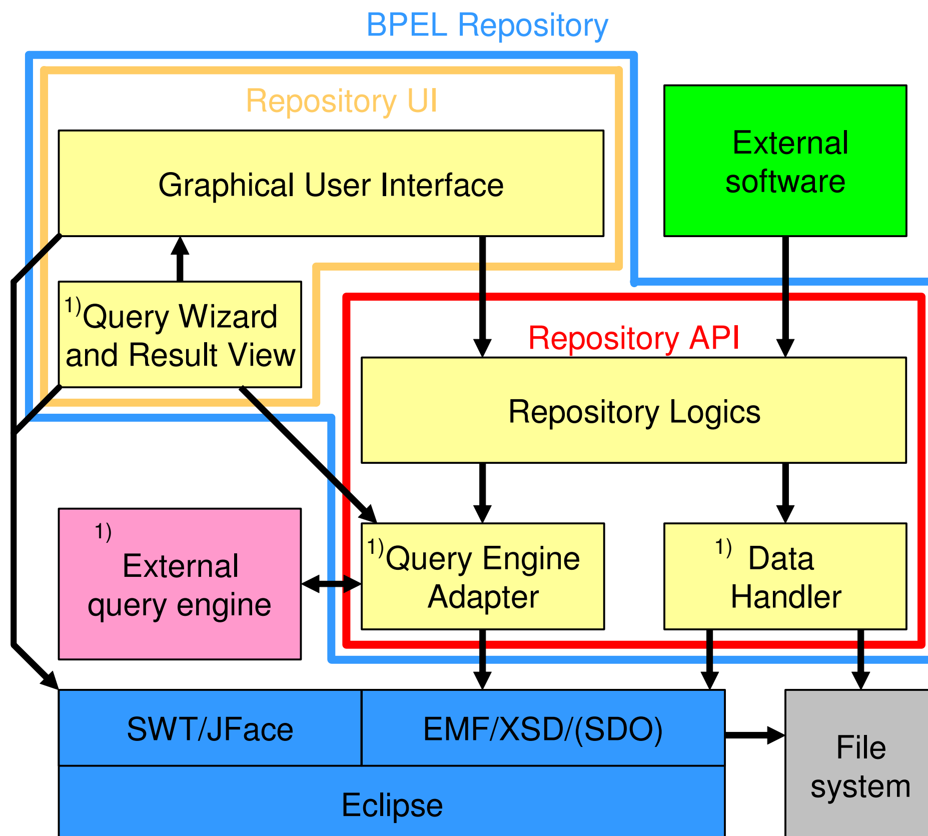

Figure 4.1: The BPEL repository is divided into two Eclipse plug-ins: the repository API and the repository user interface. The plug-ins are further divided into components. 1) Replaceable components.

The repository implementation is divided into two Eclipse plug-ins. This separation is based on the common recommendation to separate the application logic from the user interface. Thus, the repository API plug-in provides the repository business logic functionality, which can be used by other applications or the user interface plug-in.

4.2.1 Repository API Plug-in

The repository API is further divided into three sub-components: repository logics, data handler, and query engine adapter components. Each of them has own responsibilities and provides a specific functionality.

4.2.1.1 Repository Logics Component

The repository logics component is responsible for providing the services as an API for other applications, which are using the repository as a basis of their implementation. The main services that it provides are:

- saving and loading data into the repository
- providing a query mechanism to the repository

However, it uses the services provided by the data handler component for loading and saving data. Similarly, the query engine component is used to execute a query in a loaded object. Nevertheless, in both cases only one file is handled at a time by the used component and the repository logic component iterates over the whole search query scope or the list of persisted files. It selects the repository files that are related to the query.

The class structure of the repository logics component is presented in Figure 8.6 of Appendix III.

4.2.1.2 Query Engine Adapter Component

The query engine adapter component provides an implementation for the query engine interface in the repository logics. With this adapter component, an external query engine can be plugged into the repository. The repository logics component uses a generic interface, which can be implemented by multiple query engines. The query interface is illustrated in Figure 8.3 of Appendix III.

As an example, two different object-oriented query engines are already plugged into the repository by implementing a simple class, which adapts the public methods to implement the query engine interface. The selection of the query engines is illustrated in Figure 4.2.

The primary query engine for the public version of the repository is the OCL Tool from the University of Kent. It is an open-source query engine that implements standard OCL 2.0 and some OCL extensions its authors have decided to implement in it.

For IBM internal use the repository is using an object-oriented query engine that is built at IBM. It is not publicly available and therefore not further discussed in this public Master’s Thesis.

In addition, a simple query engine has been implemented especially for the repository in order to illustrate how easily a custom-made query engine can be plugged into it. The simple query engine searches for a substring in the name attribute of the EMF objects.

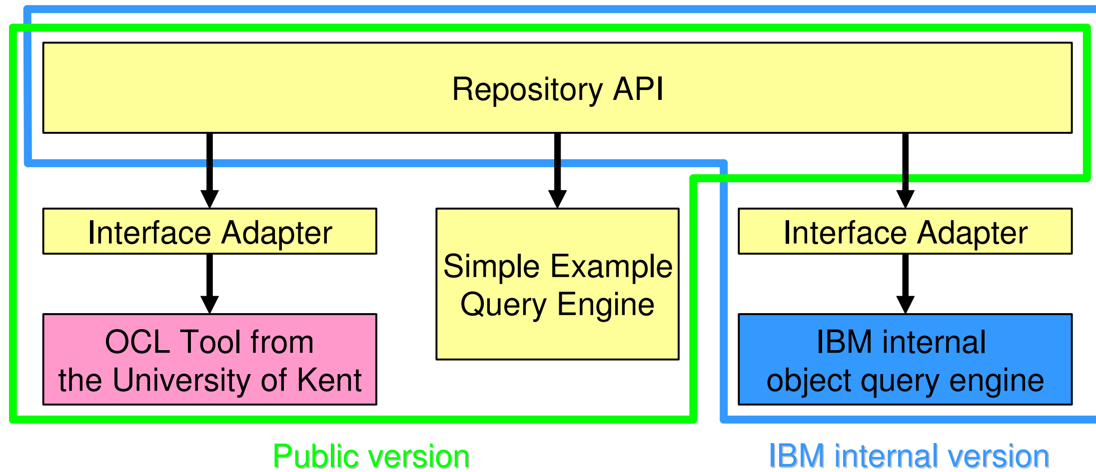

Figure 4.2: Query engine options that are available for the BPEL repository. An external query engine can be adapted to the repository query interface, or a query engine may implement the interface directly.

The query engine interface follows the guidelines of the adapter [GHJ+95], low coupling and indirection design patterns [Lar02]. It allows hiding unnecessary complexity of query engines from the repository logics component as it is interested only the couple of querying services, which are provided by implementing the interface. This design enables a flexible selection of a suitable query engine for the needed purpose. Different query engines can be plugged into the repository, even at runtime by changing the used query engine to another. A reference to a selected query engine is passed to the repository API before querying. At query time, the repository passes the query parameters and the object to be queried through the query interface to the query engine, which it has a reference to.

4.2.1.3 Data Handler Component

The data handler component is responsible for providing access to the underlying persistent data storage system. It hides the selection of the data storage system from the rest of the repository, so data storage system properties do not affect the upper layers of the repository. This approach follows the low coupling and high cohesion design patterns [Lar02].

The data handler takes care of providing access to the object model of the stored files. Thus, it returns a Java object of the retrieved file to the clients using its services. The implemented data handler interfaces are presented in Figure 8.4 of Appendix III.

4.2.2 User Interface Plug-in

The repository user interface plug-in consists of two components. The graphical user interface component is the basis of the plug-in. The other component adds query engine specific functionality to the general user interface.

4.2.2.1 Graphical User Interface Component

The graphical user interface component provides the general user interface illustrated in Figure 4.3. It shows the BPEL Repository perspective in the Eclipse workbench. It contains the Organization Explorer view showing the organization tree in the repository and the Organization Contents view displaying the files inside the selected organization.

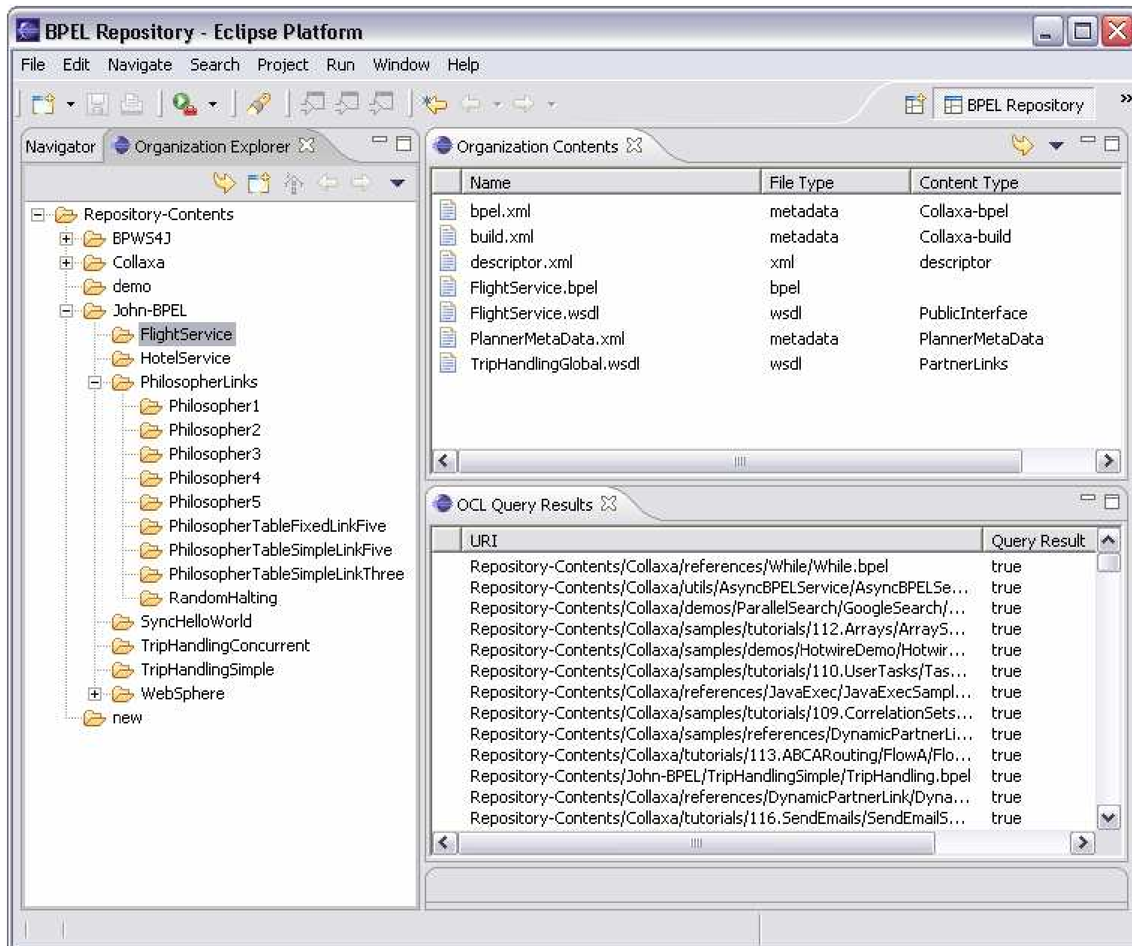

Figure 4.3: The BPEL repository user interface is integrated in the Eclipse workbench.

The user interface is integrated in the Eclipse workbench. It uses the support provided by the Eclipse user interface libraries Standard Widget Toolkit and JFace. Thus, it is seamlessly a part of the Eclipse workbench. This approach has the advantage in usability, since it follows the user interface guidelines for Eclipse having consistent look-and-feel with other Eclipse plug-ins. In addition, it eases extensibility of the

repository as any other Eclipse plug-in can be seamlessly integrated in the same workbench.

The implementation of the BPEL user interface follows the model-view separation principle [Lar02]. The Eclipse API provides the model part, while the view part is separated in the user interface plug-in.

4.2.2.2 Query Wizard and Result View Component

The BPEL user interface is extended with a component implementing a query wizard and a query result view. These widgets are query engine dependent as they collect the necessary parameters for the repository queries. The query wizard determines which query engine is used to execute the query. When a query is about to be executed, it passes a reference of the query engine to the repository logics component.

The query wizard and OCL Query Result view in Figure 4.4 are implemented in the Query Wizard and Result View component.

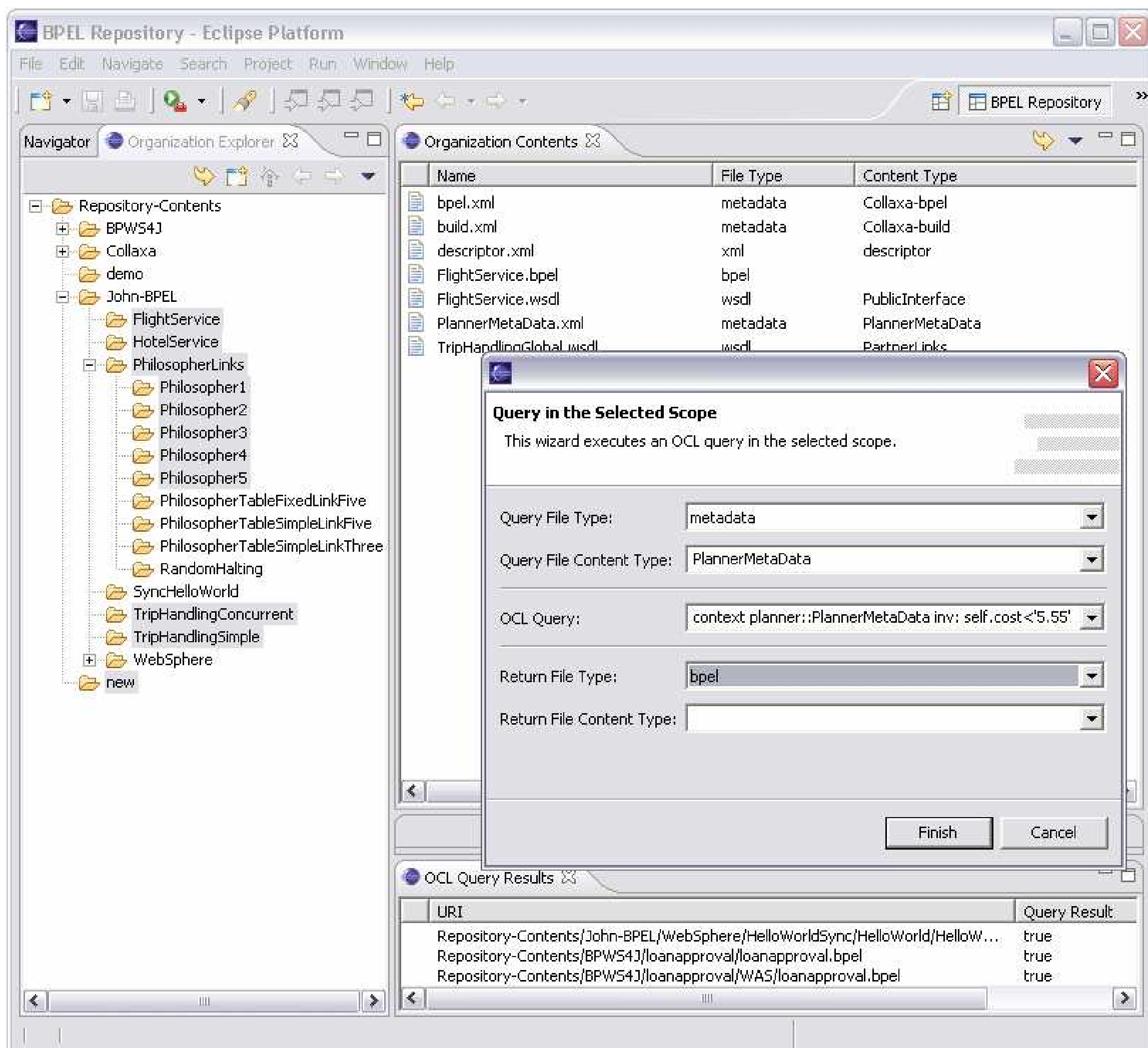

Figure 4.4: The screen shot shows the OCL query wizard and OCL Query Results view that extend the graphical user interface component of the repository.

Adding new wizards and views to the Eclipse workbench is straightforward with the support that the Eclipse platform provides. Providing a wizard for each query engine allows the wizard to collect query engine specific parameters. It also makes it possible to choose an implementation of the user interface, which is most suitable for each query engine. The query engine extends the provided extension points in the Eclipse workbench and the graphical user interface component.

4.2.3 External Software Using the API Services

The BPEL repository is designed primarily to be a service provider for other software. Therefore, the repository API has been designed to implement the services for any external software. The dependencies of the components are presented in Figure 4.5 below.

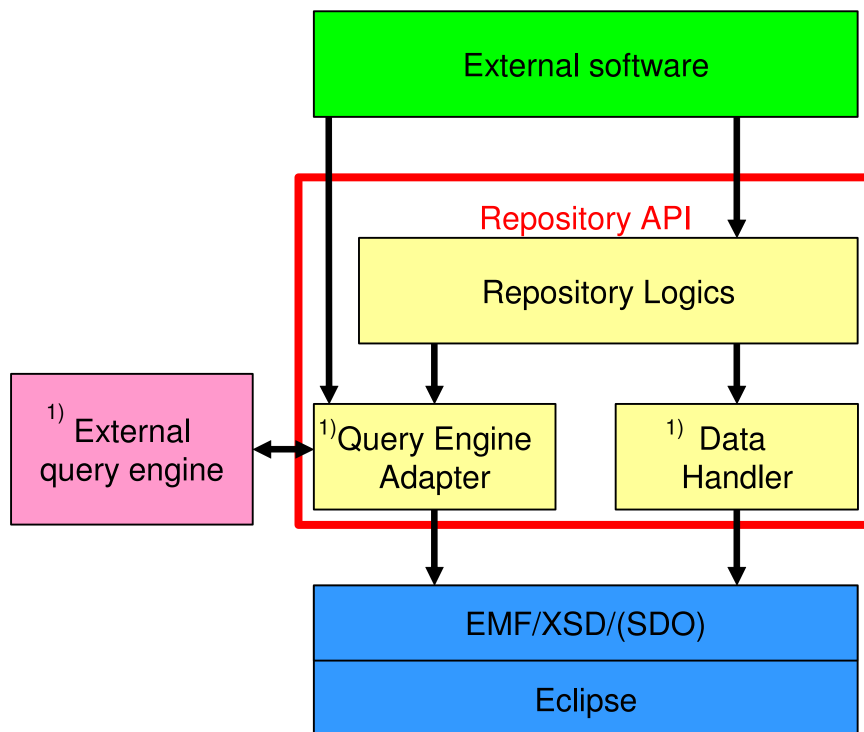

Figure 4.5: External software using the repository API plug-in. [1)] Replaceable components.

The repository logics component provides the generic API functionalities for external software, for example, querying over specified scope and file retrieval from the repository. In addition, external software must specify which query engine it is using. The external software selects the respective query engine adapter and passes a reference to the repository logics component before executing a query. The query engine can be changed between the queries by passing a reference to another query engine.

4.3 Data Structure

Typical contents of the repository are BPEL files and other related XML files. Thus, a straightforward way has been selected to group these files together. Each BPEL file has its own group, where it and the other files related to it are gathered together. This group is called an organization. It helps to find related files and organize files in a hierarchy.

4.3.1 Tree-Structured Organization Hierarchy

Each organization may have suborganizations and all organizations together form a tree-structured hierarchy. Each organization can contain files. This design principle is analogous to a file system in which folders, subfolders, and files are located. An example of the organization tree is illustrated in Figure 4.6.

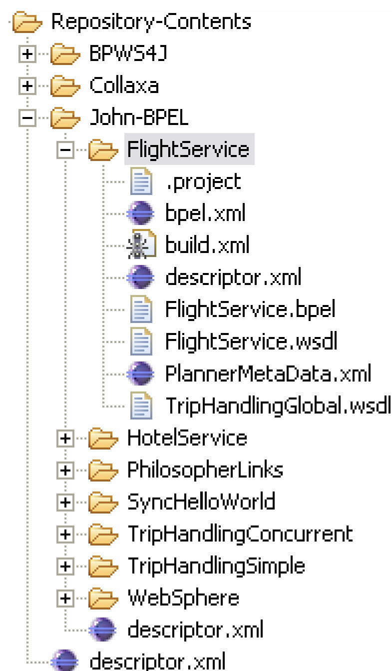

Figure 4.6: An example of the organization tree.

However, a direct mapping to a file system is not used as a basis of the repository data structure. Different operating systems have different types of file systems and the direct mapping would make the data structure dependent on absolute Uniform Resource Identifiers (URI) [BFI+98]. Instead, inside the repository relative URIs to the repository contents root location are used. This root can be a folder in the local file system or in another file system in the same Local Area Network (LAN).

- An example URI referring to a local file system folder:
  `file:/C:/Repository-Contents`
- An example URI referring to a folder in the same intranet:
  `file://Computer-Name/Share-Name/Repository-Contents`

In general, URI can be used to identify the Eclipse workbench workspace folder in a file system or a directory on the Internet. Alternatively, URIs can refer to identifiers in a

database system. This generic way of expressing file locations makes it possible that many different kinds of file storages can be used as a data storage mechanism in the repository. The BPEL repository can be extended to use these different data storages by implementing a data handler component supporting them.

4.3.2 Descriptor File for each Organization

Each organization has a descriptor XML file (descriptor.xml), which specifies how the various XML files are related to a BPEL file in the organization folder.

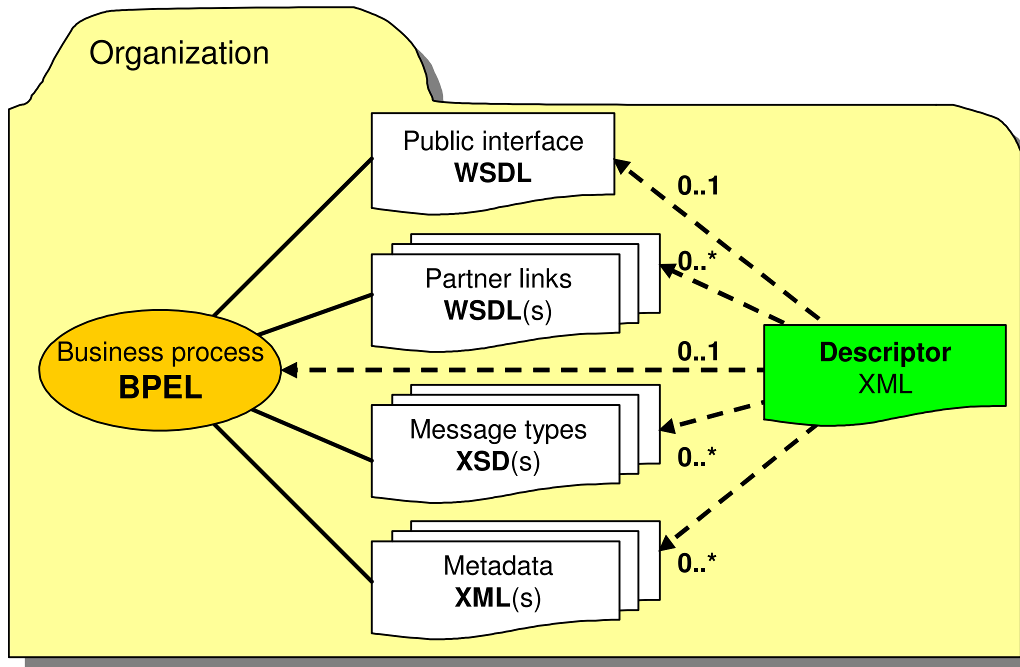

Figure 4.7: Descriptor file links other files in the same organization to the BPEL file and specifies their roles.

For the organization, a descriptor file specifies:

- **Folder Name** – identifier of the organization in its parent organization
- **Folder URI** – identifier relative to the repository root forming a tree structured hierarchy

For each file in the organization the descriptor file entry specifies:

- **File Name** – identifier of the file in its organization
- **File Type** – file type for the file. There are five different options for the file type: BPEL, WSDL, XSD, metadata, and descriptor. It is used together with the file content type identifier to determine the EMF model used in the serialization processes of the file. The pair maps an EMF model to an XML Schema.
- **File Content Type** – identifier among the files in the same organization having the same file type, but different roles and often different XML Schemas.
- **URI** – identifier relative to the repository root forming a tree structured hierarchy.

4.3.2.1 Example Contents of the Descriptor File

The XML document below is an example descriptor file. *Repository-Contents* URI is the identifier for the root organization of this repository. The *Repository-Contents/John/FlightService* organization is a grandchild node of the root organization. It includes one BPEL file and a WSDL public interface for the process. In addition, there is another WSDL public interface file of the partner processes and three different kinds of XML metadata files. Their different roles are identified with content type descriptions. The information of the descriptor file itself is also included.

```
<?xml version="1.0" encoding="ASCII"?>
<emf:EDescriptor xmlns:emf="http://com/ibm/bpia/repository/emf"
   folderName="FlightService"
   folderURI="Repository-Contents/John/FlightService">
  <process name="FlightService.bpel"
     uri="Repository-Contents/John/FlightService/FlightService.bpel"
     fileType="bpel"/>
  <descriptor name="descriptor.xml"
     uri="Repository-Contents/John/FlightService/descriptor.xml"
     fileType="xml" contentType="descriptor"/>
  <publicInterface name="FlightService.wsdl"
     uri="Repository-Contents/John/FlightService/FlightService.wsdl"
     fileType="wsdl" contentType="PublicInterface"/>
  <partnerLinks name="TripHandlingGlobal.wsdl"
     uri="Repository-Contents/John/FlightService/TripHandlingGlobal.wsdl"
     fileType="wsdl" contentType="PartnerLinks"/>
  <metadata name="bpel.xml"
     uri="Repository-Contents/John/FlightService/bpel.xml"
     fileType="metadata" contentType="Collaxa-bpel"/>
  <metadata name="build.xml"
     uri="Repository-Contents/John/FlightService/build.xml"
     fileType="metadata" contentType="Collaxa-build"/>
  <metadata name="PlannerMetaData.xml"
     uri="Repository-Contents/John/FlightService/PlannerMetaData.xml"
     fileType="metadata" contentType="PlannerMetaData"/>
</emf:EDescriptor>
```

4.3.2.2 Automatic Generation of the Descriptor Files

When a folder is opened for the first time as a BPEL repository root organization, the descriptor files are created for the whole organization tree below it. This functionality helps when importing an existing file collection into the repository.

The automatic descriptor file generator cannot fill in all the information automatically. Content type identifiers must be added manually to the descriptor files. For example, one WSDL file per organization is assumed to be a public interface for the process. When an organization contains multiple WSDL files, the repository chooses the first one in alphabetical order to be the public interface. This assumption can, of course, be incorrect and would thus need to be corrected by hand.

The primary method to add new content to the repository, is importing files individually with their descriptor data. This can be done from the user interface using the import file wizard or programmatically using the repository API methods. The automatic generation of the descriptor data is only an additional support feature.

4.4 Data Persistence

The BPEL repository provides access to objects loaded from XML files. The reason behind this is to enable other software using the BPEL repository API to access directly the Java objects representing the files, instead of needing to convert them to objects themselves. Thus, external client software does not need to be aware of the XML representation of a BPEL process and other related schemas for XML files. Instead it can fully concentrate on handling the Java objects. The BPEL repository maintains the data persistence of these objects.

4.4.1 Eclipse Modeling Framework

The data persistence of the repository is handled using the Eclipse Modeling Framework (EMF) [BSM+03], [Ecl04b]. It is an Eclipse plug-in especially designed to save and load objects in XML Metadata Interchange (XMI) [BSM+03], [Omg03b] and XML formats. The model language used to represent models in EMF is called Ecore. Ecore is itself an EMF model, and thus it is its own metamodel. Ecore is comparable to the Meta-Object Facility (MOF) model [Omg03c], as both the models define a subset of UML for describing class modeling concepts [Ecl04b]. MOF is defined by the Object Management Group (OMG).

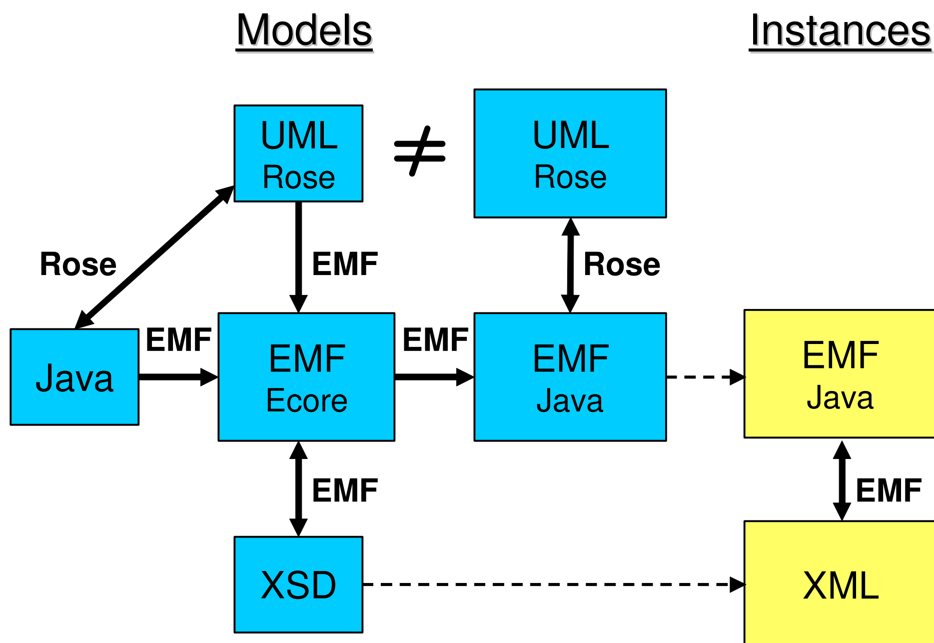

Figure 4.8: Model transformations and data format conversions related to the Eclipse Modeling Framework. The EMF Ecore model can be generated from UML, Java or XML Schema. EMF Java code can be generated from an Ecore model. EMF functionality for data persistence is added when the EMF Java code is created. Therefore, the reverse-engineered UML model from EMF Java code has more classes and methods than the original UML model. Using the EMF model Java objects, their instances can be serialized into XML and deserialized from XML.

It is possible to automatically generate Java EMF model classes directly from Ecore model. These classes have built-in support for EMF data persistence mechanism. Ecore models can be generated automatically from several different sources. These sources include Java code annotations for the classes, IBM Rational Rose UML models and XML Schemas. Especially in the case of generating Ecore models from XML Schemas, this approach is very convenient. The EMF objects serialized in XML files are compliant with the XML Schema from which their Java EMF model was created. For example, the Java classes generated from the XML Schema for BPEL are serialized as valid BPEL files. This is very useful, when an application uses standard XSDs. It produces XML files compliant to those XSDs. This functionality is very useful in the BPEL repository, which needs to support the standard XML Schemas.

The Eclipse Modeling Framework is used to create object models for the BPEL, WSDL and XML schemas. In addition, the BPEL metadata EMF object model used in the CHAMPS project is created from an UML class diagram. Similarly, the descriptor file contents are designed as UML class diagram, illustrated on the left-hand side of Figure 4.9. The right-hand side of the figure is the automatically generated EMF model.

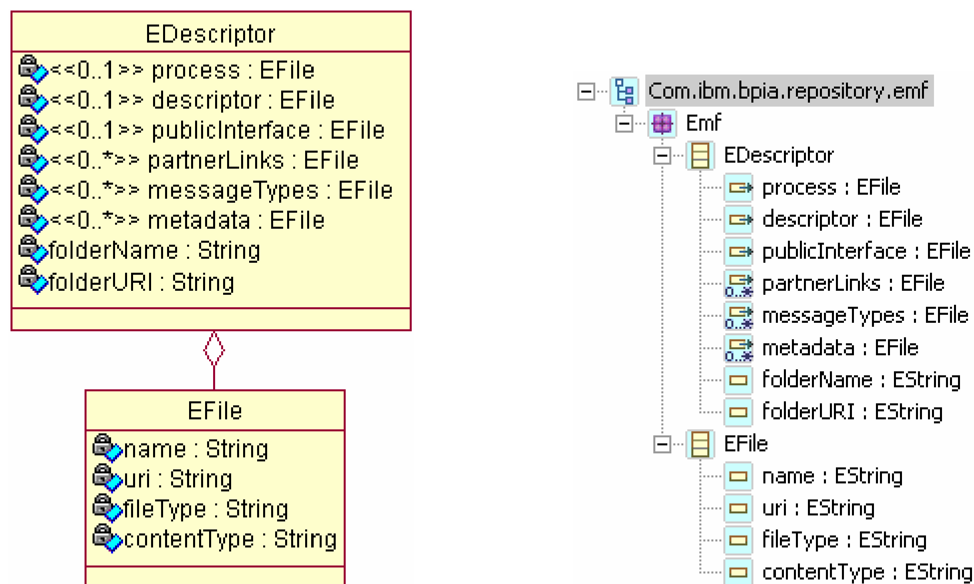

Figure 4.9: The descriptor file contents are designed as a UML model, which is converted to an EMF model. This is an example of the EMF model transformation capabilities.

4.4.2 Extensibility to Service Data Objects

The Eclipse Modeling Framework provides another powerful service for the persistence of data objects. The EMF extension Service Data Objects (SDO) functions as a data mediator service for data objects. The principle idea is to offer a common interface for software using SDO to store its data objects regardless of the medium, where the objects are persisted. This enables the data storage to be changed without modifying the program using the SDO services. SDO takes care of the different data storage mediums and how they are accessed.

An example of the use of SDO is using a file system as the first data storage, and changing to a database when the scalability requirements become more critical. This is the case with the BPEL repository, as it is implemented on top of a file system. When the repository architecture was designed SDO was not mature enough to be integrated. For example, there are no database drivers available for SDO. However, the interface for the SDO access is already known. The architecture is designed in a way that allows the current data handler component to be potentially replaced by an SDO data handler component. It is thus also possible to change the underlying data storage medium to one which supports SDO.

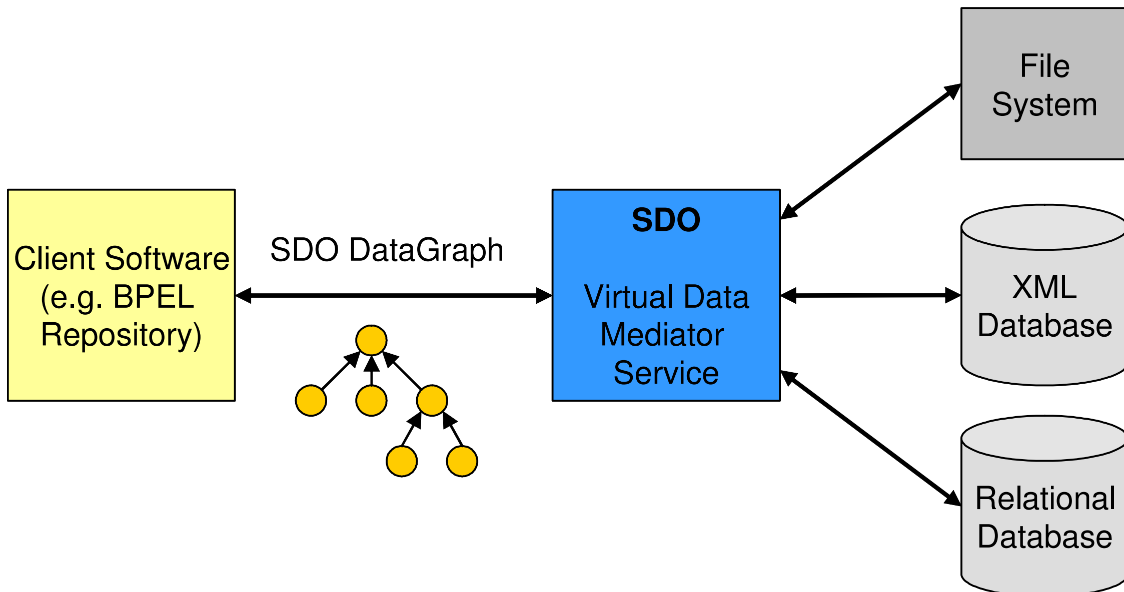

Figure 4.10: Service Data Objects (SDO) framework as a data mediator service for different data storages. [Ecl04b]

4.5 Query Mechanism

The BPEL repository provides a query mechanism, which enables querying the object model of files in the repository. It is implemented by loading a queried file in memory as an EMF object. The EMF object is handed over to a query engine together with query parameters. The query engine executes the query for the object and returns the result to the repository logics component. The repository logics component continues by iterating this process through all the relevant files inside the specified query scope. The query process is illustrated in Figure 4.11. In the following sections, the query mechanism is used in co-operation with the OCL Tool from the University of Kent.

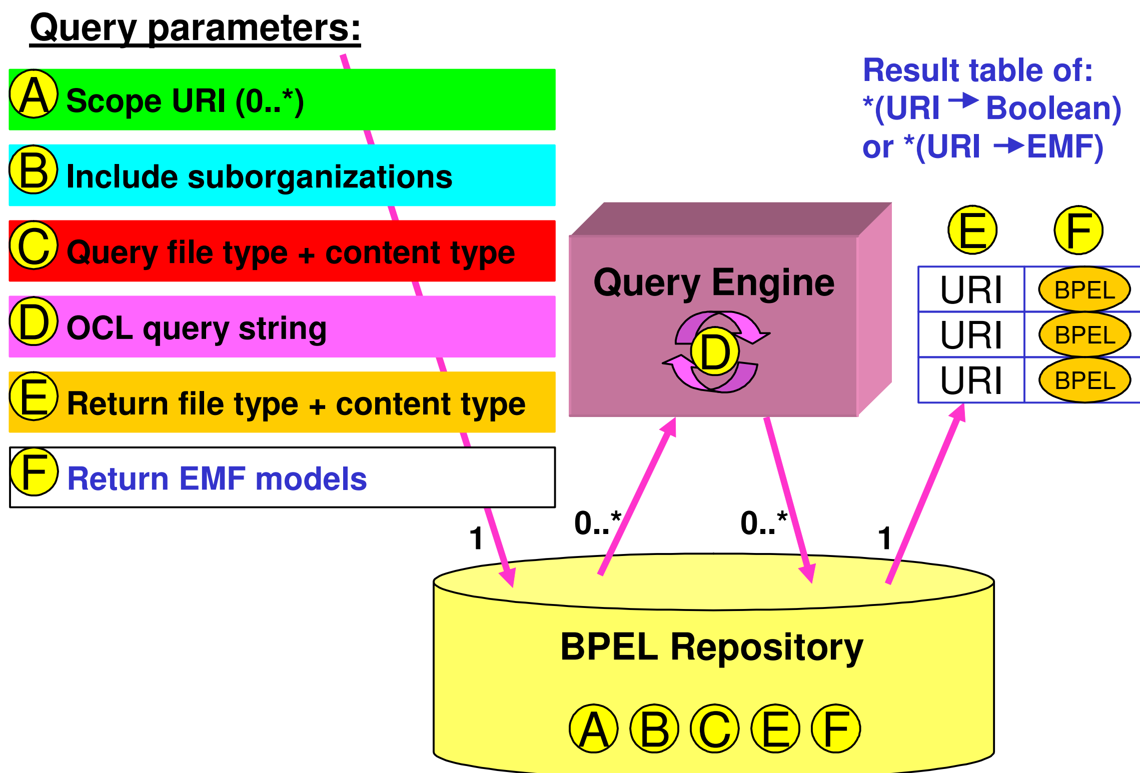

Figure 4.11: The querying mechanism of the repository. There are six query parameters. The parameter (D) OCL query string, is passed to the query engine, the others are used inside the BPEL repository. Parameters E and F determine the output types in the query result table.

4.5.1 Query Parameters

There are six parameters for a query. They are identified with letters (A, B, C, D, E and F) in the Figure 4.11 above. The parameters (A, B, C) are used to determine the files, which are sent to query engine as EMF objects. The parameter (D) contains the parameters for the query engine. The parameters (E, F) determine the choice of the query output.

A) **The query scope identifies organizations, where the query is applied.** It is possible to limit a query to a sub-scope of the repository organization tree. In this case, only the files inside the query scope are queried. The scope is determined by specifying one or more organization URIs, where the query will be executed.

B) **The include suborganizations Boolean variable determines if the query scope includes the subfolders for each URI specified in the query scope (A).**

  a) **True (default)**: All the subfolders are included for each URI in the query scope parameter. This is illustrated in the right-hand side of Figure 4.12.

  b) **False**: Only the organizations that have their URI in the query scope parameter are included. This is illustrated in the left-hand side of Figure 4.12.

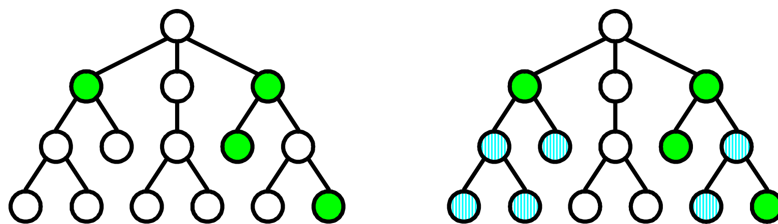

Figure 4.12: In the left tree, the green nodes represent the organizations, whose URI is in the query scope parameter (A). If the parameter (B) Boolean is true only they are included in the query. If the parameter has the Boolean value true, also their suborganizations are included in the query. This case is illustrated on the right tree, where the lightly colored child nodes are also queried.

C) **The query file type and the query content type identify the file(s) in the organization, which are queried.** For the query file type, five options are available: BPEL, WSDL, XSD, descriptor or metadata. It is a choice from a limited set of file types. The query content type parameter further refines the query. For example, a WSDL file can be a public interface or a partner link file for the BPEL file in the same organization. Content type can be any string, so it supports a broad range of metadata types. The query file type and the query content type combinations help the querying mechanism load only files relevant for the query. For example, if there is an XML metadata file describing execution properties of a BPEL file and a separate XML metadata file describing the purpose of the BPEL file, the content type is used to identify the contents of the file. It does not make any sense to look for a purpose of the BPEL file from an XML metadata file, which only contains execution time information, or the other way around. This approach does not rely on the file naming policy.

D) **The OCL query string specifies the query that is executed by the query engine for each EMF object.** It is analogous to an SQL query string. Actually, the parameter D is a map of parameters for the query engine. They are entered as a key-value pair in the map. This approach allows the query mechanism to pass to each query engine the parameters it requires. In case of the OCL Tool, the OCL query string is inserted in the parameter map. The parameter map is illustrated in Figure 4.13.

key	*value*
KEY_{OCL}	OCL context & expression

Figure 4.13: The parameter map passed to the query engine contains key-value pairs. In the case of the OCL Tool, key is a string identifier for the OCL string parameter and the value contains the OCL context & expression string.

E) **The return file type and the return content type specify, which file is returned from an organization, which gave a positive query result.** This allows the system to return files associated with the queried file. For example, if a user wishes to retrieve a BPEL file that has a certain kind of data in its metadata file, a query will be targeted for the metadata files and a BPEL file will be returned for each positive query result.

F) **The return EMF models Boolean variable determines what is included in the result table if the query result is positive.**

   a) **True:** For each queried file URI to the file that is wanted to be returned is inserted in the result table. The EMF object from the file is inserted as the value for that key. This is illustrated in the right-hand side of Figure 4.14.

   b) **False (default):** For each queried file URI to the file that is wanted to be returned is inserted to the result table. The query engine result (usually Boolean true, but can also be part of the queried EMF model depending on the query) is inserted as the value for that key. The case using Boolean variables is illustrated in the left-hand side of Figure 4.14.

4.5.2 Query Result Table

A query to the repository returns a hashtable as a result. The URI of each returned file is stored in the table. The linked value to the key is either a Boolean variable or an EMF object depending on the query parameter (E) and the query engine return value. The query result table is illustrated in Figure 4.14.

key	*value*
URI_1	$true_1$
URI_2	$true_2$
URI_3	$true_3$

key	*value*
URI_1	$BPEL_1$
URI_2	$BPEL_2$
URI_3	$BPEL_3$

Figure 4.14: The query result table is a Java Hashtable object. A return file URI is stored as a key to each entry, and its value is either a Boolean variable or an EMF object.

4.5.3 Querying Algorithm

The query mechanism is illustrated in detail in Figure 4.15. It is a refined version from the previously presented Figure 4.11.

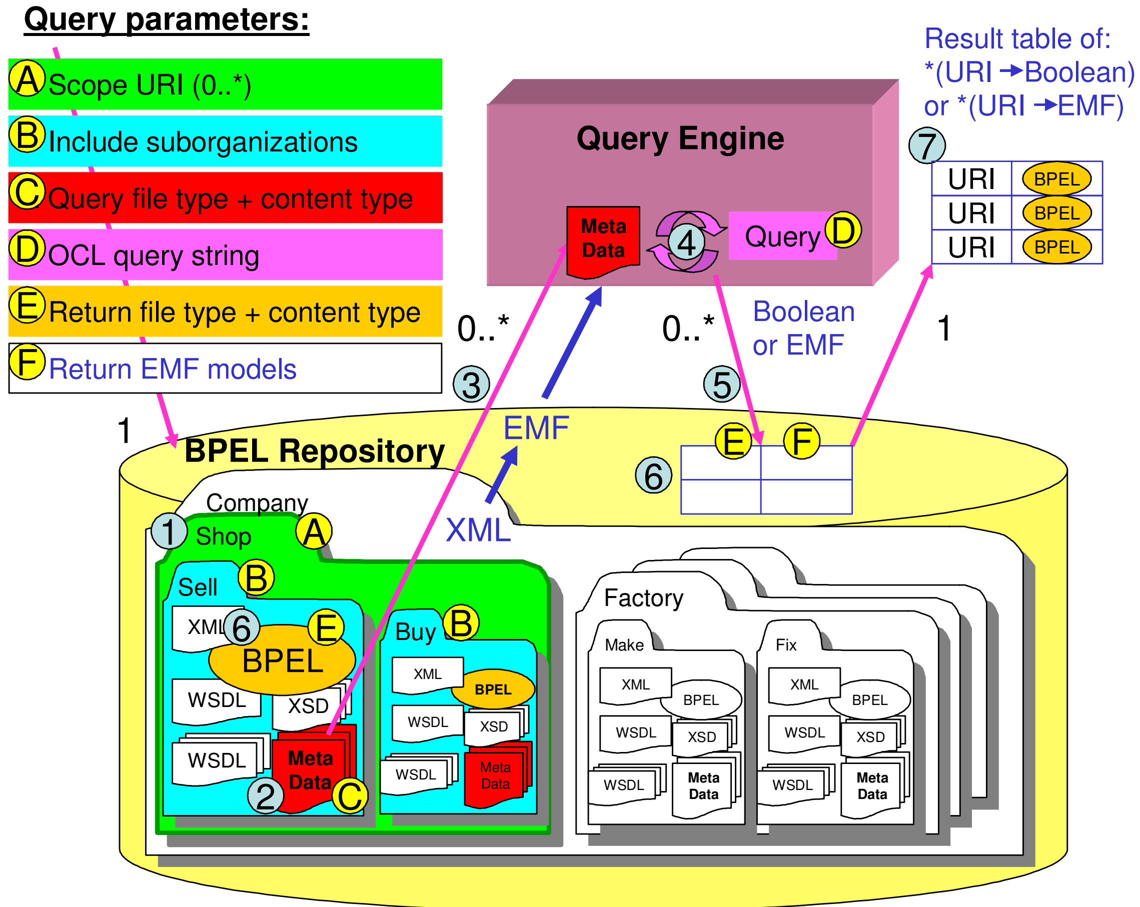

Figure 4.15: The detailed query mechanism. The query parameters are labeled with a character and colored as the objects to which they refer. The query steps are numbered.

The query parameters are used in a query process having the following seven steps:

1) The query first finds the query scope (parameters A and B) from the repository. For each organization inside the scope, the steps 2-6 are executed.

2) A file matching both the query file type and the query content type (parameter C) is searched in the organization descriptor file. If there are several files that match in a single organization, all matching files are queried (steps 3-6).

3) The file contents are loaded into an EMF object, which is sent to the query engine.

4) The query engine executes the OCL query (parameter D) on the EMF object containing the file contents.

5) The result of the query is returned from the query engine to the repository logics component. The returned result may be a Boolean variable or an object,

depending on the OCL query. For example the query might be may check, whether the object matches the query, then the result is a Boolean variable. Alternatively, a query may retrieve a subpart from the queried object, and then the result is an object. Since a Collection Java class is also an object, the result can be a collection of objects.

6) The results of the objects from each organization are collected together into the result table. There are several possibilities:

   a) If the result of the single object query is *false*, no result is added to the query result table.

   b) If the result of the single object query is *true*, and the query file type and query content type (parameter C) are exactly the same as the return file type and return content type (parameter E), the URI of the queried file is added to the result table as a key element. If the user requires the EMF object by setting parameter (F) to true, the EMF object is added as the value for the key in the result table. Otherwise Boolean variable true is put as the value.

   c) If the result of the single object query is *true*, and the query file type or query content type (parameter C) differ from the return file type or the return content type (parameter E), the file(s) in the organization matching the return file type and the return content type, the file(s) are added to the result of the query. The URI of each returned file is added to the result table as a key element. If the user requires the EMF object by setting parameter (F) true, the file is loaded as an EMF object and added as the value for the key in the result table. Otherwise Boolean variable true is put as the value.

   d) If the result of the single object query is an object, it is added to the result table as a value. The key for the value is the file URI that matches the return file type and the return content type (parameter E).

7) The result table is returned from the BPEL repository as the complete answer to the query for the whole query scope.

4.5.4 Query Performance Optimization Approaches

This query mechanism is expected to be rather heavy when executed over large collection of files. First of all, each file is loaded completely in memory as it is sent to the query engine. Secondly, this approach does not use any indexing mechanism present in database systems in general. In order to solve the scalability issues with the expected repository performance, the following performance optimizations are implemented.

4.5.4.1 Minimizing the Number of Loaded Files

It is often the case that a user does not need to query through all the files in the repository in order to find the file he is looking for. Thus, the repository supports narrowing the query by specifying a scope. This allows the user to specify the

organizations, which he wants to search. This approach allows processing power to be used more efficiently, since only necessary files to a query are processed.

Another technique to reduce the number of files in a query is to load only the files that contain the right kind of EMF object. For example, if the search target is a BPEL process object, only BPEL files are loaded for the query. The same applies to WSDL files, XSD files, and other XML metadata files. The repository stores a file type and a content type for each file in a XML descriptor file of the file's organization. This file type together with the content type tells which EMF model is used to serialize the objects. Thus, by loading only the files with the exact file type and content type specified in the query parameters, only the right kind of files are loaded from the data storage. This decreases the number of times the hard disk is accessed, which are usually time-costly operations.

The drawback of this approach is that in each organization in the query scope the descriptor file is accessed with each query in order to find the relevant files for the query. Each time at least one file per organization is accessed in addition to the queried files, but no other irrelevant files are read. In the case of a BPEL file query, this mechanism saves approximately a dozen of files per organization from an unnecessary loading operation.

4.5.4.2 Minimizing Memory Requirements Using Iteration

In order to reduce the main memory requirements when the BPEL repository is queried, the query logics component iterates in a one-by-one manner through the relevant files inside the query scope. This means that files are loaded one-at-a-time and sent to the query engine. After the query engine has returned the query result, the references to the EMF object are deleted if the EMF object is not returned as a part of the query result. This allows the Java garbage collector to free the memory for reuse. After one file is handled the next one is loaded and sent to the query engine. This iteration saves runtime memory, since objects are in the main memory only one-at-a-time.

4.5.4.3 References to the Organizations and Files in Hashtables

The BPEL repository utilizes Java Hashtable objects as the mechanism to find objects in the memory efficiently. Hashtables have constant time insertion, deletion and search operations, which provide a well-scalable search method when the number of objects in the hashtable increases.

The repository has a common hashtable containing a reference to all of its files and organizations. The Uniform Resource Identifier (URI) serves as a hashing key. A key points to the organization itself or the organization where the file is found.

In addition, each organization has a hashtable containing a reference to its suborganizations and another hashtable for its files. An organization name or a filename serves as a key in these cases. Moreover, each organization has a reference to its parent organization. These search mechanisms provide robust navigation in and through organizations, for example, in the case of a query with an organization subtree as the query scope.

5 Repository Performance Measurements

In order to measure the scalability and performance of the repository, a set of tests to profile the implementation was planned and executed. The first section 5.1 introduces the profiling environment. It is followed by repository start-up performance tests in section 5.2. Finally, the querying performance is inspected in section 5.3.

5.1 Profiling Environment

A profiling toolset from the Hyades project [Ecl04c] was initially used to profile execution time performance and memory usage. It is an Eclipse workbench plug-in extending its functionalities to runtime data collection. The profiling toolset allows detailed performance measurements of Java applications to be logged.

However, executing the Hyades data collection engine slows down the tests significantly, since it logs all the function calls and memory used by each object. For this reason, after the BPEL repository performance was optimized with the Hyades tool, the following test scenarios were executed without the Hyades tool. Instead, a minimum number of timestamps was recorded and printed on the screen as the test scenario advanced. This allowed observing the memory usage of the tests during the different phases of the execution with the Windows Task Manager. The plans of the relevant test cases, as well as the results from the measurements, and their analysis are introduced in the following sections.

The test cases are executed on an IBM ThinkPad A31p laptop having 1.00 GB of main memory and an Intel Pentium 4 (1.70 GHz) processor. The profiling tests are executed from the Eclipse 3.0 workbench with the following setups. A test class directly called the repository API methods. The amount of memory for the Java Virtual Machine is set to 1024 MB for the process. Only a minimal number of other applications is running together with a test, so the test process can reach almost 100% of the processing capacity.

5.1.1 Repository Contents Used in the Performance Tests

A file collection was gathered as repository contents for the performance tests. This collection contains BPEL files and related files. There are three sources for these files, our Business Process Integration and Automation project at the IBM Zurich Research Laboratory [Nov04], the Oracle-Collaxa website [Ora04] and the BP4J project of IBM.

All these files were collected together. All BPEL and other XML files conforming to their XML schemas were held in the contents, all the non-compliant XML files were deleted. Two example repository contents A and B were created:

A) A primary set of files presenting an ideal collection of files for the BPEL repository.

- o The subfolders were deleted if they did not contain BPEL files or XML files in them or deeper in the organization tree related to them.

B) A secondary set of files presenting a large heterogeneous collection of files for BPEL repository.

- o On contrary to the primary collection (A), these files included also organizations containing less directly related files to BPEL files, such as Java and JSP files providing implementation for BPEL related Web services.

Table 5.1 illustrates the differences between the example repositories A and B.

Table 5.1: The differences between the example repository contents used in the performance tests.

	Repository contents A	**Repository contents B**
Number of folders	221	534
Number of files	1171	2031
Number of BPEL files	165	165
Size	1.85 MB	3.62 MB
Size on disk	5.08 MB	9.42 MB

5.2 Repository Start-Up Performance Test Cases

The following sections present each a set of tests, which are executed in order to measure the performance of the repository API plug-in. Each test set is described individually followed by a hypothesis formulating the expected performance.

In order to query the repository, two computations must be performed, and they can be executed separately.

1) First, the repository must be started to load the folder and file structure of the repository. Once this is done, the repository is ready to be queried. If the repository is queried several times consecutively, the loading must be performed only once.

2) The second computation is the query itself. It includes loading the descriptor files in the query scope and locating relevant files based on this information. Then, these files are loaded and sent to the query engine, which returns the results of the query. Based on these query engine results, the repository returns URIs and possibly EMF objects depending of the query parameters.

This section 5.2 concentrates on measuring the start-up performance and it is followed by the section 5.3 testing the scalability of the querying mechanism.

5.2.1 Repository Start-Up Time as a Function of the Repository Size

The overhead of starting the repository is measured in this test. The test is executed for several multiples of the repository contents A presented previously in chapter 5.1.1. The repository contents are multiplied 1, 2, 4, 8, 16, 32, 64, 128 and 256 times for the tests. The following three scenarios are tested:

1) The repository contents have not yet been loaded before and they do not have the BPEL repository specific descriptor files. Therefore:
    - o Descriptor files are created automatically for each organization.
    - o The files are not yet in the computers main memory.
2) The repository contents are not loaded recently, but descriptor files already exist.
    - o The files are not yet in the computers main memory.
3) The repository contents have been loaded recently, and descriptor files exist already.
    - o The files are already in the main memory of the computer.

The computer is started just before the first two tests to ensure that the relevant files are not loaded in the main memory before the tests. This approach is followed in any of the subsequent tests sharing the same assumption.

Hypothesis

i) The start-up time increases linearly with the repository size.

ii) Significantly more time is consumed when the files are not in the main memory, as reading data from a hard disk is a much more time consuming operation than accessing the same data from the main memory.

iii) If the descriptor files are not already available additional time is needed for building them, but it is also linear.

Results

Figure 5.1 shows that more computing time is needed to start-up the repository as the repository size increases. In addition to observing the processing time, the CPU usage rate of the repository process is logged. In test cases (1) and (2), the files are not in main memory and the CPU usage is mostly below 50%.

In test case (3), the files have been loaded previously and thus are already in main memory. In this case the process is utilizing most of the time 99% of the processing power.

However, this behavior changes when the repository size grows over 32 times of the original content collection A. It seems that instead of reading large amount of files from the main memory, the hard disk is accessed. This is sure for the largest size of the repository, since the overall memory usage went up to 1.5 GB, but the computer has only 1.0 GB of main memory. This means that the operating system is swapping the main memory to the hard disk, in order to fulfill the processing memory demand exceeding the main memory size.

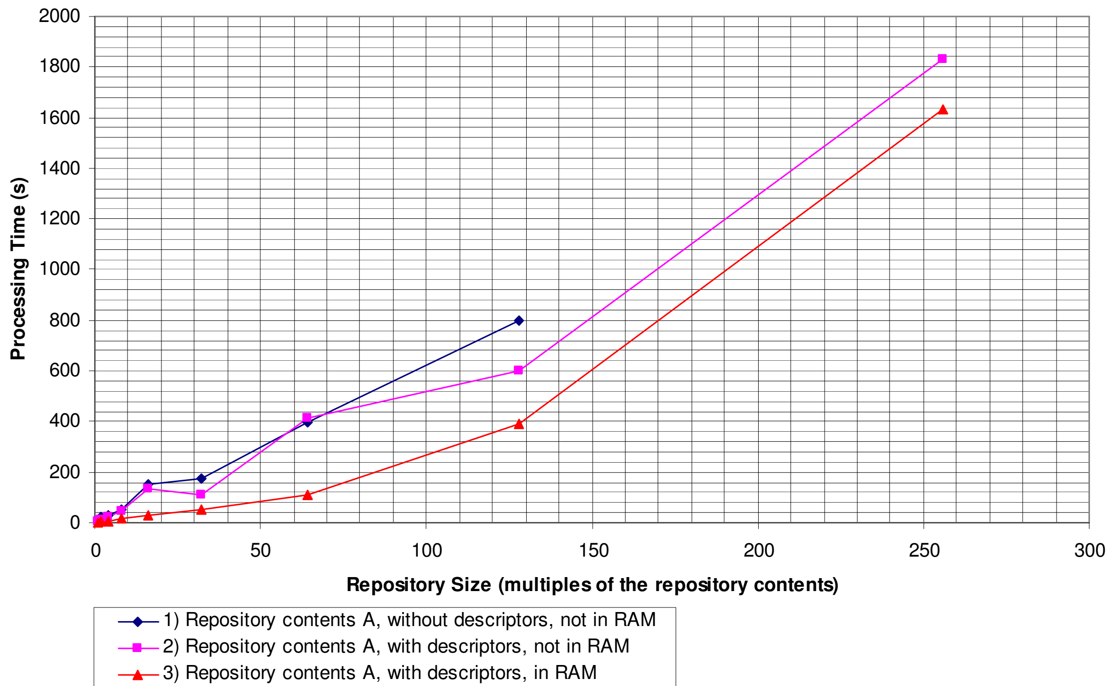

Figure 5.1: Repository start-up time as a function of the repository size.

Analysis

Hypotheses (i) and (ii) are correct as long as the repository size keeps below an order of 64 times the repository contents A. The file loading overhead from the hard disk is significant, since roughly twice as much time is needed as if the data is read from the main memory.

Hypothesis (iii) is correct; the utilized processing time is linear compared to the repository size. The extra effort needed to create the descriptor files is not overwhelming compared to the other work.

However, when the repository size is bigger (starting from size 64) hypothesis (i) is not correct anymore. The files are not handled solely from the main memory and the needed processing grows faster than linear. In addition, case (3), where the files were already accessed recently before, gets more close to case (2), where the files are not

already in the main memory. The obvious reason is that if the files are read from the hard disk anyway, it is not faster to read them from the hard disk swapped from the main memory than from the files themselves in the main memory.

Nonetheless, as the files were accessed before, case (3) is still faster, because most probably the most recently accessed data is still kept in the main memory. This data can be for example folder data, which would not be swapped to the hard disk, but it is still rather kept in the main memory, since the data is often accessed compared to the files accessed mostly only once.

5.2.2 Repository Start-Up Time with Different Repository Contents

In the following test scenarios, the preceding tests are applied to the repository contents A and B. The tests 1A, 2A and 3A are the same tests as previously presented 1, 2 and 3. The tests 1B, 2B and 3B are the same tests with the repository contents B.

In addition, the tests are limited to observe the behavior in a subinterval of the previous test. The repository size is increased up to 32 times of the example repositories A and B. This size was selected as the repository scalability stays linear in this interval, because the computer is still capable of keeping the data in its main memory.

In the case of repository contents A, this maximum size is still sufficient for the planned purposes of the repository, since A includes over 37 000 files and over 7 000 folders. They need 59 MB of memory and they take on the hard disk 162 MB of memory. More memory is needed on the disk, since most of the files are smaller than the minimum file size in the Windows XP operating system with a 30 GB hard disk as a single partition. The repository size well exceeds the size of any BPEL collection that is publicly available nowadays.

Hypothesis

i) The start-up time for the repository B is supposed to be around twice as long as for the repository A, because repository B uses twice as much hard disk memory and has twice as many files and organizations than repository A.

Results

The results from these six different tests with multiple repository sizes are presented in Figure 5.2. Test results from scenarios 3A and 3B are an average from three consecutive executions. No significant variation is noticed between the execution times with the same parameters. The other tests are run only once being more difficult to execute, because the computer has to be restarted before rerunning the test. The computer is restarted to ensure that the files are not yet in the main memory. In addition, the loading time from the hard disk depends more on the retrieval speed of the hard disk and the efficiency of the file locations in the hard disk than the functionality of the BPEL repository. Thus, the cases 3A and 3B illustrate the reachable lower limit of the start-up time.

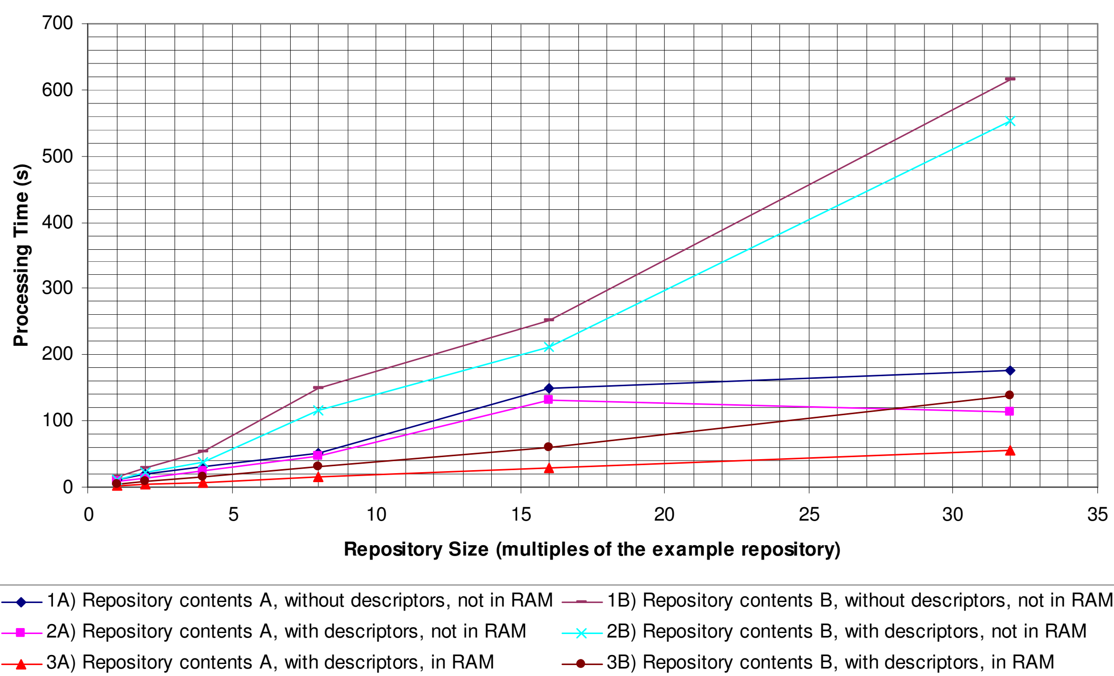

Figure 5.2: Repository start-up time with repository contents A and B.

Analysis

The hypothesis is correct, executing 3B takes roughly twice longer than executing 3A. The same applies to the tests 1A and 1B, as well as to 2A and 2B, respectively. All the repository start-up times grow linearly as the repository size increases.

The scenarios 3A and 3B have more interesting graphs. They illustrate the lower limits of the repository loading times with different contents. The repositories B have many irrelevant files for the querying purposes of the BPEL repository having double as many files as the repositories A. The larger content doubles the start up time. The most likely reason is that at each size the repository B contains twice as many folders as repository A, because each folder is accessed to find out the filenames it contains.

The repository start-up time becomes significant as the repository size increases. It is not reasonable to wait some tens or hundreds of seconds before each query as the repository hashtables are built up. For this reason, it is rewarding if the repository is started up once and let run, if other software or user needs it consecutively when it is needed consecutively by other software or a user. These data structures are necessary to provide robust querying mechanisms into the repository. The drawback of starting up the repository is the memory consumption, which is surveyed in the following test scenarios.

5.2.3 Repository Start-Up Memory Usage

The repository start-up memory usage is measured under the same test scenarios that were presented in the previous sections. The memory is mainly used to build up the structure of the repository contents to make the queries faster to execute. The memory is

reserved since the repository start-up. The memory consumption is the drawback of keeping the repository alive between queries. In any case, the data structure is also built up for a single repository query.

Hypothesis

i) The repository start-up memory usage grows linearly as the repository size increases, because the number of folders is directly dependent on the repository size in the test scenarios.

ii) The querying memory consumption with repository B is twice as large as with repository A.

Results

Figure 5.3 below shows the start-up memory usage of the repository. All the tree test scenarios behave similarly. The only differences are visible with extremely big repository sizes 128 and 256 times the example repository, where the test scenario 3 uses 1-4 MB more memory than in the cases 1 and 2, which use identically memory.

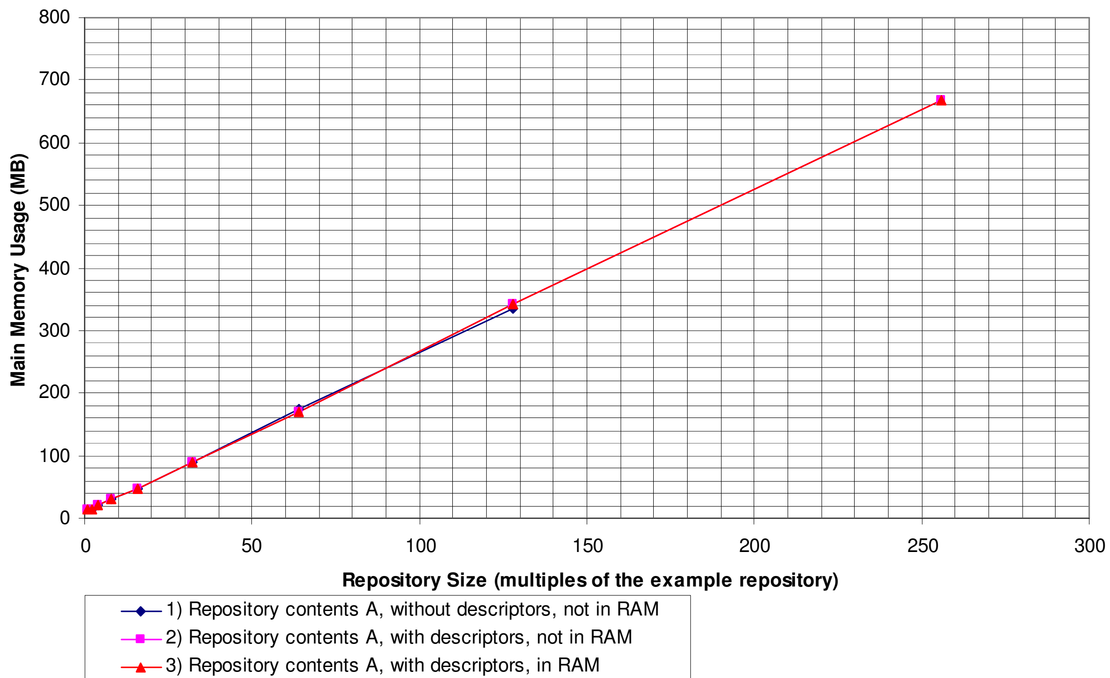

Figure 5.3: Repository start-up main memory consumption with the various sizes of repository contents A.

The next Figure 5.4 below illustrates the start-up memory consumption for the two different repository contents A and B. With repository contents B, there is slightly bigger memory consumption if the descriptor files are built at the start-up time.

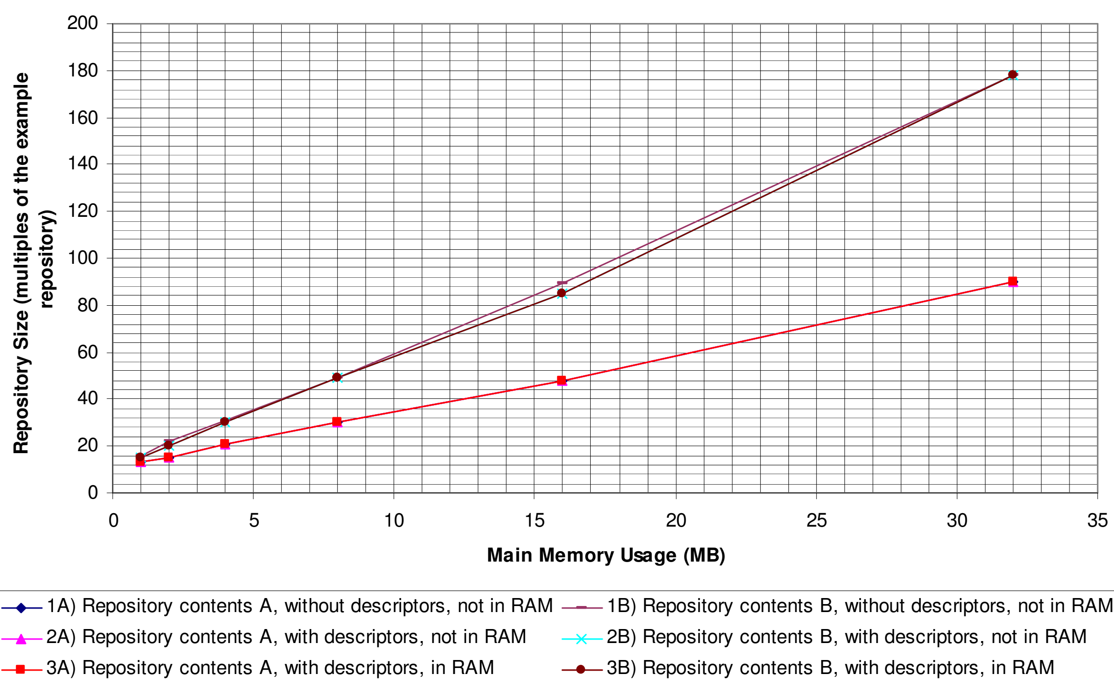

Figure 5.4: Start-up memory usage with various repository sizes and different repository contents A and B.

Analysis

Hypothesis (i) is correct; the start-up repository memory consumption is linear in the repository size. This holds well even if the repository start-up time starts to grow faster (repository sizes 64, 128, 256 in Figure 5.4), when the repository size approaches the size of the main memory. The order of magnitude of the main memory usage is not significantly larger, when the descriptor files are also created during the first repository start-up.

As hypothesis (ii) forecasted, the memory consumption with repository B is almost twice as large as with repository A. The data structure for repository B keeping files and organizations in memory must be twice the size of the structure for A.

5.3 Querying Performance Test Cases

In the following subchapters, the scalability of the repository querying mechanism is under evaluation. The test scenarios are set up to evaluate how large collections of files can be efficiently queried, when they are first deserialized from files to objects. Another goal is to find out how the increase in the repository size affects its scalability. It would be interesting to know how many files can be queried within a tolerable querying time period.

The test scenarios use the OCL Tool from the University of Kent as the query engine. The same repository contents A and B are utilized as in the previous tests.

5.3.1 Scalability of the Querying Mechanism

The first experiment measures the query processing time as the repository size increases. The repository contents A introduced in section 5.1.1 are utilized in the tests. The relevant details for the query tests are presented in Table 5.2 below.

Table 5.2: Number of folders and files in the test repository contents, which are built by cloning repository contents A multiple times and forming larger content collections by grouping them together in order to have larger test sets of files.

Size of the contents (multiples of contents A)	Folders	Files	BPEL files	Size (MB)	Size on hard disk (MB)
1	221	1 171	165	1.85	5.08
2	442	2 343	330	3.70	10.1
4	881	4 682	660	7.41	20.3
8	1 763	9 363	1 320	14.8	40.6
16	3 524	18 725	2 640	29.3	81.2
32	7 051	37 451	5 280	59.4	162
64	14 101	74 901	10 560	118	325
128	28 202	149 202	21 120	238	650
256	56 405	299 605	42 240	477	1 290

All these repositories are queried with a simple Object Constraint Language (OCL) query:

```
context process::TProcess inv: not self.flow.oclIsUndefined()
```

This query is executed for all BPEL files in the repository. The query means: "Is the root element of the BPEL process activity tree a flow?" *TProcess* refers to the BPEL EMF object, which corresponds to *process* XML element in BPEL files (see the BPEL example in section 2.1.1). In the test repository contents, the *process* XML element can have a *sequence*, *flow* or *pick* element as a child element. Whereas in the EMF object model, each *TProcess* object can have an activity tree, which can contain either a *sequence*, *flow* or *pick* as a root object of the tree. We are interested to find all the BPEL files that have a *flow* as a root. If the *flow* contains *null*, then *oclIsUndefined()* returns *true* for the *flow* attribute. If it is not *null*, it exists. Negating this with the *not* operator, the OCL query returns *true* when there is a *flow* as the root object.

There are 165 BPEL files in the repository A. Three of them have a *flow* as the activity tree root element, 161 have a *sequence* element and one of the files has a *pick* element. The numbers of the BPEL files matching the query in the bigger repository contents are presented in the last column of Table 5.3. For all the BPEL files that match the query, a URI to the file is added to the query result table.

There are two different cases for the test queries in the first test scenario:

1) The BPEL files have not been queried recently and thus the operating system has not loaded them yet from the hard disk to the main memory.
2) The BPEL files have been queried recently and thus, they are already in the main memory.

The second test experiment applies this same query to the repository content B, and compares it to the previous results of the repository contents A. The content collection B has the same BPEL files as A, thus the query results are equal. The difference between the test cases is that content collection B has roughly twice as many non-BPEL files as A and it has twice as many folders as well. Thus, this experiment measures whether it makes any difference in the query speed if there are additional files and folders in addition to those, which are the target of the query. The differences of the test repositories A and B are presented in the Table 5.3 below.

Table 5.3: A comparison of the differences between the repository contents A and B with the used test repository sizes.

Size of the contents (multiples of contents A or B)	Folders in A	Folders in B	Files in A	Files in B	BPEL files in both A and B	BPEL files with a flow in both A and B
1	221	534	1 171	2 031	165	3
2	442	1 068	2 343	4 062	330	6
4	881	2 134	4 682	8 122	660	12
8	1 763	4 267	9 363	16 243	1 320	24
16	3 524	8 532	18 725	32 243	2 640	48
32	7 051	17 067	37 450	64 971	5 280	96

The third test experiment uses the same test cases as the first one, but instead of measuring the processing time needed, the necessary main memory for the query is observed, in addition to the memory occupied already at start-up time. The purpose is to find out the required main memory that must be available for efficient querying with different repository sizes. The total main memory needed is also observed. The proportion of the memory occupied already at the start-up time is measured separately of the proportion used to query the BPEL files.

The fourth test experiment explores the memory consumption differences between repository contents A and B. The same test conditions are used as in second test.

Hypothesis

i) The processing time increases linearly as the repository size increases. All the files in the scope are loaded and queried iteratively. It should not affect to a single file how many other files are iterated through. The processing time might increase slightly faster than linearly, because of the bigger indexing data structures, but not at the same rate as the number of files increases.

ii) The query time is larger if the files are loaded from the hard disk instead of the main memory.

iii) A query in repository B takes longer than a query in repository A.

iv) Used memory needed for a query is constant, since files are loaded in the main memory iterating them through one-by-one. The required memory depends on the file size.

v) Memory for repository indexes increases proportionally as the number of files and folders in the repository increases.

5.3.1.1 Results of the Query Processing Time Tests

Figure 5.5 illustrates the processing time of the repository query under various repository sizes.

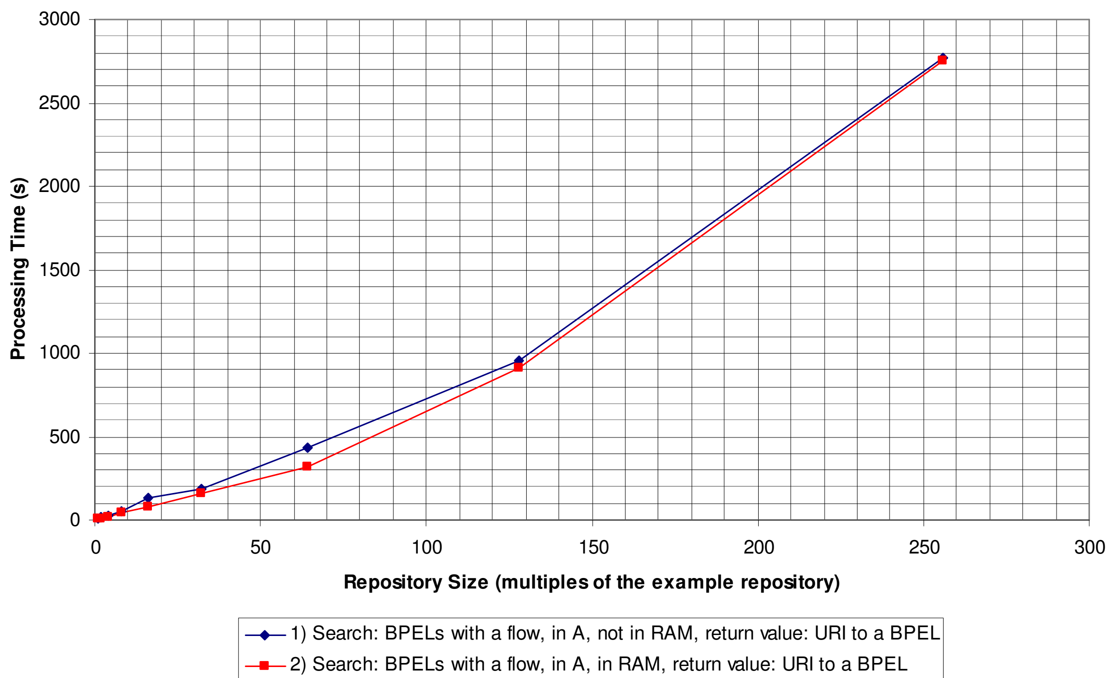

Figure 5.5: Limits of linear scalability for the repository querying mechanism. Query finds 3 BPEL files per each content collection A, which contains 165 BPEL files, thus 1.8 percent of the queried files match the query.

In addition to the information visible in the figure, the behavior of the CPU usage changes if the repository size is over 64 times the content collection A. If the files are used recently and thus, still in the main memory with the smaller repository sizes, CPU usage reaches almost 100%. If the repository size is bigger, the CPU usage drops under 50% and the rest of the time is used by idle process of the operating system.

Figure 5.6 shows the same the test results as above, but it restricts to the repository sizes from 1 to 36. In addition it shows a behavioral difference between repository contents A and B.

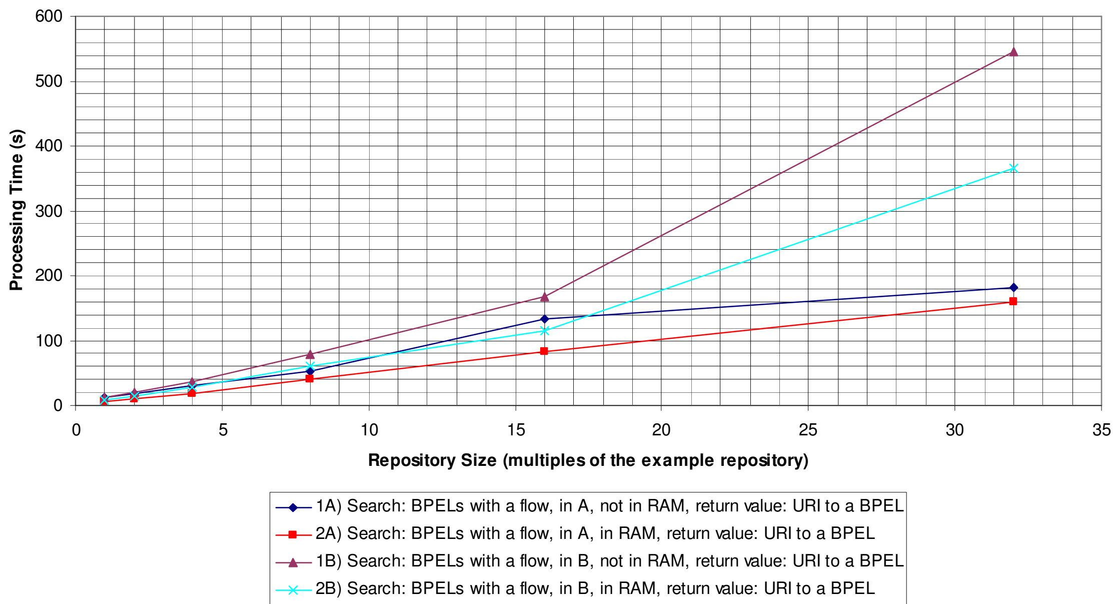

Figure 5.6: Processing time usage for a BPEL query with repository contents A and B. This figure focuses to show concentrates the results with small repository sizes having querying time less than few hundreds of seconds.

Analysis

With repository contents A, the querying time remains linear until the repository content size 64. Hypothesis (i) is accurate for these small repository sizes. In the limit size, the contents include 14 000 folders, 75 000 files, and 10 500 BPEL files. As it takes 5 minutes to query through 10 000 BPEL files, or 30 seconds for 1 000 BPEL files, it can be stated that the query performance is suitable for the purpose it was designed.

There is only a slight difference in querying time depending on whether the queried files are loaded in the main memory already before, or not. Hypothesis (ii) is correct, but the extra need of time is not significant.

As hypothesis (iii) states, querying the repository B takes twice as long than querying the repository A. Most probably, the key factor is the double number of the folders, since for each folder its descriptor file is loaded and references for a BPEL file are searched in it. With content collection B the change in the linear increase happens already before the repository content size 32 is matched. At this size, the repository contains 17 000 folders, 65 000 files and over 5 000 BPEL files. The change appears at around 15 000 folders for the repositories A and B, but it is assumed that the number of other files in the memory affect the limit as well.

Above the limit sizes, the query processing time grows faster than linearly, and hypothesis (i) is no longer valid. Thus, it can be said that the repository querying mechanism does not scale well enough for bigger contents. Beyond the limit, the operating system starts to swap main memory to the hard disk, or all the needed files are not kept anymore in the main memory. Instead, files are accessed again from the hard disk, even though they were recently used before. Whereas with the small repository

sizes, the CPU usage reaches almost 100%, with the bigger repository sizes it drops below 50%. The rest of the processing power is consumed by the operating system idle process. Thus, the whole CPU power cannot be utilized. This behavior can be improved by keeping more files in the main memory or having a faster hard disk drive. However, these options are beyond the scope of this work.

5.3.1.2 Results of the Main Memory Usage Tests

Figure 5.7 below shows the main memory consumption for the same test cases for which the processing time was illustrated in the Figure 5.5.

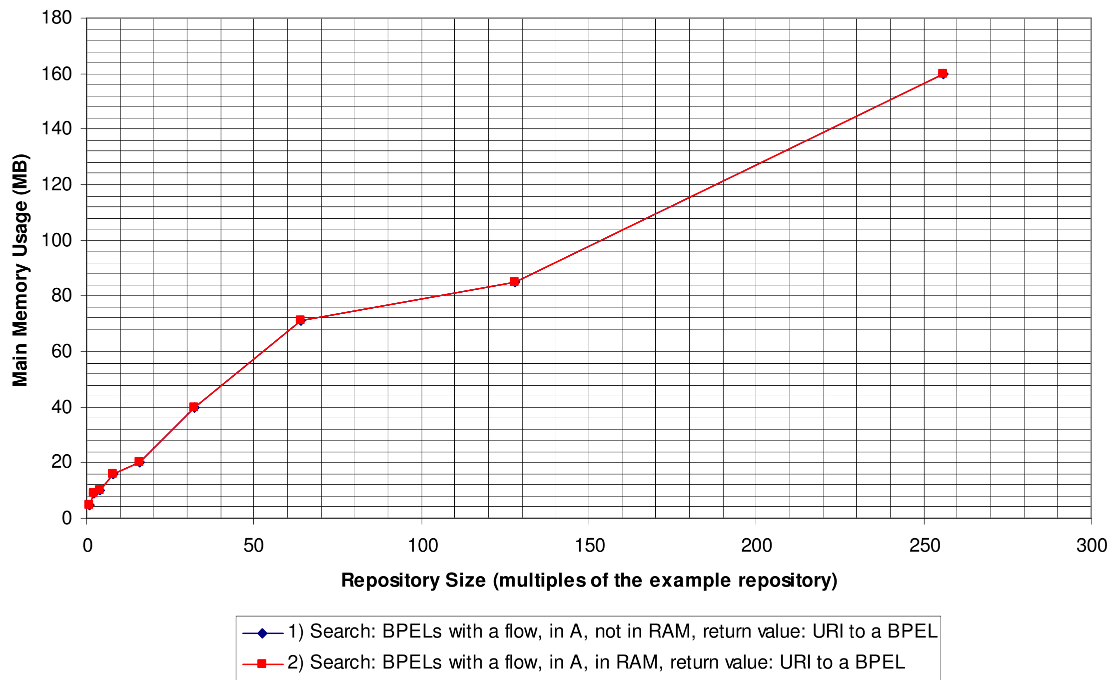

Figure 5.7: Repository main memory consumption with different repository sizes while querying all the BPEL files in a repository.

If the repository size is below 64 times the content collection A, the memory usage increases steadily throughout the querying period. Above that size the main memory usage increases most of the time, but time-to-time it dropped suddenly by several dozens of megabytes. This sudden drop seems to be affected by waking up the Java garbage collector. It removes all the instances, which do not have anymore references to them in the Java program.

The next Figure 5.8 shows results from the same case as the previous Figure 5.7. In fact, the red graphs (2 and 1Q) are the same. While the previous figure showed clearly the memory usage change after repository size 64, this following figure shows the total memory consumption of the BPEL repository when it is querying through the BPEL files. Graph 1 shows the total memory consumption, 1S shows the static part that is occupied since the start-up time and 1Q illustrates the extra memory needed to process the query. Thus, graph 1 is the sum of graphs 1S and 1Q.

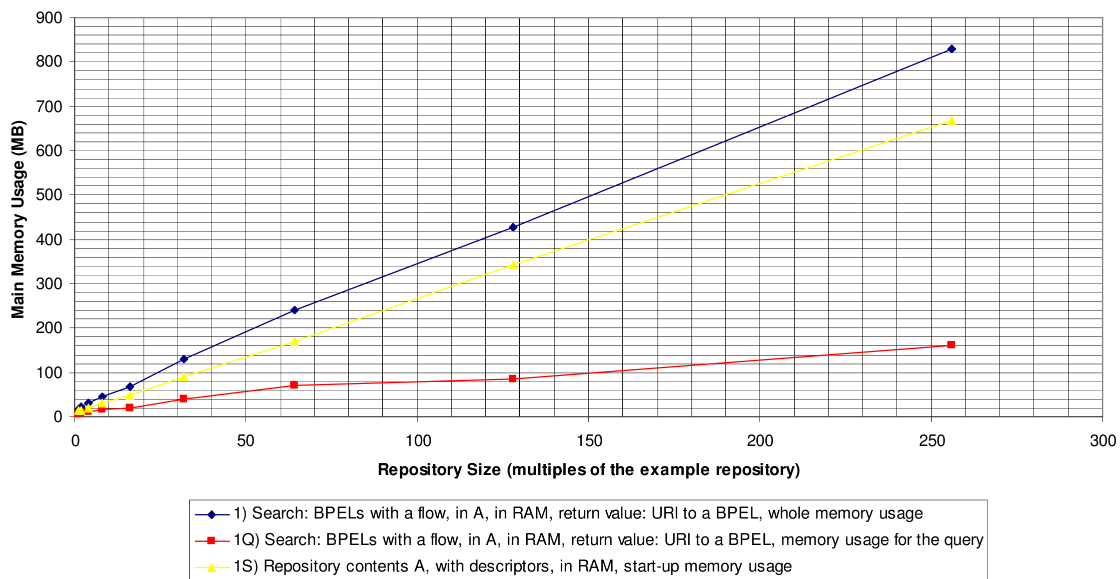

Figure 5.8: Repository main memory usage – the required repository start-up memory versus the extra memory requirements for querying BPEL files.

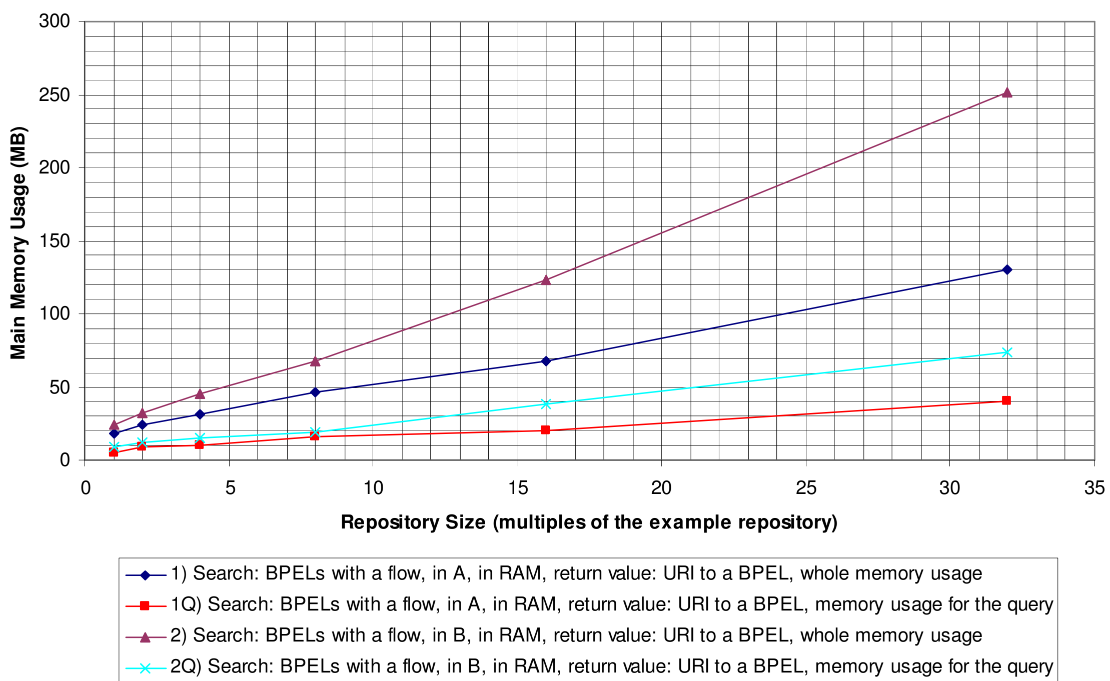

Figure 5.9: Memory usage differences of the repository contents A and B.

Figure 5.9 above plots the same graphs 1 and 1Q as in the previous Figure 5.8. The difference is that the graph concentrates on illustrating the memory consumption for smaller repository sizes than 64. In addition, it compares the memory consumption of the content collection A (graphs 1 and 1Q) to the content collection B (graphs 2 and 2Q). It shows a major difference between the repository contents even though both repositories contain exactly the same BPEL files if the repository sizes are the same.

Analysis

Unlike stated in hypothesis (iv), the main memory usage grows linearly especially with small repository sizes instead of staying constant. However, this behavior changes when the repository size grows. The graph curves down and the memory need increases slower than linearly compared to the repository size.

With the large repository, there are sudden drops of dozens of megabytes of the memory utilization. Otherwise the memory need grows steadily similarly to the small repositories. The Java garbage collector is invoked, when the computer is running out of main memory. This behavior suggests that by invoking the garbage collector with short regular intervals the memory usage could be kept constant.

However, the approach of invoking the garbage collector must have a hindrance of increased querying time, since the Java garbage collector uses its share of computing power. For this reason, the responsibility of invoking the Java garbage collector is left to the program that is using the repository API. If it prefers to use less memory with the expense of increased query time, it can invoke the Java garbage collector periodically while querying the repository.

In this testing environment, the main memory usage of the querying mechanism is at maximum around one half of the start-up memory usage. With larger repositories the need is even noticeably lower. Thus, the start-up memory need is still dominant compared to the query memory need. Of course this changes if the queried files are larger, or if there are proportionally more queried files in the repository.

The last Figure 5.9 reveals the important fact that the memory usage for a query is not solely dependent on the number of queried BPEL files. Both content collections A and B contain the same number of BPEL files with equal file content at each multiple of the content collections. Nevertheless, the memory need for repository B is nearly the double compared to the repository A. Thus, hypothesis (v) is correct. For example, at repository size 32, both repositories contain 5 280 BPEL files, which are queried. The difference of the repositories is the amount of folders in them. At repository size 32, repository A has over 7 000 folders, whereas repository has B over 17 000 folders. Per each folder, a descriptor file is loaded in the memory during the query, and references to the BPEL files in the folder are searched in it. Since this is done for repository B more than twice as many times as for repository A, repository B needs more memory.

There are also more files in repository B (at size 32, there are around 65 000 files) compared to repository A (around 37 500 files). The sizes of these files have no direct impact on the querying memory usage, since non-BPEL files are not loaded, when BPEL files are queried. This applies as well for other file types and file content types. Only the number of the irrelevant files slightly affects to the query memory consumption, since references to all files are examined in the descriptor files.

5.3.2 Scalability of the Querying Mechanism with Fixed Scope

This test scenario finds out, whether the concept of query scope helps to find results faster for a query. Instead of querying all the BPEL files in a repository, the query is narrowed to a subset of the organizations.

The test uses the same repository contents A as introduced before. For each size of the repository contents a subtree of file system folders is selected for the repository query scope. With each repository size, the subtree contains exactly the same files and folders. It is observed if the overall repository size affects the required processing time or main memory consumption, when the query scope remains equal.

URI to the BPEL files that match to the query are returned. The OCL query is the same as in the previous query test cases:

```
context process::TProcess inv: not self.flow.oclIsUndefined()
```

Hypothesis

i) The query time does not increase if the repository size increases, and the query scope remains constant.

ii) The extra memory required for a query does not increase if the repository size increases, and the query scope remains constant.

Results

Figure 5.10 presents the required processing time for the BPEL query with a narrowed scope. Regardless of the repository size, the same 3 BPEL files are returned as the result.

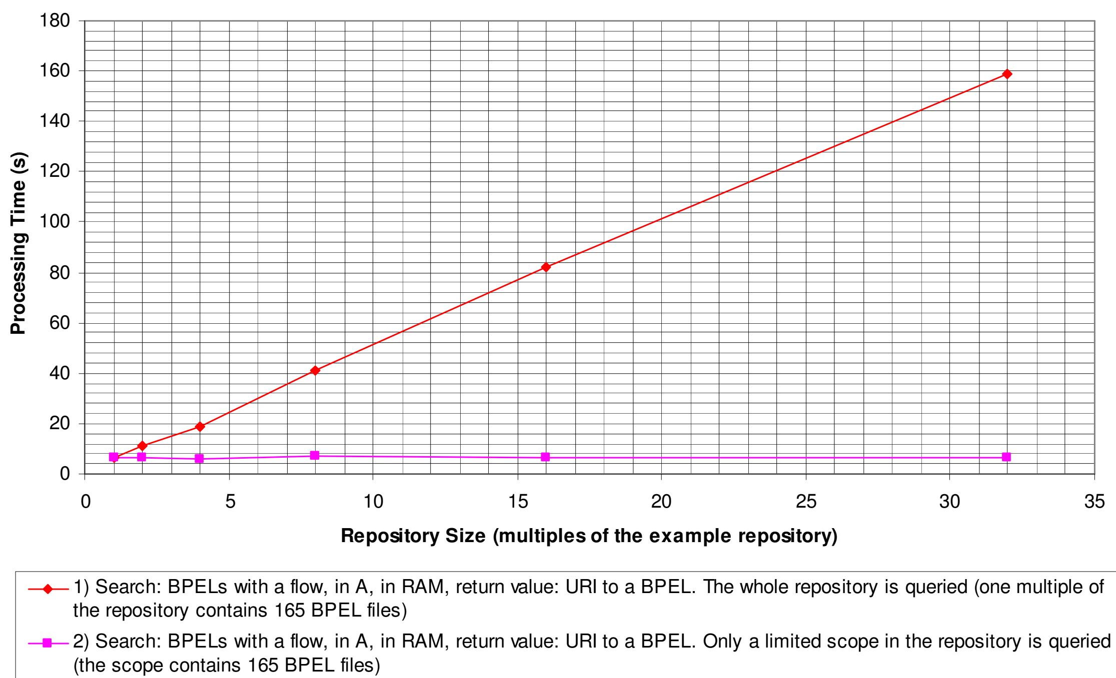

Figure 5.10: Processing time for a query if the scope is fixed for various repository sizes. It is compared to the case where the whole repository is queried.

As the reference test case (1) from the previous chapter increases linearly as a function of the repository size, the test (2) with the fixed scope has almost constant

query processing time. The times fluctuate between 5.95 seconds and 6.93 seconds, however no linear behavior can be found, since these extreme values are results from the medium repository sizes, whereas with the minimum repository size the query time was 6.46 seconds and with the maximum repository size 6.51 seconds. The details can be seen in Figure 5.10 above.

Figure 5.11 presents the main memory consumption as a function of the repository size with a fixed query scope. It is compared to the case where the scope is not fixed, and instead, the whole repository is queried.

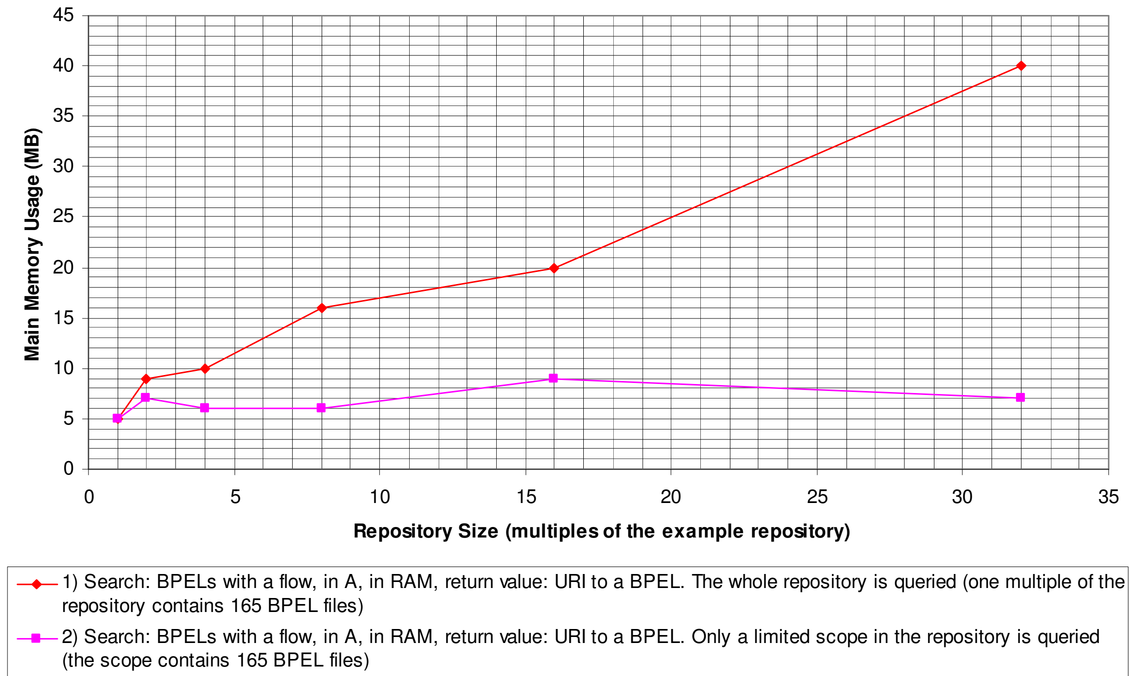

Figure 5.11: Main memory usage of the query with a fixed scope.

The main memory consumption fluctuates relatively more than the processing time consumption. The minimum need is 5 megabytes, whereas the maximum is almost twice as much, 9 megabytes. However, no clear linear trend is visible as the repository size increases.

Analysis

Hypotheses (i) and (ii) are reasonably correct since both the query processing time and the main memory consumption are pretty much constant regardless of the repository size. They do not grow linearly as a function of the repository size. Instead they fluctuate in a narrow range of values. It is likely that a part of these differences can be caused by the variation of the file locations in the file system. A retrieval time from the hard disk can depend on the physical data location. Another part is likely caused by the operating system retrieving the data, or by the scalability behavior of the Java data structures under different repository sizes.

Nonetheless, as a conclusion, using a scope to narrow the queries is a powerful way to improve the querying performance. This approach is useful if the repository size

grows so big that the query performance becomes insufficient for the intended purpose. If specifying the scope for a query can reduce a set of files needed to be queried, it can decrease the query times noticeably. The same mechanism is useful if queries are chained. A result collection of URIs from the previous query can be used as a scope for the following query.

5.3.3 Query Performance Depending on the Result Types

The test explores the influence of the result type on the query performance. For example, when the BPEL files are queried with a query having a Boolean result, a collection of BPEL files is returned that matches the query. Depending on the query parameter (F), either a URI of each BPEL file is returned or additionally the EMF object of each BPEL file is returned.

Alternatively, if a BPEL file is queried depending on the query parameter (E) another file can be returned for each matching BPEL file. For example, in this test scenario the WSDL public interface is returned. Either a URI to each WSDL file can be returned, or additionally the EMF object of each WSDL file is returned.

In addition, it is tested if it makes any difference to the querying performance, whether most of the queried files match the query or not. Instead of using the same query as in the previous test cases, another OCL query is used:

```
context process::TProcess inv: not self.sequence.oclIsUndefined()
```

Whereas the previous query returned 3 positive results among the 165 BPEL files in the content collection A, this query gives 161 positive results. A *sequence* is much more common in the BPEL files as the root element of the activity tree than the previously queried *flow*.

Hypothesis

i) There is no difference in the querying speed depending on the number of files matching the query.

ii) The number of file matches should not affect the main memory usage, since the only difference is that URIs are returned to the positive cases, the URI object is relatively small.

iii) If the queried BPEL file is returned as an EMF object, it will take as much processing time as returning only its URI.

iv) Returning URI to another file increases the processing time only slightly.

v) The loading time will double if another file is returned as an object.

vi) If a returned URI refers instead of the queried file to another file, it does not increase the memory usage.

vii) The memory needed to store the BPEL objects increases proportionally with the repository size and thus, as the number of positive results increases, because they are all kept in the memory after they are queried.

viii) If another file type (WSDL) is returned, the memory consumption is equal to the case, where the queried BPEL file is returned, taking into account that the average size of BPEL files is equal to the average size of WSDL files.

Results

The Figure 5.12 below shows the query processing times with different return values.

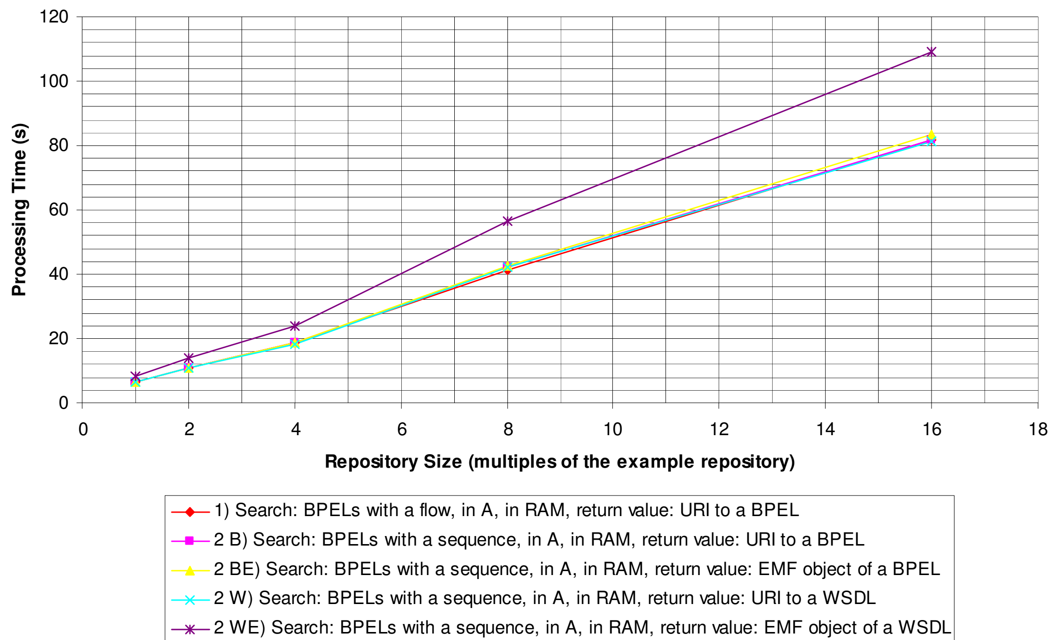

Figure 5.12: Query processing time with different return value options.

There is no noticeable difference depending on whether there are many files matching the query or only a few. Both the query searching for flows (1), and the query searching for sequences (2B) use the same amount of querying time.

There is no significant difference depending on whether a BPEL file URI, or an EMF object of a BPEL file, or a URI to its WSDL public interface file is returned. The only query differing from the others is the one, which returns the EMF object of the WSDL public interface file for a matching BPEL file.

The following Figure 5.13 below illustrates the differences in the memory consumption depending on the selected return type. All there queries (1, 2B, 2W) that return a URI to a file have the same memory usage. When the matching WSDL files are returned as an EMF object (2WE), the memory usage is significantly bigger. But the when the matching BPEL files are returned (2BE), the memory usage is even larger.

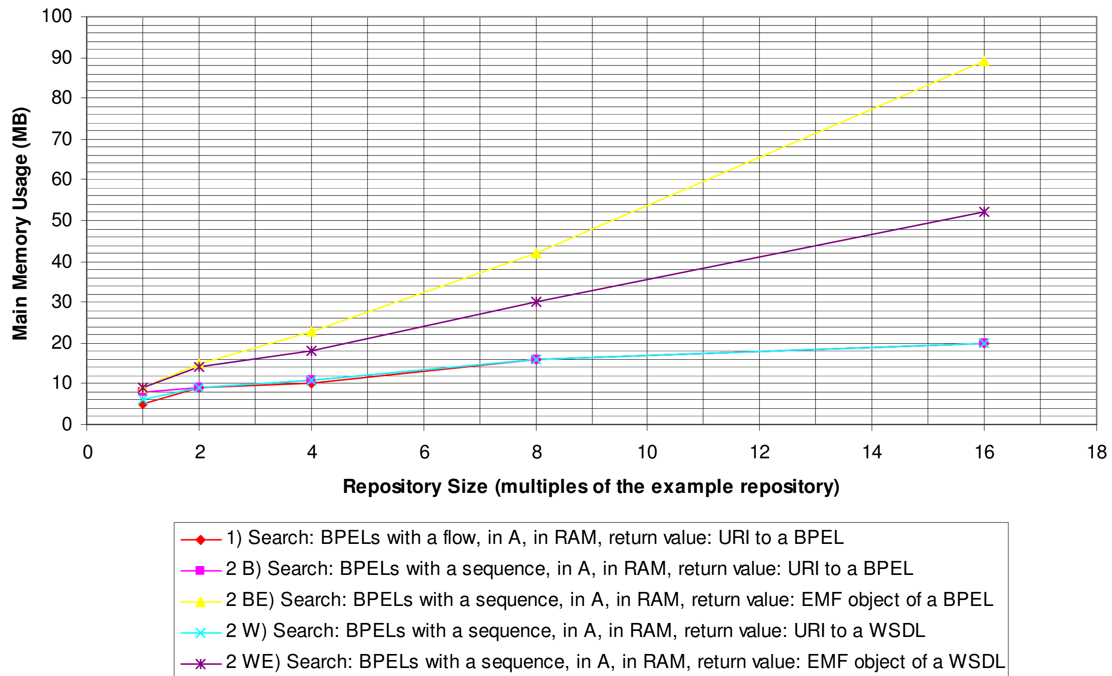

Figure 5.13: Query main memory usage with different return value options.

Analysis

As hypothesis (i) stated, the query processing time is independent of the number of positive matches. The query mechanism loads all the descriptor files and all the BPEL files in the memory for the query. Most of the time is spent in this loading process. A query engine search evaluates the query for all the BPEL EMF objects. The only difference amongst the cases is, whether *true* or *false* is returned from the query engine. The same reasoning explains why the main memory usages are equal as well, as the hypothesis (ii) forecasted.

The same behavior is also visible when the queries with different return values are compared. The processing time remains the same independently of whether a URI to the BPEL, an EMF object of the BPEL or a URI to its WSDL public interface file is returned. Therefore, hypotheses (iii) and (iv) are correct. The two first objects are needed for the query anyway and the URI to the WSDL file can be found fast from the descriptor file that is already in the memory for the query. If a WSDL file is deserialized to an EMF object from a WSDL file, the only difference occurs, which is forecasted in hypothesis (v). The loading operation of another file consumes extra time compared to the other queries.

For the same reason, the memory behavior matches hypotheses (vi), (vii) and (viii). There is no difference in the memory consumption whether a URI refers to a BPEL file or a WSDL file. The number of URIs returned as a result is not significant when compared to all the other memory usage of the repository.

It makes a difference whether an EMF object is returned as a result or only a URI. If the BPEL EMF object is returned, it cannot be unloaded from the memory after the

query has been finished. Instead, it must be kept in memory. If its WSDL public interface EMF object is returned, it must be loaded in memory for all the positive results. However, the queried BPEL EMF object can be unloaded in this case. Returning BPEL EMF objects takes more memory than returning the WSDL EMF objects, because the BPEL objects are larger in the example repository than the WSDL objects.

6 Qualitative Analysis of the Repository Design

In the previous chapter, a quantitative analysis was conducted examining the repository performance and scalability. This chapter evaluates the qualitative side of the repository design and the fulfillment of the requirements.

First, the BPEL repository is compared to other repositories. Then, the section 6.2 focuses on the query mechanism. Finally, the extensibility of the repository is evaluated.

6.1 Comparison to Other Repositories

In chapter 3, the state-of-art of the alternative approaches to a BPEL repository are reviewed. They each provide a way to support storing and querying BPEL and related XML files. However, none of the approaches is currently optimal for this purpose. In the following sections, the BPEL repository is compared to the other applications:

- File systems (e.g. in Windows XP)
- Version control systems (e.g. CVS [Ced04])
- XML databases (e.g. Natix [FHK+02])
- Relational databases (e.g. DB2 [Ibm04])

6.1.1 Object Representation of the Data

The BPEL repository takes advantage of the strong support of the Eclipse Modeling Framework (EMF), which provides powerful support for serialization of the EMF Java objects to XML files [BSM+03], [Ecl04b]. In addition, this mapping between the models is provided automatically. One model can be generated automatically from another, for example an EMF model from an XML Schema, and vise versa.

The goal of the BPEL repository is to provide this strong support for other applications. Other repositories do not provide a mechanism to transform standard XML

Schemas to an object model and a support to serialize or deserialize the instances of the model.

File systems, version control systems, and XML repositories are file containers. Thus they are not capable of building object models from the data they store. Relational databases are built to support applications, but mapping database data to objects can be achieved in multiple ways. No automatic linkages between the models and their serialization are provided.

The Service Data Object (SDO) framework drivers can provide the linkage between an object model and databases [Ecl04b], but these drivers are not yet available. In anticipation for the SDO release, the BPEL repository is built in a way that SDO can be added by changing the file system data handler component to another component that provides an SDO access to data storages.

6.1.2 Support of Object-Oriented Querying

The power of the BPEL repository querying support is also derived from the Eclipse Modeling Framework. The EMF object data presentation is used for the queries, which allows the user to query the object model with the object-oriented query language Object Constraint Language (OCL). Other compared repositories do not provide any object-oriented querying mechanism.

The advantage of an object-oriented approach is that the application developers building their programs on top of the BPEL repository do not need to be aware of the data storage syntax of the objects. Instead, they can concentrate on handling everything using the object model. This is a helpful approach if an application is built in the spirit of the Model Driven Architecture (MDA). Only the object model must be designed and EMF takes care of the data serialization. The BPEL repository provides a querying mechanism for these objects in the BPEL domain.

The disadvantage of file systems and version control systems is that they do not provide a query mechanism, which would take advantage of the structured data presentation of the XML files. Their contents can be queried with a file search, which is an inefficient way of querying and can easily lead to misinterpretations of the meaning of the data. Since a file search does not take into consideration the structure of XML files, a searched string can be out of its anticipated context. For this reason, it cannot be used reliably as a sophisticated query mechanism.

XML repositories support native querying for XML data, since they provide XPath and XQuery query mechanisms. This is a powerful approach to querying XML data, since the languages are tailored to XML. However, the disadvantage is that the knowledge of the XML format for the data is needed when the queries are formulated. This might not be straightforward to derive from the object model.

The power of relational databases is their highly sophisticated and efficient data retrieval mechanisms. However, it is not completely straightforward to handle data of object models or XML files with the relational approach. For this reason, there has been scientific research for object-oriented databases and XML databases. Nonetheless, relational databases are still the most common data storage method for applications handling a large collection of data.

6.1.3 Performance of the Querying Mechanism

The disadvantage of the conventional relational databases is their lack of native support for XML data. The BPEL repository uses many large XML schemas, which would need a separate mapping to a relational database and another mapping to the object model. If these must be done manually, extending the repository with new XML schemas would require a major effort. The same effort is needed if the mappings are built for the initial set of XML schemas. The EMF approach for the BPEL repository decreases the amount of work in this case.

A hindrance of querying an object model is the limited performance. Since models are loaded completely into the main memory, lots of memory and processing time is needed even for a simple query. This approach does not take advantage of the structured format of XML in order to improve the querying speed. The XML structure is only used to find the data in the right place of the structure. Instead, it could also be used for loading only the necessary data for queries, which would drastically improve the querying performance.

XML repositories are planned to provide this kind of querying mechanism by supporting XQuery and XPath queries. Unfortunately, the queries are applied to the XML data, not to the object representation. XML repositories support indexing and other querying speed-ups.

Well-scalable query performance is the power of relational databases. They have been optimized to handle large amount of data. Much scientific research has been dedicated developing efficient data retrieving and querying mechanisms for relational databases. However, the relational databases do not provide support for object-oriented queries and Oracle 10g is the only database with XQuery and XPath querying support. Nonetheless, as a commercial product, the Oracle database is not an ideal basis for an open-source research prototype.

The purpose of the BPEL repository is to provide useful services for other applications, which handle file collections from hundreds to thousands of files. The performance tests show satisfying performance results in this order of magnitude. Relational databases and XML databases could probably handle millions of files efficiently, but it is not needed at the current time when such a large collections of BPEL files do not exists. Instead, the BPEL repository provides a querying capability for object models.

6.1.4 Links amongst the XML Documents

One purpose of the BPEL repository is to provide links amongst the files that relate to each other. The BPEL repository handles links by grouping the related files in a single organization and maintaining the links to the file locations and roles in a XML descriptor file. The descriptor files are accessed while querying the repository in order to locate files with specific roles for queries.

The BPEL repository provides the linkage service in the API. File systems or version control systems cannot provide this functionality, so they had to be extended. In XML repositories, this information could be stored in the tags of the files. In relational databases, the information could be in separate tables. However, none of the other

repositories provide API functions like the BPEL repository. It would be necessary to build the service on top of the other repositories as a separate layer, such as the BPEL repository is built on top of the file system likewise at the moment.

6.1.5 Summary of the Repository Comparison

The differences between the compared data storages are summarized in Table 6.1 below.

Table 6.1: Summary of the repository comparison.

	Data representation	**Query support**	**Links amongst the XML documents**
BPEL Repository	Files, EMF objects	OCL queries and Java extensions, or other query engines for Java objects	Descriptor file for each organization having an object and XML format
File systems	Files	File search only, no structured approach	No native support
Version control systems	Files	File search only, no structured approach	No native support
XML databases	Structured XML data	XQuery, XPath	Tags for files
Relational databases	Relations, tuples in flat tables. Unstructured, shredded or structured XML data	SQL (XQuery only with Oracle 10g)	Keys between the tables

6.2 Querying Capabilities

The strength of the BPEL repository is its ability to execute object queries. In co-operation with the OCL Tool from the University of Kent [AP04], EMF Java objects can be queried using the Object Constraint Language (OCL) [WK03]. OCL is both a query and constraint language. Nowadays it is not yet widely used as a query language most probably because of lack of object-oriented repositories being widely used. It is used more to enrich UML models by defining constraints for them, since it is a powerful method to add formal information to UML models. This information can be used while creating implementations with the Model Driven Architecture. Hopefully, the use as a constraint language would cause it to become more popular as a query language.

Another use for the Object Constraint Language would be in the Query / Views / Transformations (QVT) framework of Meta-Object Facility (MOF) [Omg02]. QVT is a

request for proposals in the Object Management Group (OMG). OCL has been recommended to be the query language of the standard [GGK+03].

OCL is a powerful query language for object models. It supports Boolean queries and object retrieval. It is possible to navigate through associations in an object model, when building up the queries. Thus, it is possible to directly use the object model for queries and the user does not have to be aware of any other models, such as its serialization to XML. It supports the type systems of object-oriented programming languages, for example casting of object types, inheritance, and polymorphism. In addition, OCL provides many high-level set operations. It has hierarchical sets and it uses sets instead of relations used in the SQL queries of relational databases.

The strength of OCL is that it is designed for object queries unlike the competitor query languages XQuery and SQL used in the other repositories. Otherwise, XQuery would be a good query language alternative for the BPEL repository, since it would support querying in the standard XML storage format for BPEL, WSDL and XSD files.

A disadvantage of querying the object model is that the queried objects are loaded completely into the main memory. Furthermore, no indexing of the data is used to speed up the queries. One query speed-up technique for simple queries could be using lazy loading of XML to objects. Then only the parts of XML files, the objects that are used in the query, would be loaded to the object model. The parts would be loaded if they are necessary. The lazy loading of XML files to a Document Object Model (DOM) is explored in the Xaluca prototype [Cha04]. Implementing lazy loading for EMF objects is out of the scope of my thesis.

However, as the BPEL repository uses a general query engine interface, other query engines can be integrated into the BPEL repository. If another query language is preferred, it can replace the OCL Tool by implementing the query interface. For instance, an IBM internal query engine has already been plugged into the BPEL repository.

6.3 Extensibility

A goal of the BPEL repository development is releasing it as an open-source project to enable use with other software prototypes. Furthermore, it can be extended and adapted to their specialized needs. Therefore, the overall architecture is planned for general BPEL domain use.

The BPEL repository is built on top of the universal Eclipse platform. Eclipse is designed as an open extensible integrated development environment (IDE) for anything and nothing in particular [Ecl04a]. Implementing software on top of this platform makes it easier for other developers to extend it and integrate other software. The BPEL repository uses the Eclipse workbench as the basis of its graphical user interface, and EMF to present and manipulate the persisted objects. Use of these frameworks is helpful for the BPEL repository implementation, and meets the extensibility requirements for the solution. The user interface can be easily integrated with other software based on the Eclipse workbench using Standard Widget Toolkit (SWT) and JFace libraries.

The EMF data models can be used and serialized together with other EMF models. A good example of this capability is the co-operation with the OCL Tool from the

University of Kent. The OCL Tool has an EMF specific version, which takes advantage of the EMF model properties. It uses EMF to provide a richer OCL query framework than on a simple Java model. All together this is a good example of how through a common data framework, applications can take advantage of each other. This is a source for the good extensibility of the Eclipse platform.

Since the BPEL repository is built to support the standard XML schemas for BPEL, WSDL and XSD files, it means that any files compliant with those standards can be handled. In addition, the design policy, which allows keeping the extensions of these standards in the repository as well, provides an even better basis for the repository to be extended. It makes it possible to use the BPEL repository even though proprietary extensions of the standards are manifested in files. This is important, because the BPEL specification is still under standardization process and there exits already multiple kinds of extensions of the standard.

Fortunately, it has been possible to have a good approach to build a multi-purpose BPEL repository as an independent project from other related research projects. This implies that the repository is not custom-made for the partner projects. Instead, the architecture is general and applied to the context of the other projects. Being successful, this gives a good example of the extensibility of the BPEL repository.

7 Synthesis of the Repository Work

In chapter 3, the state of the art of existing repository technology is considered insufficient with respect to the requirements of this project. This encouraged us to design a BPEL specific repository.

The goal for the BPEL repository design was to provide strong object-oriented support for other software, which could leverage the services of the repository. Support for handling BPEL, WSDL and XSD files based on the standard schemas is supported. On the other hand, client programs can omit this data presentation completely and focus on using only the EMF object model for these file contents. The Eclipse Modeling Framework takes care of the object persistence in XML.

This approach allowed us to provide an object-oriented query mechanism for the repository data using the Object Constraint Language (OCL). The drawback of this querying approach is the querying mechanism scalability, which was evaluated quantitatively. A deep look into the overall performance has been undertaken. These issues are each discussed in detail in the following sections and they lead us to future work possibilities in this area.

7.1 Object-Oriented Querying Capabilities

The main contribution of this thesis is to support an object-oriented querying of the BPEL repository. Object-oriented queries are not a new concept, since both the Object Constraint Language (OCL) and the utilized OCL query engine existed before the repository. Instead, a standard XML representation format for different file types has been given as a requirement for files such as BPEL, WSDL and XSD. An object-oriented querying mechanism is linked to these standard file types, which are not object-oriented themselves. However, since it is predicted that many Java applications will be built using this data presentation standard, the goal is to provide a service to handle them completely within an object-oriented approach.

The linkage amongst XML and object data formats of the new Eclipse Modeling Framework is leveraged to bind the representations together. Then OCL is utilized to query the EMF objects, which reflect the content of the XML files. This allows the use of the intuitive object-oriented approach to query the BPEL repository. Navigation through associations, strong typing, inheritance, polymorphism and other support mechanisms, which object-oriented programming languages provide, can be utilized for data retrieval and querying.

7.2 General Analysis of the Repository Performance

The drawback of the implemented object-oriented querying mechanism is its modest performance compared to relational databases. Starting up the BPEL repository takes time and consumes main memory. Resources are spent to build up the data structure that is used to make queries more efficient. The start-up time increases linearly as the repository size grows. The number of files and organizations in the BPEL repository are the main factors that influence the repository size.

At start-up time, the file index of the organizations and their descriptor files are loaded into the main memory. Other files are loaded at query time, when required. This improves the query and retrieval times, since the structure of repository is often used independently of the file selection. Of course, the cost of this speed-up is the main memory usage for the repository structure. However, keeping the organization file indexes in the main memory consumes relatively little memory compared to the alternative, where all the files of the repository would be kept in the main memory.

The repository start-up takes several seconds for a repository size, which contains a couple of hundreds of organizations and one thousand files. For bigger repositories, it is recommended to keep an instance of the repository running, since it makes consecutive queries reasonably faster. If the repository is started up separately before each query, the execution time of a simple query roughly doubles.

As with the repository start-up, the repository query resource utilization increases linearly compared to the repository size. The processing time is spent in searching the queried files from the descriptor files and loading the queried files from a file system to build the objects and then querying those objects in a query engine. Similarly, the memory consumption increases with the number of loaded descriptor files in organizations and the number of loaded files for querying increase.

In the test environment, the querying mechanism scaled reasonably well for repositories containing up to 15 000 organizations, a total of 75 000 files and 10 000 queried BPEL files. A query consumed 5 minutes of processing time and the main memory usage of the repository was around 250 MB. Nonetheless, BPEL file collections with this size are rare nowadays, so the BPEL repository is targeted for smaller needs. For example, 1000 BPEL files could be loaded and queried in 30 seconds.

The repository scalability changes after the repository size grows above 15 000 organizations. The execution time for the queries grows faster than linearly as the repository size increases. A reason for this is that the files are not kept in the main memory anymore, but they are loaded again from the hard drive. Also the main memory

used by the repository is swapped to the hard disk. However, in the test environment, the BPEL repository is capable to query even repositories with 300 000 files, 60 000 organizations, and 40 000 queried BPEL files.

An upper-limit of the scalability was not tested, since this repository would not be a candidate for such big content collections. The repository querying mechanism does not take advantage of indexes and other efficient methods existing in relational databases. Thus, querying times are rather linear compared to the repository size than for example logarithmic. The querying mechanism should scale better than linearly in order to be efficient for huge amount of data.

Retrieving a file from the repository is efficient, since the entire repository structure is loaded in a hashtable and a tree structure. The hashtable allows reading, creating, and deleting a file in the repository in constant time. The tree-structure keeps subtree queries efficient, since all the files in an organization and its direct suborganizations are found without a separate search from the repository-wide hashtable. Thus, the retrieval time of a file is efficient, being more dependent on the hard disk operation time than on the repository size.

In addition, mechanisms are developed for speeding up the queries. A query can be targeted to one or more subtrees of the repository organization structure. The performance tests show that with a given fixed query scope, the repository size does not influence the query performance. A descriptor file classifies the file contents in the same organization, so only the needed files are loaded for the queries.

7.3 Suitability to the Intended Purpose

Despite the limited scalability compared to relational databases or XML databases, the BPEL repository is still suitable for the purpose it was constructed. The repository is built to provide query, storage and organizational services for other software prototypes. It saves them handling object serialization to XML files.

The object representation is generated automatically from the XML Schemas with the power that the Eclipse Modeling Framework offers. The BPEL repository can be extended with other XML formats with the same automatic generation of the object model. Another way to bring data models into the repository is to generate an EMF object model from UML diagrams or Java annotations. Thus, the BPEL repository provides good extensibility for new data formats and their object representations. This is important for a multipurpose repository, which is likely to contain new proprietary data formats.

Additionally, this approach makes the development of applications easier, since they do not have to handle the data in any other format than in the object representation. They can query the repository with this same object representation using OCL. Other query engines can be plugged in, for example, for customized purposes.

The repository is made extensible by building the user interface on top of the Eclipse platform and using the multi-purpose Eclipse Modeling Framework as the data object representation. All these factors together make the BPEL repository a useful component for other software prototypes handling BPEL and related XML files.

7.4 Future Work

The main challenge for the future related to the BPEL repository is how to improve the scalability of the object-based querying and retrieval to handle larger collections of data. Maybe then querying objects would make sense even more generally with XML databases. The Eclipse Modeling Framework would be useful serialization mechanisms even then.

Actually, the Service Data Objects (SDO) would be a useful toolkit using EMF with databases. Since it is providing a data mediator service for multiple data storage mechanisms, it would be a useful way to load objects from database systems. This was foreseen, when the repository development started. Because no database driver was available for SDO, it was left out of the scope of this thesis. Nevertheless, the BPEL repository is designed in a way that the data handler component can be replaced with another data handler component supporting SDO.

An advantage of using an XML database storing the XML files is that they do not need to be loaded completely as a file. Instead, only the elements that are of interest to the query could be loaded in objects. This could utilize a lazy loading mechanism of an entire file, such as lazy DOM loading. EMF provides as well mechanism of how an object, such as a BPEL process, can be scattered to multiple files. This way, only the files that have a relevant part for a query could be loaded while querying. The drawback is that this separation must be tailored by hand. This approach would also need a separate import and export mechanisms for the files. The repository should support inserting and retrieving a BPEL file containing a whole BPEL process. Applying the EMF framework would be more time consuming to develop.

Since the main source of query inefficiency is that the whole file is loaded for a query, the problem could be handled also with another approach. Instead of sending the whole object to the query engine, the queries could be executed directly in a database. The query would be sent to the data storage, instead of loading the data in the query engine. This would allow taking advantage of the efficient querying mechanisms that a database provides, such as indexing the often-queried data. However, neither relational databases nor XML databases provide an object-oriented query mechanism to query their contents. This could be achieved by mapping OCL queries to native database queries, such as SQL for relational databases and XQuery for XML databases.

In fact, there is some on-going research in an early stage to map XML/XQuery to UML/OCL [SG04a], [SG04b]. The Eclipse Modeling Framework could be utilized if the OCL queries to EMF objects are mapped to XQuery querying XML documents. A possible framework for the future work is presented in Figure 7.1.

An EMF model can be created automatically from an XML Schema. The linkage of the models provides a way to serialize EMF objects to XML documents. OCL navigates in an object hierarchy using the EMF model while the instances of the model are queried. Similarly, XQuery navigates in the structure of the XML Schema while querying the compliant XML documents. An OCL transformation to XQuery could be designed by taking advantage of the existing transformation technique between EMF models and XML Schemas.

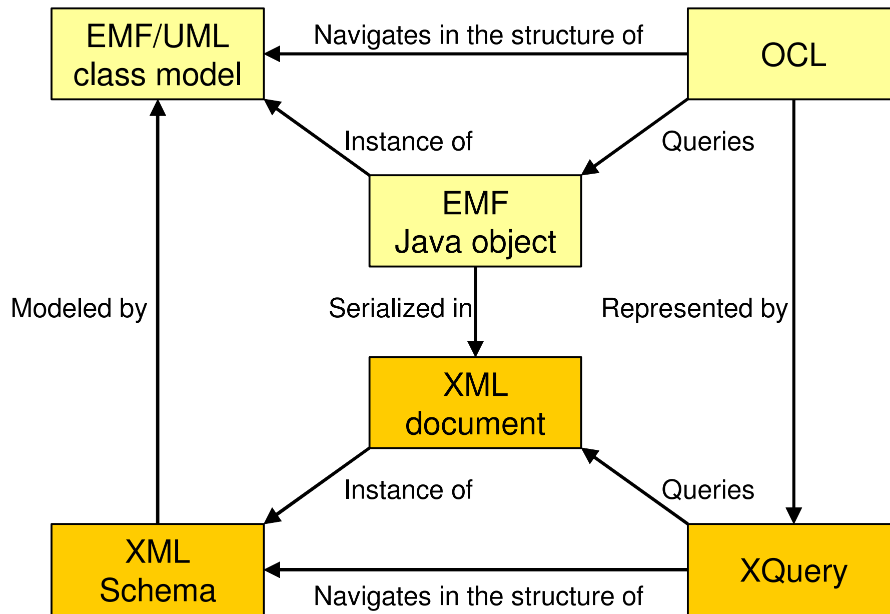

Figure 7.1: The framework to map OCL to XQuery using Eclipse Modeling Framework.

8 Conclusions

There is a need in IBM research projects for a BPEL repository that can handle BPEL files and other related XML files. There are no BPEL repositories available that offer querying mechanism to the XML file contents and handle the linkages to related files. However, there are multiple research projects, which are interested in the possibilities that the BPEL standard allows. Therefore, one goal of the project is to serve other BPEL research projects by offering them an open-source BPEL repository.

In addition, the suitability of the Object Constraint Language (OCL) as an object-oriented query language is explored in the project. The Eclipse Modeling Framework provides a new technology for object serialization to XML files. The work tests how well it serves its purpose in the BPEL domain.

The main requirement for a BPEL repository is to search objects persisted in XML files by querying their contents and contents of other linked XML files. For example, a BPEL file could be retrieved based on its XML metadata file contents. Thus, support for linking files is needed. The repository needs to be extensible with new file formats, which can be based on some new XML schemas. Since the purpose is to serve multiple research projects with a generic approach, the repository is needed to be extensible and support several ways to use it. In order to make other development work easier, a goal is to offer a Java API for other software that they can use and not force them to build this functionality for each product separately.

The designed BPEL repository relies on a completely object-oriented approach. It uses the power of the Eclipse Modeling Framework, to provide objects to be handled for the other applications by taking complete care of their serialization to XML files. EMF is capable of creating EMF object models from XML Schemas and UML class diagrams. It also takes care of keeping the serialized data compliant with their XML Schemas. This is important as BPEL, WSDL, and XSD files all have their standard schemas, which must be supported to enable file exchange.

The BPEL repository architecture is divided into components, which have their own responsibilities and can be replaced by alternative components. The repository logic

component takes care of executing a query over the repository contents. It is served by the data handler component, which provides object access to a data storage, and the query engine component, which adapts a query engine to the repository. These components form the repository API for other applications. On top of it, a graphical user interface is built integrated with the Eclipse workbench.

The query mechanism iterates through a query scope by loading all the queried files and sending them to a query engine. The result of each sub-query is stored and they are returned all together when the query is finished. The query mechanism scales linearly compared to the repository size. It does not utilize query speed-ups that are common in database systems, such as indexing of the stored data. Instead, all the relevant files for a query are loaded, when the query is executed.

The power of this querying mechanism is an intuitive object-oriented approach using the Object Constraint Language (OCL). Navigation through associations, strong typing, inheritance, polymorphism, and other support mechanisms that object-oriented programming languages provide, can be utilized with data retrieval and queries.

The querying mechanism is still efficient enough for the purposes it is created. BPEL file collections are not so large that the limited performance of the repository would be an issue. It provides a useful and completely object-oriented API for applications that can build features on top of it. Applications can concentrate on using the EMF object model that the BPEL repository offers. They do not have to pay attention to how the data persistence is handled, even though the files are compliant with their standard XML schemas.

Future work could explore how only those parts of the files that are necessary for the querying mechanism are loaded into objects using a lazy XML loading mechanism. Another option is to map the object-oriented queries to a query language provided by a database system.

References

[ACD+03] Andrews T., Curbera F., Dholakia H., Goland Y., Klein J., Leymann F., Liu K., Roller D., Smith D., Thatte S., Trickovic I., Weerawarana S., *Business Process Execution Language for Web Services, Version 1.1*, (05/05/2003), http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, OASIS Org.

[AP04] Akehurst D., Patrascoiu O., *Object Constraint Language Library*, http://www.cs.kent.ac.uk/projects/ocl/index.html, (referenced: 01/07/2004), OCL project, University of Kent

[BBC+04a] Bajaj S., Box D., Chappell F., Curbera F., Daniels G., Hallam-Baker P., Hondo M., Kaler C., Langworthy D., Malhotra A., Nadalin A., Nagaratnam N., Nottingham M., Prafullchandra H., von Riegen C., Schlimmer J., Sharp C., Shewchuk J., *Web Services Policy Framework (WS-Policy)*, (09/2004), ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf, BEA Systems, IBM, Microsoft, SAP, Sonic Software, VeriSign

[BBC+04b] Berglund A., Boag S., Chamberlin D., Fernández M., Kay M., Robie J., Siméon J., *XML Path Language (XPath) 2.0*, (23/07/2004), http://www.w3.org/TR/2004/WD-xpath20-20040723, W3C Working Draft

[BCF+04] Boag S., Chamberlin D., Fernández M., Florescu D., Robie J., Siméon J., *XQuery 1.0 : An XML Query Language*, (23/07/2004), http://www.w3.org/TR/2004/WD-xquery-20040723/, W3C Working Draft

[BCH+03] Box D., Curbera F., Hondo M., Kale C., Langworthy D., Nadalin A., Nagaratnam N., Nottingham M., von Riegen C., Shewchuk J., *Specification Web Services Policy Framework (WSPolicy)*, (28/05/2003), http://www-106.ibm.com/developerworks/library/ws-polfram/

[BFI+98] Berners-Lee T., Fielding R., Irvine U., Masinter L., *Uniform Resource Identifiers (URI): Generic Syntax*, (1998), http://www.ietf.org/rfc/rfc2396.txt, RFC 2396, Internet Engineering Task Force (IETF)

[BPS+04] Bray T., Paoli J., Sperberg-McQueen C., Maler E., Yergeau F., *Extensible Markup Language (XML) 1.0 (Third Edition)*, (04/02/2004), http://www.w3.org/TR/2004/REC-xml-20040204, W3C Recommendation

[BSM+03] Budinsky F., Steinberg D., Merks Ed., Ellersick R., Grose T., *Eclipse Modeling Framework*, (2003), Pearson Education, Inc., MA, USA

[CCM+01] Christensen E., Curbera F., Meredith G., Weerawarana S., *Web Services Descriptor Language (WSDL) 1.1*, (15/05/2001), http://www.w3.org/TR/2001/NOTE-wsdl-20010315, W3C Note

[Ced04] Cederqvist P. et al, *Version Management with CVS; for CVS 1.12.9*, https://ccvs.cvshome.org/files/documents/19/207/cederqvist-1.12.9.pdf, (referenced: 26/07/2004), Concurrent Versions System

[Cha04] Charuel J., *Optimized XML Parsing with Lazy DOM*, (2004), Master's Thesis, Institute Eurecom – École Polytechnique Fédérale de Lausanne – IBM Research GmbH

[CR04] Clayberg E., Rubel D., *Eclipse: Building Commercial-Quality Plug-ins*, (2004), Pearson Education, Inc., MA, USA

[Ecl04a] Eclipse Org., *Eclipse Org*, www.eclipse.org, (referenced: 01/07/2004)

[Ecl04b] Eclipse Org., *Eclipse Modeling Framework*, www.eclipse.org/emf/, (referenced: 01/07/2004)

[Ecl04c] Eclipse Org., *Hyades Automated Software Quality Evaluation framework*, www.eclipse.org/hyades/, (referenced: 01/07/2004)

[EH04] Edelkamp S., Hoffmann J., *PDDL2.2: The language for the Classical Part of the 4th International Planning Competition*, (2004), Technical Report No. 195, International Planning Competition (IPC-4) hosted at the International Conference on Automated Planning and Scheduling (ICAPS)

[FHK+02] Fiebig T., Helmer S,. Kanne C.-C., Mildenberger J., Moerkotte G,. Schiele R., Westmann T., *Anatomy of a Native XML Base Management System*, (2002), University of Mannheim

[GB03] Gamma E., Beck K., *Contributing to Eclipse: principles, patterns, and plug-ins*, (2003), Pearson Education, Inc., MA, USA

[GGK+03] Gardner T., Griffin C., Koehler J., Hauser R., *A Review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the Final Standard*, (2003), MetaModelling for MDA Workshop, York, England

[GHJ+95] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software,* (1995), Addison Wesley Longman Inc., USA

[HHW04] Le Hors A., Le Hégaret P., Wood L., Nicol G., Robie J., Champion M., Byrne S., *Document Object Model (DOM) Level 3 Core Specification, Version 1.0*, (07/04/2004), W3C Recommendation

[IB04] IBM Corp., BEA Systems, Inc., *Service Data Objects*, ftp://www6.software.ibm.com/software/developer/library/j-commonj-sdowmt/Commonj-SDO-Specification-v1.0.doc, (referenced: 01/07/2004)

[Ibm04] IBM Corp., *DB2 Universal Database for Linux, UNIX and Windows*, http://www-306.ibm.com/software/data/db2/udb/, (referenced: 26/07/2004)

[KB04] Keller A., Badonnel R., *Automating the Provisioning of Application Services with the BPEL4WS Workflow Language*, (2004), in Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2004), IBM T.J. Watson Research Center

[KHW+04] Keller A., Hellerstein J., Wolf K.-L., Krishnan V., *The CHAMPS System: Change Management with Planning and Scheduling*, (2004), in Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS 2004), Seoul, Korea, IBM T.J. Watson Research Center

[Lar02] Larman C., *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd edition, (2002), Prentice-Hall Inc., NJ, USA

[Ley01] Leymann F., *Web Services Flow Language (WSFL)*, (2001), http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, IBM Corp.

[McC04] McCown S., *Databases Flex Their XML*, (26/04/2004), Infoworld.com

[MSS+04] Mendling J., Strembeck M., Stermsek G., Neumann G., *An Approach to Extract RBAC Models from BPEL4WS Processes*, (2004), in Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WET ICE 2004), Modena, Italy

[Nov04] Novatnack J., *A BPEL file collection implemented for the Business Process Integration and Automation project*, (2004), IBM Zurich Research Laboratory

[Omg02] OMG, Inc., *MOF 2.0 Query / Views / Transformations RFP*, (2002), Request for Proposal, http://www.omg.org/docs/ad/02-04-10.pdf, (referenced: 19/08/2004)

[Omg03a] OMG, Inc., *OMG Unified Modeling Language Specification, version 1.5*, (01/03/2003), http://www.omg.org/docs/formal/03-03-01.pdf

[Omg03b] OMG, Inc., *XML Metadata Interchange (XMI) Specification, version 2.0*, (02/05/2003), http://www.omg.org/cgi-bin/apps/doc?formal/03-05-02.pdf

[Omg03c] OMG, Inc., *Meta Object Facility (MOF) 2.0 Core Specification*, (04/10/2003), Final Adopted Specification, http://www.omg.org/docs/ptc/03-10-04.pdf, (referenced: 19/08/2004)

[Ora04] Oracle Corp., *Collaxa: Model, deploy and manage BPEL business processes*, http://www.collaxa.com/home.index.jsp, (referenced: 01/07/2004)

[SG04a] Sakr S., Gaafar A., *Towards Complete Mapping between UML/OCL and XML/XQuery*, (2004), in Proceedings of the IADIS e-Society 2004 conference (ES2004), Avila, Spain

[SG04b] Sakr S., Gaafar A., *Towards a Framework for Mapping between UML/OCL and XML/XQuery*, (2004), in Proceedings of the 7th conference in the UML series (UML2004), Lisbon, Portugal

[TBM+01] Thompson H., Beech D., Maloney M., Mendelsohn N., *XML Schema Part 1: Structures*, (02/05/2001), http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, W3C Recommendation

[Tha01] Thatte S., *XLANG Web Services for Business Process Design*, (2001), http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, (referenced: 25/8/2004), Microsoft Corp.

[W3c04] W3C DOM Interest Group, *Document Object Model (DOM)*, (2004), http://www.w3.org/DOM/, (referenced: 26/07/2004)

[WK03] Warmer J., Kleppe A., *The Object Constraint Language, Getting Your Models Ready for MDA*, 2nd Edition, (2003), Pearson Education, Inc., MA, USA

Appendix I Dictionary of the Repository Terms

The BPEL repository specific concepts and their terms are explained below in an alphabetical order:

BPEL file A file that contains a business process expressed with the language specified in the BPEL specification. Thus, the file is a valid XML file complying with the BPEL XML Schema.

content type A string that specifies the type of contents of a file. It can be freely selected. Together with the content type information, it is used to determine which EMF model is used in the serialization and deserialization processes of the file. They are used to map which XML Schema matches to an EMF model.

file A file in a file system or the part of another data storage contents, which would be stored as a single file in a file system. For example, a database may store a file inside one data type or it can be divided and scattered for example in different tables in a database.

file type A type for a file stored in the BPEL repository. It has a value from a limited set of options: *bpel*, *wsdl*, *xsd*, *descriptor* or *metadata*. Together with the content type information, it is used to determine which EMF model is used in the serialization and deserialization processes of the file.

folder A folder is a structure in a file system that matches to an organization inside the BPEL repository. If the data storage is a file system, the folder term can be used as an equivalent to the organization term.

metadata file A file that contains some XML metadata for the BPEL file, which is stored in the same organization, or other kind of XML file that contains some data related to the organizations or the files in the same organization where it is stored.

organization An organization is a collection of files inside the repository. An organization may contain a single BPEL file and other files that are related to it. If the data storage mechanism is a file system, an organization matches to a folder in the file system.

scope (for a query) A repository query can be narrowed to a subset of the repository contents by specifying a scope for the query. A scope includes URIs of the repository organizations that belong to the scope. The scope can contain only the root organizations of the organizations subtrees that belong into the scope.

WSDL file A file that contains a public interface for a web service or a BPEL file expressed with the language specified in the WSDL specification. Thus, the file is a valid XML file complying with

the WSDL XML Schema.

XSD file A file that contains an XML Schema (XSD) expressed with the language specified in the XSD specification. Thus, the file is a valid XML file complying with the XML Schema. An XSD file may contain a message type for a BPEL process.

Appendix II Example WSDL File

A Hello World WSDL file example is presented below. It also contains an XML schema (XSD) part.

The WSDL file consists of five parts. In the first part, the used namespaces are referred in the attributes of the *definitions* element. The second part is an XML Schema, which defines XML element types for the message types of the BPEL process. The third part message types based on the types defined previously in the schema part.

The fourth part defines port types for the Web service using the message types. The fifth part defines the roles for the Web services using this interface.

```
<?xml version="1.0"?>
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     NAMESPACE DECLARATIONS (Part 1)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<definitions name="SyncHelloWorld"
   targetNamespace="http://zurich.ibm.com/bpia/bpel"
   xmlns:tns="http://zurich.ibm.com/bpia/bpel"
   xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     TYPE DEFINITION (Part 2)
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <types>
    <schema attributeFormDefault="qualified"
       elementFormDefault="qualified"
       targetNamespace="http://zurich.ibm.com/bpia/bpel"
       xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="SyncHelloWorldRequest">
        <complexType>
          <sequence>
            <element name="name" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="SyncHelloWorldResponse">
        <complexType>
          <sequence>
            <element name="helloString" type="string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     MESSAGE TYPE DEFINITION (Part 3)
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <message name="SyncHelloWorldRequestMessage">
    <part name="payload" element="tns:SyncHelloWorldRequest"/>
  </message>
  <message name="SyncHelloWorldResponseMessage">
    <part name="payload" element="tns:SyncHelloWorldResponse"/>
  </message>
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
     PORT TYPE DEFINITION (Part 4)
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <portType name="SyncHelloWorld">
    <operation name="process">
      <input  message="tns:SyncHelloWorldRequestMessage"/>
      <output message="tns:SyncHelloWorldResponseMessage"/>
    </operation>
  </portType>
  <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     PARTNER LINK TYPE DEFINITION (Part 5)
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
  <plnk:partnerLinkType name="SyncHelloWorld">
    <plnk:role name="SyncHelloWorldProvider">
      <plnk:portType name="tns:SyncHelloWorld"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Appendix III UML Diagrams of the Repository API

This appendix contains the UML class diagrams for the main components of the BPEL repository. The structure of the implementation and some architectural choices can be seen in detail from them.

First the package tree is shown in Figure 8.1. It is followed by the main interfaces that the repository API components provide. The Query class, which contains the query parameters is represented in Figure 8.5. Finally, the main classes of the repository API plug-in are illustrated with their associations and attributes in Figure 8.6. The method signatures are suppressed, since if all of them had been visible, the diagram would have been too complex to be shown as a single figure to provide a good overview to the main classes. The detailed UML models will be released together with the open-source Java code.

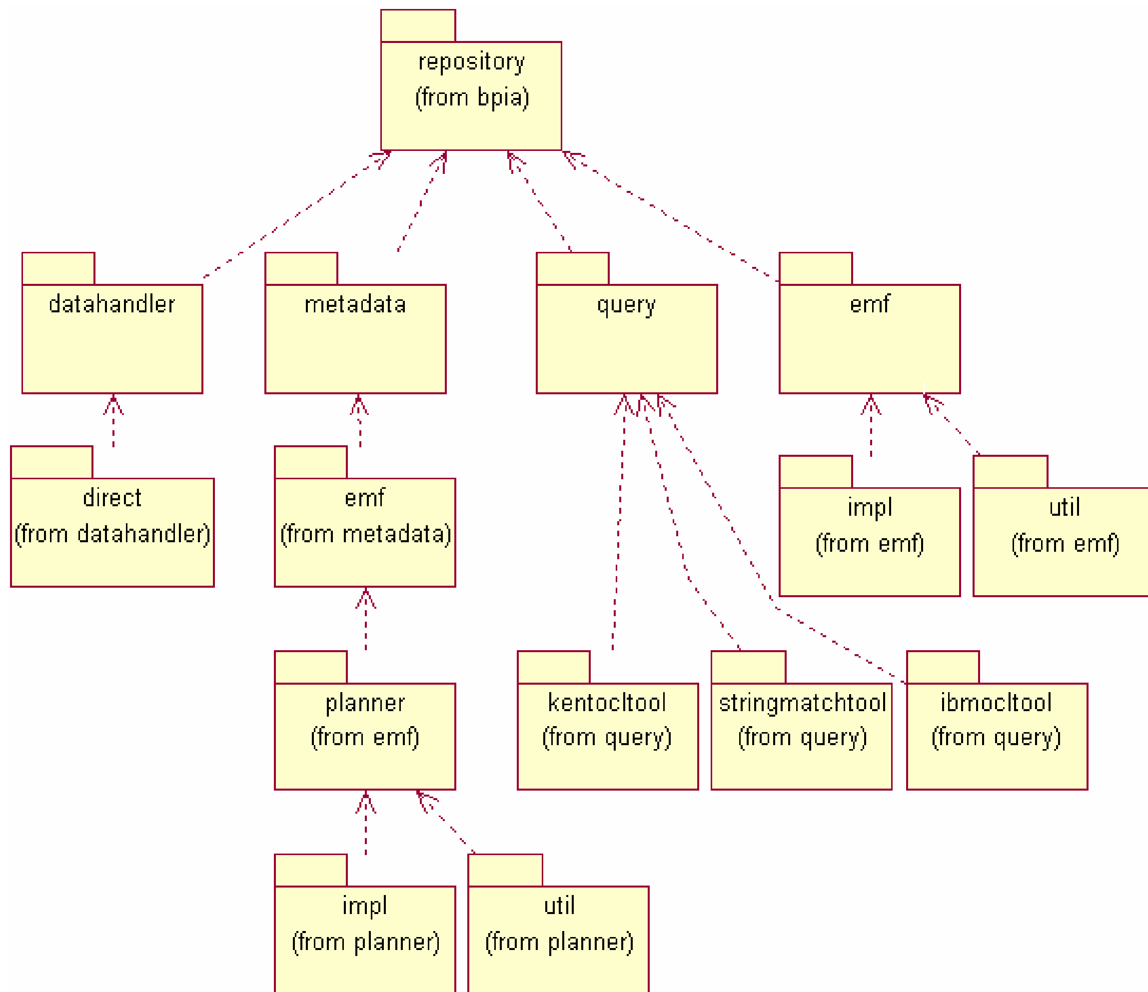

Figure 8.1: Package hierarchy of the repository API plug-in.

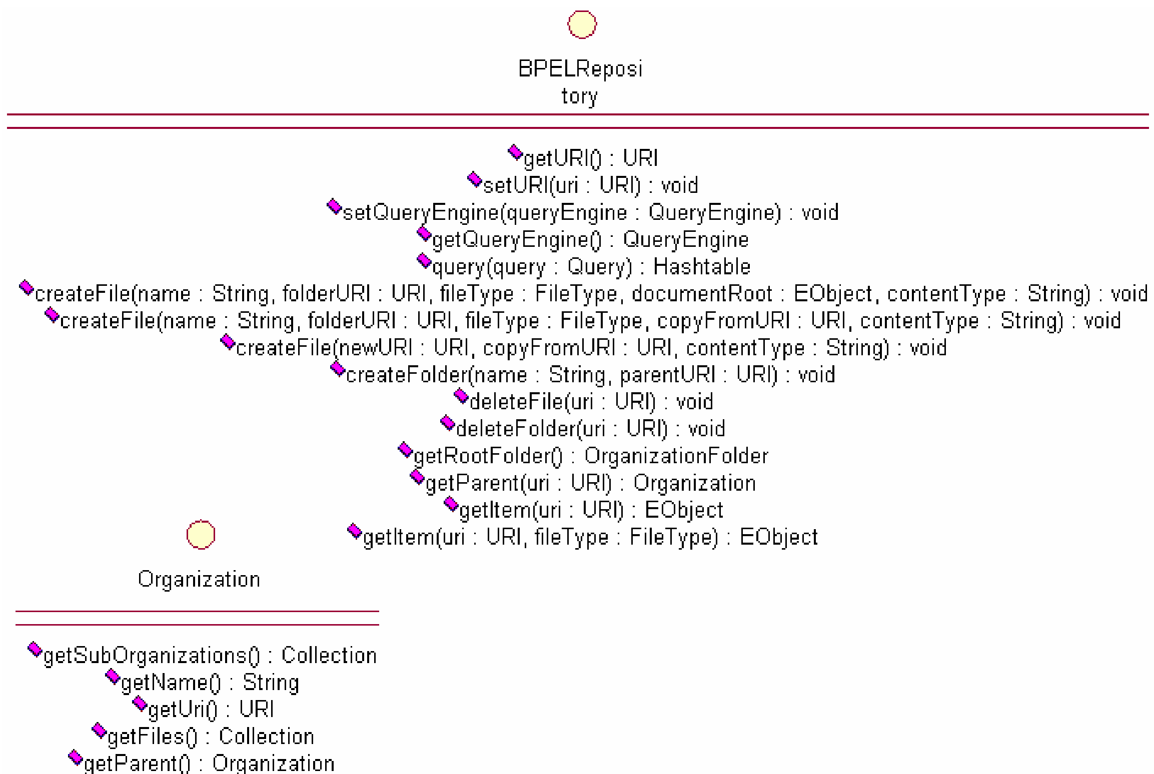

Figure 8.2: The repository API plug-in interfaces provided by the repository logics component.

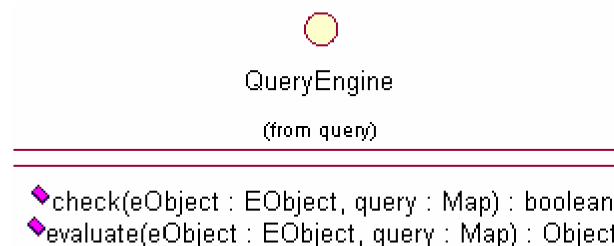

Figure 8.3: The query interface for external query engines.

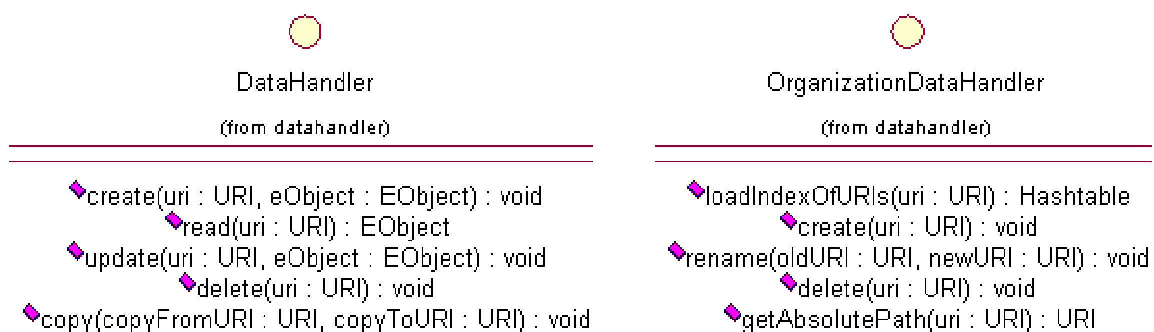

Figure 8.4: The interfaces provided by the data handler component.

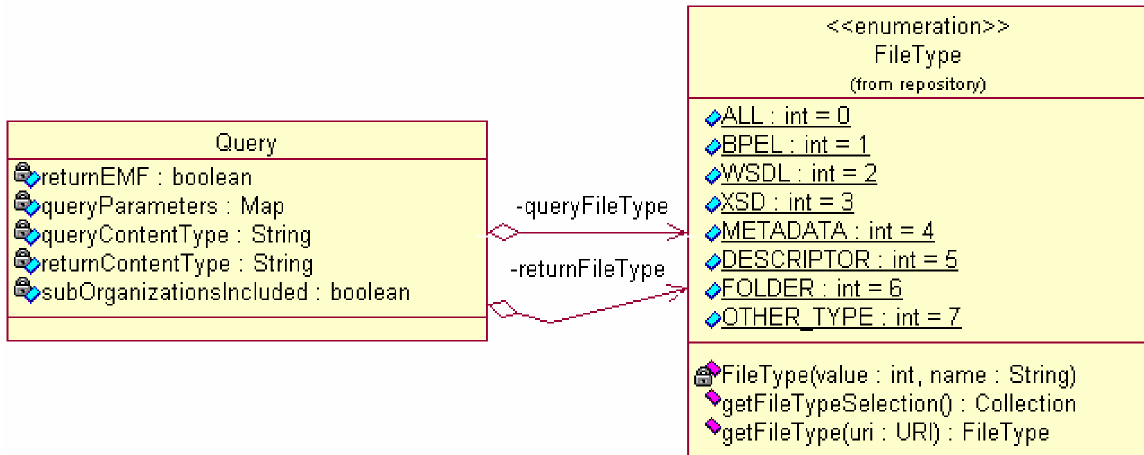

Figure 8.5: Query parameters are defined by passing a Query object to the repository API.

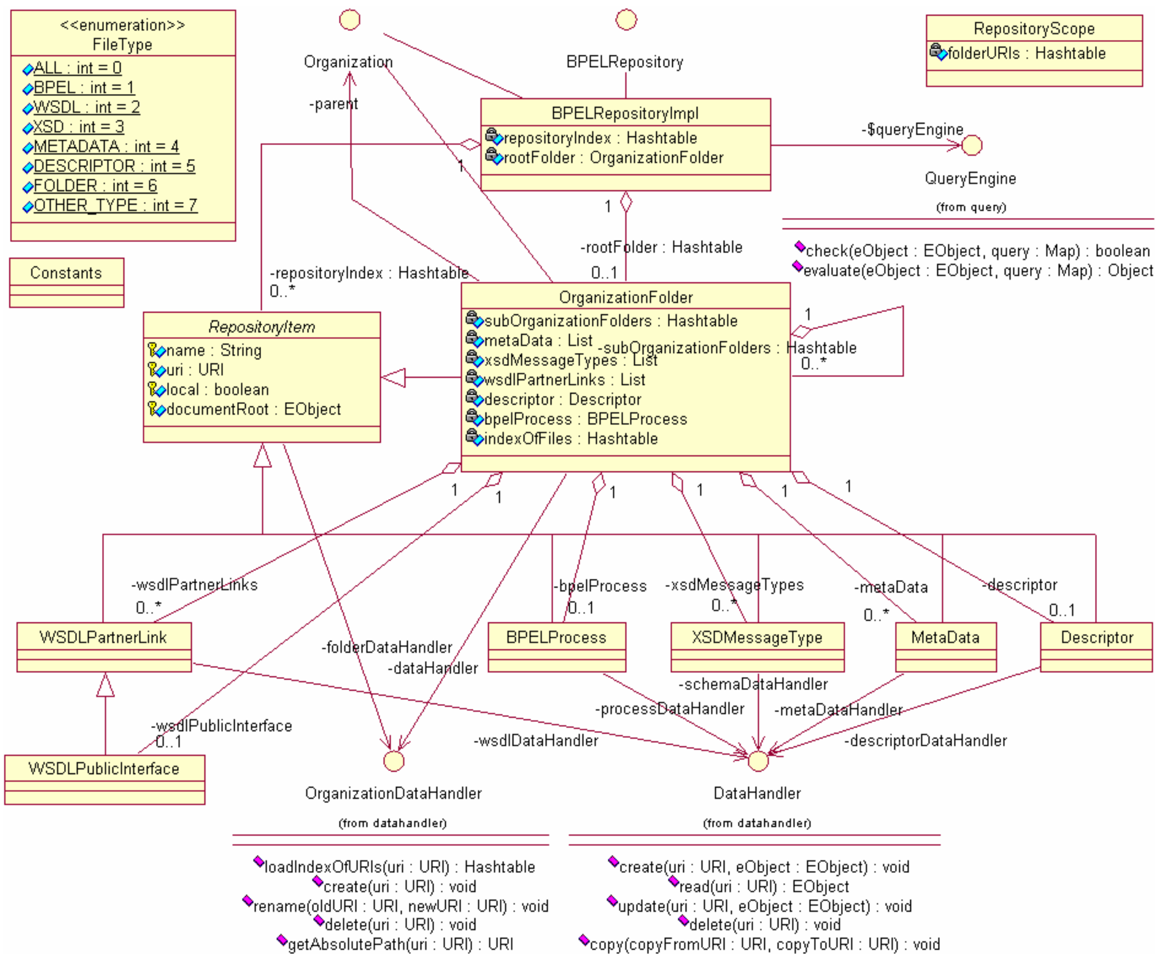

Figure 8.6: The repository API plug-in UML class diagram.