Mining Software Repositories to Support Software Evolution

Shafique Ahmed

A Thesis

In

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science (Software Engineering) at Concordia University Montreal, Quebec, Canada.

March, 2009

© Shafique Ahmed, 2009
NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.
Abstract

Mining Software Repositories to Support Software Evolution

Shafique Ahmed

Software evolution represents a major phase in the development life cycle of software systems. In recent years, software evolution has been recognized as one of the most important and challenging areas in the field of software engineering. Studies even show that 65-80% of the system lifetime will be spent on maintenance and evolution activities. Software repositories, such as versioning and bug tracking systems are essential parts of various software maintenance activities. Given the often large amounts of information stored in these repositories, researchers have proposed to mine and analyze these large knowledge bases in order to study and support various aspects of the evolution of a software system. In this thesis, we introduce a common ontological representation to support the mining and analysis of software repositories. In addition to this common representation, we introduce the SVN-Ontologizer and Bugzilla-Ontologizer tools that provide automation for both data extraction from remote repositories and ontology populations. A case study is presented to illustrate the applicability of the present approach in supporting software maintainers during the analysis and mining of these software repositories.
Acknowledgements

I am most grateful to my supervisor, Dr. Juergen Rilling, for his encouragement, support, and patience throughout this research. Without his help, advice and positive attitude, my studies in this area would not have been possible. I have collected many cherished moments and unique experiences from my supervisor and colleagues at CONCEPT (Comprehension Of Net-CEntered Programs and Techniques) Concordia University.

I express my deepest gratitude to my beloved wife, Rozina and my children, Haris, Ammar and Omar, for their enormous support, infinite patience, and unwavering belief towards me, as always. There is a litany of family members and friends who are not individually mentioned here, but they certainly made a difference.

I would also like to extend my appreciation and thanks to Mr. Philipp Schugerl and Ali Asghar Sheikh for their helpful comments and suggestions, they always helped me in times of great needs and deeds.
# Table of Contents

List of Figures .............................................................................................. viii
List of Tables ................................................................................................. x

1. Introduction ............................................................................................. 1

2. Background ............................................................................................. 3
   2.1. Software Evolution ........................................................................... 3
   2.2. Software Repositories ....................................................................... 9
       2.2.1. Subversion (SVN) ................................................................. 10
       2.2.2. Bugzilla ............................................................................... 15
   2.3. Ontologies ....................................................................................... 18
       2.3.1. Why Ontology? ..................................................................... 18
       2.3.2. Applications ......................................................................... 22
       2.3.3. Web Ontology Language OWL and SPARQL ......................... 24
       2.3.4. Ontology editing tools .............................................................. 26
       2.3.5. Ontological Reasoners ............................................................. 27
       2.3.6. Ontology Alignment ................................................................. 30

3. Contribution ............................................................................................ 32
   3.1. Motivation ...................................................................................... 32
   3.2. Specific Contributions (Sub-goals) and Acceptance Criteria ................. 34
   3.3. Acceptance Criteria ........................................................................ 38
4. Subversion Ontology ................................................................. 39
  4.1. Subversion Ontology Design ................................................... 39
  4.1.1. SVN Repository Schema .................................................... 39
  4.1.2. Initial Mapping SVN Repository Schema to an Ontological Model ........ 42
  4.1.3. Enhanced SVN Ontology ..................................................... 45
  4.2. SVN-Ontologizer ................................................................. 49
  4.2.1. SVN Profile Setup ........................................................... 49
  4.2.2. SVN Connection and Data Extraction .................................... 51
  4.2.3. Data Pre-processing ........................................................ 53
  4.2.4. Ontology population ......................................................... 54
5. Bugzilla Ontology ................................................................. 55
  5.1. Bugzilla Ontology Design ..................................................... 55
  5.1.1. Bugzilla Repository Schema ............................................... 55
  5.1.2. Mapping Bugzilla Repository Schema to an Ontological Model .................................................. 60
  5.1.3. Refining the Bugzilla Ontology ........................................... 63
  5.2. Bugzilla-Ontologizer ........................................................... 69
  5.2.1. Connection to Bugzilla and Data Extraction .......................... 69
  5.2.2. Data Pre-processing ........................................................ 71
  5.2.3. Bugzilla-Ontologizer User Interface .................................... 72
6. Initial Experimental Evaluation and Ontological Queries ....................... 77
List of Figures

Figure 2-1: Cost of the software life cycle [DAC01] ......................................................... 4
Figure 2-2: Common elements of software comprehension models [MPO03] ................... 6
Figure 2-3: SVN architecture [SV09A] ........................................................................ 11
Figure 2-4: An example of directory structure in SVN repository .................................. 13
Figure 2-5: Life cycle of a bug [BU03A] ........................................................................ 16
Figure 2-6: An example of ontology [MAR09] ............................................................... 23
Figure 2-7: Architecture of a Pellet reasoner [PL09A] .................................................... 29

Figure 4-1: SVN relational schema ............................................................................... 40
Figure 4-2: An initial SVN ontology ............................................................................... 43
Figure 4-3: Overview of an enhanced SVN ontology .................................................... 46
Figure 4-4: Reasoning services applied to initial SVN ontology model ....................... 47
Figure 4-5: Reasoning services applied to enriched and enhanced SVN ontology ..... 48
Figure 4-6: SVN-Ontologizer tool overview .................................................................. 49
Figure 4-7: SVN-Ontologizer main user interface ......................................................... 50
Figure 4-8: SVN connection process ............................................................................ 51

Figure 5-1: Bugzilla repository schema ....................................................................... 57
Figure 5-2: An example of issue dependency ............................................................... 59
Figure 5-3: Initial Bugzilla ontology model ................................................................. 61
Figure 5-4: Reasoning services applied to initial Bugzilla ontology ......................... 66
Figure 5-5: Reasoning services applied to enhanced Bugzilla ontology ................. 66
Figure 6- 1: Results of SPARQL query ................................................................. 79
Figure 6- 2: Results obtained from extended query ............................................. 80
Figure 6- 3: Releases and their creation dates ....................................................... 81
Figure 6- 4: Query results based on a revision and committed files ................. 82
Figure 6- 5: Query results showing the contribution of a specified person ......... 83
Figure 6- 6: Reasoning example .................................................................... 84
Figure 6- 7: Knowledge inference based on property DependsOn .................. 84
Figure 6- 8: Example of inferred transitive closure ............................................ 85
Figure 6- 9: Results derived from issue dependency .......................................... 86
Figure 6- 10: Linked Bugzilla and SVN ontologies ............................................ 88
Figure 6- 11: Date/time between issue reporting and resolution ....................... 89
Figure 6- 12: Bug resolution history in SVN ontology ....................................... 90
Figure 6- 13: SPARQL query and results for resolution history ....................... 91
Figure 6- 14: Transitive relationships ............................................................... 91
Figure 6- 15: Query results showing inferred knowledge .................................. 92
Figure 6- 16: Overview of SE-Advisor framework ........................................... 94
List of Tables

Table 2-1: Comparison of some version control systems [VEB09] ........................................ 14

Table 2-2: Comparison of some bug tracking systems [WKB09] ........................................ 17

Table 2-3: Ontologies vs. databases [IAN07] ........................................................................ 20

Table 2-4: Ontologies vs. databases [MUG04] ....................................................................... 20

Table 2-5: Comparison of available reasoners [REA09] ....................................................... 30

Table 4-1: Main classes of initial SVN ontology ................................................................. 43

Table 4-2: Data type properties in SVN ontology ............................................................... 44

Table 4-3: Object properties in initial SVN ontology design ............................................. 44

Table 4-4: Object properties added to enrich SVN ontology ............................................ 45

Table 4-5: DL restrictions applied to classes ....................................................................... 46

Table 4-6: Substitution of invalid characters ....................................................................... 54

Table 5-1: Bugzilla ontology classes .................................................................................... 61

Table 5-2: Data type properties in the Bugzilla ontology .................................................. 62

Table 5-3: Object properties of Bugzilla ontology .............................................................. 63

Table 5-4: Bugzilla ontology classes .................................................................................... 64

Table 5-5: Object properties of Bugzilla ontology .............................................................. 65

Table 5-6: Substitution of invalid characters ....................................................................... 72
1. Introduction

Software evolution represents a major phase of activities involved in the development, use, and maintenance of software systems. In recent years, software evolution has been recognized as one of the most important and challenging areas in the field of software engineering. Studies pointed out that 65-80% \[\text{LEH01}\] of a system's lifetime will be spent on maintenance and evolution activities. The majority of the costs of evolution of a software system are incurred in software comprehension, rather than in making the necessary corrections to the system. Available estimates indicate that the percentage of maintenance time consumed on software comprehension ranges from 50% up to 90% \[\text{COR89, LIV94, and STA84}\].

Software repositories, such as versioning systems and bug tracking systems are essential parts of supporting various software maintenance activities. These tools not only support software maintenance activities, but they also store important information related to software development and maintenance history. Given the often large amount of information stored in these repositories, researchers have proposed to mine and analyze these large knowledge bases in order to study and support various aspects of the evolution of software systems, such as impact analysis, software architecture, development process, software reuse, product reliability, and artifact traceability.

One of the key challenges in analyzing these software repositories is that they lack a common representation. These repository specific data models, often introduced as information silos, do not allow for a semantic rich integration of these resources and therefore limit the analysis support across repository boundaries. In order to address these
challenges with respect to information integration and the analysis across repository boundaries, a common, semantic rich representation is needed to integrate information from various software repositories.

In this thesis we introduce a common ontological representation to support the mining and analysis of software repositories. In addition to this common representation, we introduce the SVN-Ontologizer and Bugzilla-Ontologizer tools that provide automation for both data extraction from remote repositories and automated ontology population. A case study is presented to illustrate the applicability of the approach in supporting the analysis and mining of the repositories in order to provide support to software maintainers.

The remainder of the thesis is organized as follows: Section 2 introduces a general background related to software evolution, software repositories, and ontologies. Section 3 details the motivation and objectives of our approach. Section 4 and 5 introduces the implementation of SVN and Bugzilla-Ontologizer tools respectively. An initial case study is presented in Section 6, followed by related work in Section 7. Section 8 presents conclusions and future work.
2. Background

In an attempt to make this thesis self-contained, we review some background relevant to the presented research. Background on software evolution and software comprehension process is presented in Section 2.1. Section 2.2 introduces software repositories with a specific focus on Subversion, a version control system, and Bugzilla, a defect tracking repository. Section 2.3 provides a brief introduction to ontologies, semantic web technology, and their use as a modeling approach.

2.1. Software Evolution

Software evolution represents the cycle of activities involved in the development, use, and maintenance of software systems [WIKI09]. Software evolution includes software maintenance, which is defined as part of the IEEE Standard 1219[IEEE90] as, “the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment”. The term ‘software evolution’ is now often preferred as a replacement for ‘maintenance’ [KHV00]. Lehman [LEH80] concludes the fact that maintenance is evolutionary development. Organizations have made large investments in their software systems. These systems become often critical business assets. In order to maintain the value of these assets to the business, the software must evolve. All software systems are subject to such an evolution, as these systems must evolve over time as new requirements emerge, or they have to adapt and extend the existing functionalities to meet changed requirements [LEH01]. As a result, the majority of software budgets in large
organizations are devoted to evolve existing software in order to maintain the value of their software assets. Studies pointed out that 65 to 80% [LEH01] of a system lifetime will be spent on maintenance and evolution activities. Figure 2-1 shows the software life-cycle costs.

![Software Life-Cycle Costs](image)

It has also been shown [LEH01] that the majority of the maintenance costs are related to enhancements of the existing software product, rather than corrections.

One of the major reasons for the overall significant effort and cost involved in software evolution is the need to comprehend systems that are either not well documented or have out of date and inconsistent documentation. Whenever a change is made to a piece of software, it is important that the maintainer gains some understanding of the structure, behavior and functionality of the system being modified. As a consequence, maintainers spend a large amount of their time reading the code and the accompanying documentation to comprehend its logic, purpose, and structure [GCM00]. Available estimates indicate that the percentage of maintenance time consumed on software
comprehension ranges from 50% up to 90% ([COR89, LIV94, and STA84]). Software comprehension is necessary because the maintainer is rarely the author of the code (or a significant period of time has elapsed between development and maintenance) and a complete, up-to-date documentation is even more rarely available [CAG96].

Burd et al. [BUR98] define software comprehension as “the activity of understanding existing software systems”. Muller [MUL94] defines software comprehension as “the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself, to models of the underlying application domain, for maintenance, evolution, and reengineering purposes”. The author further states that software comprehension is “a process whereby a software practitioner understands a software artifact using both knowledge of the domain and/or semantic and syntactic knowledge, to build a mental model of its relation to the situation” [MUL94]. Many software comprehension models (i.e. Mental and Cognitive Models) have been proposed. These models help to better identify what information needs to be provided to maintenance programmers, and when and how this information should be provided [LET86]. Software comprehension models normally consist of four common elements, namely, a knowledge base, external representation, assimilation process, and mental models [MPO03]. Figure 2-2 shows the common elements of comprehension models.
Figure 2-2: Common elements of software comprehension models [MPO03]

An external representation corresponds to the external views available in assisting the maintainer comprehending a software system. This external support may be in the form of system documentation, the source code, expert advice from other maintainers familiar with problem domain or similar source code from the other system [MPO03]. A knowledge base can be defined as the maintainer’s accumulated knowledge prior to the attempt to understand the software system. It may consist of an understanding of the domain and general information that may be pertinent to that domain, along with programming standards and practices [MPO03]. The knowledge base develops and expands as the level of maintainer understanding the changes [MPO03]. Rouse et al. [RWM85] defined a mental model as “mechanisms whereby the humans are able to generate descriptions of system purpose and form, explanations of system functioning and observed system states, and predictions of future states”. Davis [DSP93] defines an assimilation process as “the actual strategy, which the programmer employs to comprehend the source code”. One method of assimilation is where maintainer’s hypotheses are refined and elaborated during comprehension [BRK83]. Storey mentioned in [STO01] that a mental model describes a developer's mental representation of the
program to be understood. A cognitive model describes the cognitive processes and temporary information structures in the programmer's mind that are used to form the mental model. In the past, several cognitive models have been developed to explain how maintainers comprehend the software system.

Letovsky [LET87] introduced a cognition model that consists of three main components: a knowledge base, a mental model, and an assimilation process. The first component contains the general knowledge that a programmer has about the programming discipline and the problem domain. It also includes rules of discourse, i.e. conventions in programming such as algorithm and data structure implementations and coding standards. The mental model is organized into three different levels of abstraction. First is the specification level, which describes the program goals. Second, the implementation level expresses the lowest level of abstraction, and contains the data structures and functions as entities. The third level of abstraction is the annotation level which links each goal in the specification level with its realization in the implementation level [CAG99].

Brooks [SOL84] proposed a model based on the top-down approach. The approach starts from the assumption that in the design phase a designer makes a number of decisions which will be reflected in the code. Comprehension involves recovering these decisions and mapping them onto the programming domain through the reconstruction of intermediate domains. The construction of the mental model happens through a top-down process that successively formulates and verifies hypotheses. At the top there is the primary hypothesis that expresses a high level description of the program function. Next, subsidiary hypotheses are formulated to support the primary hypothesis [CAG99].
Pennington [PEN87] proposed a bottom-up approach that starts by comprehending code line by line and discovering familiar patterns, called chunks, whose aggregation and next abstraction can bring to the identification of new patterns a higher level of abstraction. Pennington's model faces comprehension problems with the development of two different mental representations: the program model and the situation model. The first is a low level mental model of the program and its structure. Indeed, the first model that maintenance programmers build when dealing with unfamiliar code is typically a control flow abstraction. New and more abstract program models are then built by chunking code structures into more abstract structures. The situation model is developed after the program model. It creates a data flow/functional abstraction and requires knowledge of the application domain to mentally represent the code in terms of real-world objects organized as a functional hierarchy [CAG96].

Mayrhauser et al. [VMY93] proposed an integrated model that combines the models previously proposed. They start from the observation that a comprehension process proceeds either top-down, bottom-up, or a combination of the two. The integrated model consists of four main components: program model, situation model, top-down model, and knowledge base. The first three components are borrowed from the models already introduced. The integrated model exploits the top-down model when the code is familiar and the bottom-up model when it is completely new. By proceeding in a top-down way it can happen that an unfamiliar section of code is met, and a swap to the bottom-up investigation is required. The knowledge base is necessary for the construction of the other three components. Each model component consists of an internal representation of the code and the strategy to build this internal representation. The knowledge base
furnishes related information and knowledge which has been previously acquired. During understanding, new information is developed and stored in the knowledge base for future usage. [CAG96].

2.2. Software Repositories

Software repositories help the users to manage the progress of software projects. A repository refers to a central place where data is stored and maintained within a persistent storage. Repository distributions can be either shared across a network or locally hosted on an individual computer. Software repositories such as version control and defect tracking systems are common examples of repositories used as part of modern software engineering and software development processes. These repositories provide shared understanding of the development processes of the software product.

Revision control (also known as version control) is the management of multiple revisions of the same unit of information. Version control systems (VCS) such as CVS [XMB09] and SVN [SVN09] are widely used examples of version control systems. These version control systems keep the development history of software projects in order to avoid modification conflicts among different revisions. Version control systems play also an important role during software evolution, since changes performed as part of maintenance requests can be traced and tracked (e. g. who made what changes and when. ).

During the development of a software system certain problems may arise. Such problems can emerge from one revision to another and are referred to as “bugs.” Thus, bug tracking is a necessity in addition to a revision control system. When considering bug tracking we understand the storage and management of issues related to programmatic or
even systemic instabilities, faults or conflicts during development. These issues are stored in a dedicated bug or issue tracking system. These systems mainly consist of a database (open source or proprietary) where the data pertinent to a specific issue is stored. The client and administrator side access layers (usually web based access, like Bugzilla [BUG09]).

2.2.1. Subversion (SVN)

Subversion (SVN) is a free/open-source version control system; it manages files and directories. Subversion places these files and directories into a central repository [SVN09]. The Subversion repository supports the tracking of changes to files and directories. SVN furthermore allows for the recovering of older versions of data, or examining the history how the data changed.

History of Subversion

CollabNet [CLB09] offers a collaboration software suite called CollabNet Enterprise Edition (CEE) [CL09A] of which one component is version control. Before August 2001, CEE used CVS (Concurrent Version System) as its initial version control system. CVS's limitations were obvious from the beginning. In early 2000, CollabNet planned to develop a new version control system from scratch which would match CVS's features and preserve the same development model but not duplicate CVS's most obvious flaws. It did not need to be a drop-in replacement for CVS. It should be similar enough that any CVS user could make the switch with little effort. After fourteen months of coding, the Subversion (SVN) became fully functional on August 31, 2001 by replacing CVS [SV09B].
Architecture of Subversion

Subversion can access its repository across networks, enabling a collaborative environment for users where they can modify and manage the same set of data. Figure 2-3 shows the architecture of SVN repository. Subversion works in two ways. First, it provides repository that holds all of the versioned data. On the other end it works as a client program, which manages the local operations of the portions of that versioned data called working copies. Between these two ends there are multiple routes through various Repository Access (RA) layers. Some of these routes go across computer networks and
through network servers that then access the repository. Others bypass the network altogether and access the repository directly.

**Functions and Features**

The basic functionalities provided by SVN are the same as in CVS, including the storage of file history information about users who checked out a working copy of a file to work locally on it. Users can easily compare the different versions of the file. In next paragraph we will discuss features exclusively supported by SVN.

SVN provides branching, tagging, and release concepts, where tags are common file metadata that are managed and kept in files or directories. Branches are separate directory trees made out of current main “trunk” directory. When a branch is made for a file, the revision enumeration continues on. The only property that changes is the path to the file or directory that moved from the main trunk to a branch. SVN tracks the changes made to both the main trunk and the branch as a log of the same file, telling the user where a particular change (main trunk or branch) was made and whether a revision corresponds to the main trunk or the branch. Figure 2-4 shows the example of SVN repository directory structure.

Releases are sets of revisions without explicit concepts or mechanisms corresponding to releases. Subversion creates branches and tags by simply copying the project, using a mechanism similar to a hard-link.
Subversion supports add, delete, copy, and rename both files and directories. Every newly added file begins with a fresh and clean history of its own. Subversion provides consistent data handling by providing differences of the files in binary and human readable format. Subversion allows atomic commits; it stores complete collections of modification into repository.

Subversion uses a copy-modify-merge model as an alternative to locking. In this model, each user's client contacts the project repository and creates a personal working copy (i.e. a local reflection of the repository files and directories). Users then work simultaneously and independently to modify their private copies.

Finally, the private copies are merged together into a new, final version. The version control system provides support for merging, but ultimately, a human being is responsible for the final merging decisions.

Table 2-1 provides an overview of functionalities of some popular version control systems (including Subversion).
<table>
<thead>
<tr>
<th>Features</th>
<th>Subversion</th>
<th>CVS</th>
<th>BitKeeper</th>
<th>Git</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic Commits</td>
<td>Commits are atomic.</td>
<td>CVS commits are not atomic.</td>
<td>need to verify</td>
<td>Commits are atomic.</td>
</tr>
<tr>
<td>Files / Directories Moves or Renames</td>
<td>Supported</td>
<td>Not supported.</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Intelligent Merging after Moves or Renames</td>
<td>Partially supported.</td>
<td>Not supported.</td>
<td>Partially supported</td>
<td>Not supported.</td>
</tr>
<tr>
<td>File and Directories Copies</td>
<td>Supported also utilized for branching.</td>
<td>Not supported.</td>
<td>Supported.</td>
<td>Not supported.</td>
</tr>
<tr>
<td>Remote Repository Replication</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>very intrinsic feature of Git.</td>
</tr>
<tr>
<td>Propagating Changes to Parent Repositories</td>
<td>supported</td>
<td>Not supported</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>Repository Permissions</td>
<td>The WebDAV-based service supports defining HTTP permissions.</td>
<td>Limited. &quot;Pre-commit hook scripts&quot;</td>
<td>FILL IN</td>
<td>No, but a single server can serve many repositories.</td>
</tr>
<tr>
<td>Change sets Support</td>
<td>Partial support. There are implicit change set that are generated on each commit.</td>
<td>Not supported Changes are file-specific.</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Branching and Merging</td>
<td>Supported</td>
<td>Not Supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Tracking Uncommitted Changes</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>Command Set</td>
<td>A CVS-like command set.</td>
<td>A simple command set that includes (cvs commit, cvs update and cvs checkout) and several others.</td>
<td>A CVS-like command set</td>
<td>Command set is very feature-rich, and not compatible with CVS.</td>
</tr>
<tr>
<td>Portability</td>
<td>Excellent. Clients and Servers work on UNIX, Windows and Mac OS X.</td>
<td>Good. Client works on UNIX, Windows and Mac OS. Server</td>
<td>Very good. Binaries are available for most common UNIX systems and for Windows 98 and above.</td>
<td>The client works on most UNIXes, but not on native MS-Windows.</td>
</tr>
<tr>
<td>Web Interface</td>
<td>Supported</td>
<td>Supported</td>
<td>Its own built-in web tool</td>
<td>Its own built-in web tool. Gitweb</td>
</tr>
<tr>
<td>Availability of Graphical User-Interfaces</td>
<td>There are many available GUIs: RapidSVN (cross-platform), TortoiseSVN (Windows Explorer plug-in), Jsvn (Java), etc. development.</td>
<td>There are many available GUIs: WinCVS, Cervisia (for KDE), TortoiseCVS (Windows Explorer plug-in).</td>
<td>BitKeeper ships with several GUIs for performing common tasks.</td>
<td>Git is included in distribution. Qgit and Git-gui tools are also available.</td>
</tr>
</tbody>
</table>

*Table 2-1: Comparison of some version control systems [VEB09]*
2.2.2. Bugzilla

Bugzilla [BUG09] is a bug or issue tracking system. Bug tracking systems allow individuals or groups of developers to keep track of outstanding problems with their product effectively.

History

Bugzilla was originally developed by Terry Weismann in a programming language called TCL to replace a rudimentary bug-tracking database used internally by Netscape Communications. Weismann later ported Bugzilla to Perl from TCL, and it remains in Perl to this day. Most of the commercial defect tracking software vendors at the time charged enormous licensing fees. Being an open source project, Bugzilla became a favorite of the open-source crowd (with its genesis in the open-source browser project, Mozilla) [BUG03]. Initially, Bugzilla was used to manage issues in the Mozilla Foundation projects. Now external projects (both open source and proprietary), can submit their bug reports too. It has become the de-facto standard defect-tracking system against which all others are measured.

Architecture and Functionality

Bugzilla is a web-based, open-source issue tracking tool. It is the most widely used web based tool to manage bugs. Bugzilla can also track enhancements, feature requests, and to-do items. Bugzilla allows individuals or groups of developers to keep track of outstanding problems with their product effectively.

The architecture of Bugzilla as a tool is rather simple. It requires an installed server and a database management system (PostgreSQL, MYSQL, etc.) to be operational. Further,
Bugzilla requires a suitable release of Perl 5 along with a set of Perl modules for the installation and a mail transfer agent, such as Sendnote, qmail, Postfix or Exim.

The central concept of the Bugzilla is the issue, all other information within the Bugzilla database being directly associated with an issue. As a result, issues cannot be merged, branched, or versioned. One issue can block another issue, which can be in a different
state depending on the priority of bugs. Bugzilla is also used to file feature requests and enhancements.

Bugzilla follows the life cycle of the bug as shown in Figure 2-5. When a bug is submitted, it enters the state “new” as either confirmed or unconfirmed. Then it is assigned to a developer. When the developer has resolved the bug, it can either be verified, if the solution worked out, or it can be reopened if the solution was not satisfying. If a bug is verified it is closed.

This life cycle is currently hard-coded into Bugzilla. It manages the entire work-flow for a bug and defines clear states a bug goes through. Further, Bugzilla stores comments from different users, activities performed on the bug, and files attachments attached by users for the bug. Table 2-2 compares some popular bug tracking systems, including Bugzilla.

<table>
<thead>
<tr>
<th>Bug Tracking System</th>
<th>Integration with version control</th>
<th>Test Planning integration</th>
<th>Customizable Workflow</th>
<th>Unicode Support</th>
<th>LDAP user Authentication</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIRA</td>
<td>ClearCase, AccuRev, Perforce, CVS, Subversion, Visual SourceSafe (beta)</td>
<td>Atlassian Bamboo (continuous integration &amp; testing, via plug-in)</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>eTraxis</td>
<td></td>
<td></td>
<td>Supported, unlimited # of workflow templates</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>BugTracker.NET</td>
<td>Subversion</td>
<td></td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Debbugs</td>
<td>VCS Agnostic, DAK integration</td>
<td>Not Supported</td>
<td>Not Supported</td>
<td>Supported</td>
<td>N/A</td>
</tr>
<tr>
<td>Bugzilla</td>
<td>CVS, Subversion, Perforce, AccuRev</td>
<td>Testopia</td>
<td>Supported, as of Bugzilla 3.2</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Mantis</td>
<td>Supported, support for CVS, Subversion</td>
<td>Not Supported</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
</tbody>
</table>

*Table 2-2: Comparison of some bug tracking systems* [WKB09]
2.3. Ontologies

Ontology is a specification of a conceptualization. In the context of computer and information sciences, ontology defines a set of representational primitives with which to model a domain of knowledge or discourse [GUR93]. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members) [GUR93].

2.3.1. Why Ontology?

Ontologies have been widely used to conceptualize and define domains of interest. Ontologies include machine-interpretable definitions of basic concepts in the domain and relations among them. Why would someone want to develop an ontology?

*Sharing common understanding* of information structures among people or software agents is one of the more common goals in developing ontologies [MUS92, GUR93]. For example, several different web sites contain medical information or provide medical e-commerce services. If these web sites share and publish the same underlying ontology of the terms they all use, then computer agents can extract and aggregate information from these different sites. The agents can use this aggregated information to answer user queries or as input data to other applications.

*Enabling reuse of domain knowledge* was one of the driving forces behind recent development in ontology research. For example, models for many different domains need to represent the notion of time. This representation includes the notions of time intervals, points in time, relative measures of time, and so on. If one group of researchers develops such ontology in detail, others can simply reuse it for their domains. Additionally, if we
need to build a large ontology, we can integrate several existing ontologies describing portions of the large domain.

*Making explicit domain assumptions* underlying an implementation makes it possible to change these assumptions easily if our knowledge about the domain changes. A hard-coding assumption about the world in programming-language code makes these assumptions not only hard to find and understand but also hard to change, in particular for someone without programming expertise. In addition, explicit specifications of domain knowledge are useful for new users who must learn what terms in the domain mean.

*Separating domain knowledge from operational knowledge* is another common use of ontologies. We can describe the task of configuring a product from its components according to a required specification and implement a program that does this configuration independent of the products and components themselves [MGW00].

*Analyzing domain knowledge* is possible once a declarative specification of the terms is available. Formal analysis of terms is extremely valuable for reusing existing ontologies and extending them [MGW98].

Ian [IAN07] illustrates some of the basic differences between ontologies and databases. Table 2-3 compares ontologies with databases observed by Ian [IAN07].
Ontologies

Open world assumption (OWA)
- Missing information treated as unknown

No Unique name assumption (UNA)
- Individuals may have more than one name

Ontology axioms behave like implications (inference rules)
- Entail implicit information

Ontology axioms play a powerful and crucial role
- Answer may include implicitly derived facts
- Can answer conceptual as well as extensional queries
- Query answering amounts to theorem proving (i.e. logical entailment)

Databases

Closed world assumption (CWA)
- Missing information treated as false

Unique name assumption (UNA)
- Each individual has a single, unique name

Schema behaves as constraints on structure of data.
- Define legal database states

In Database querying, Schema plays no role
- Data must explicitly satisfy schema constraints.
- Query answering amounts to model checking (i.e. a “look-up” against the data).

Table 2-3: Ontologies vs. databases [IAN07]

Uschold et al., [MUG04] also discuss some interesting differences between ontologies and databases. Table 2-4 describes the difference between databases and ontologies mentioned by [MUG04].

<table>
<thead>
<tr>
<th><strong>Ontologies</strong></th>
<th><strong>Databases</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontologies have a range of purposes including interoperability, search, and software specification. One or more parties commit to using the terms from the ontology with their declared meaning.</td>
<td>The primary use of most DB schema is to structure a set of instances for querying a single database. This difference impacts heavily on the role of constraints.</td>
</tr>
<tr>
<td>For ontologies, constraints are called axioms. Their main purpose is to express machine-readable meaning to support accurate automated reasoning. This reasoning can also be used to ensure integrity of instances in a knowledge base.</td>
<td>For databases, the primary purpose of constraints is to ensure the integrity of the data (i.e. instances). These ‘integrity constraints’ can also be used to optimize queries and help humans infer the meaning of the terms.</td>
</tr>
<tr>
<td>The main role for cardinality constraints in ontologies is to express meaning, and ensure consistency (either of the ontology, or of instances).</td>
<td>Cardinality and delete constraints are important kinds of integrity constraints which have highly DBspecific uses those are outside the scope of most or all ontology systems.</td>
</tr>
<tr>
<td>Support for taxonomic reasoning: it is fundamental for nearly all ontology applications</td>
<td>It is not supported by most DBMS.</td>
</tr>
<tr>
<td>Reasoning over ontologies normally is done by general logic-based theorem provers, specific to the language. The fundamental role of a reasoning engine is to derive new information via automated inference. Inference can also be used to ensure the logical consistency of the ontology itself.</td>
<td>Logical consistency through Reasoner is not supported by most DBMS.</td>
</tr>
</tbody>
</table>

Table 2-4: Ontologies vs. databases [MUG04]
Robert et al. [RJS99] highlight some of the benefits of using ontologies as an enabling technology for interpersonal communication and inter-operability. For communication between people, an unambiguous but informal ontology may be sufficient. Interoperability among computer systems can be achieved by translating between the different modeling methods, paradigms, languages, and software tools. The ontology is used as an interchange format.

**Systems Engineering Benefits**

**Re-Usability:** The ontology is the basis for a formal encoding of the important entities, attributes, processes and their inter-relationships in the domain of interest. This formal representation may be (or become through by automatic translation) a re-usable and/or shared component in a software system.

**Search:** Ontology may be used as meta-data serving as an index for a repository of information.

**Reliability:** A formal representation also makes possible the automation of consistency checking resulting in more reliable software.

**Specification:** The ontology can assist the process of identifying requirements and defining a specification for an IT system (knowledge based or otherwise).

**Maintenance:** The use of ontologies in system development, or as part of an end application, can render maintenance easier in a number of ways. Systems which are built using explicit ontologies serve to improve documentation of the software, which in turn reduces maintenance costs. Maintenance is also an important benefit if ontology is used
as a neutral authoring language with multiple target languages - it only has to be maintained in one place.

**Knowledge Acquisition:** Speed and reliability may be increased by using an existing ontology as the starting point and basis for guiding knowledge acquisition when building knowledge-based systems.

**Reasoning Services:** Reasoning refers to the evaluation of ontologies according to their specifications, including:

- Checking consistency of the ontology
- Checking concept (and role) consistency
- Concept (and role) subsumption
- Instance checking
- Instance retrieval
- Query answering

### 2.3.2. Applications

Ontologies are part of the W3C standards stack for the Semantic Web, for which they are used to specify standard conceptual vocabularies to enable exchange of data among systems. Furthermore, they are the basis for providing services for answering queries, publishing reusable knowledge bases, and offering services to facilitate interoperability across multiple, heterogeneous systems and databases. The key role of ontologies with respect to database systems is to specify a data modeling representation at a level of abstraction above specific database designs (logical or physical), so that data can be exported, translated, queried, and unified across independently developed systems and
services. Successful applications to date include database interoperability, cross database searches, and the integration of web services. Figure 2-6 shows an example of ontology.

![Ontology Diagram](image-url)

*Figure 2-6: An example of ontology [MAR09]*

Robert et al. [RJS99] describe some ontology applications as follows:

**Neutral Authoring:** An information artifact is authored in a single language and is converted into a different form for use in multiple target systems. Benefits of this approach include knowledge reuse, improved maintainability, and long term knowledge retention.

**Ontology as Specification:** An ontology of a given domain is created and used as a basis for specification and development of some software. Benefits of this approach include documentation, maintenance, reliability, and knowledge re-use.
Common Access to Information: A piece of information is required by one or more persons or computer applications, but is expressed using unfamiliar vocabulary or in an inaccessible format. The ontology helps render the information intelligible by providing a shared understanding of the terms or by mapping between sets of terms. Benefits of this approach include inter-operability and more effective use and reuse of knowledge resources.

Ontology-Based Search: Ontology can be used for searching an information repository for desired resources (e.g. documents, web pages, names of experts). The chief benefit of this approach is faster access to important information resources, which leads to more effective use and reuse of knowledge resources.

2.3.3. Web Ontology Language OWL and SPARQL

OWL stands for Web Ontology Language. The OWL Web Ontology Language is designed for use by applications that need to process the content of information instead of just presenting information to humans [W3C09]. Following are the data formats supported by Web Ontology language.

- XML provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents.

- XML Schema is a language for restricting the structure of XML documents and also extends XML with data types.
• RDF is a data model for objects ("resources") and relations between them. It provides simple semantics for this data model, and these data models can be represented in the XML syntax.

• RDF Schema is a vocabulary for describing properties and classes of RDF resources, with semantics for generalization-hierarchies of such properties and classes.

OWL is currently available in following three different types.

• OWL Full is the full specification of the language.

• OWL DL is a subset of OWL Full, making some restrictions to allow automated reasoning.

• OWL Lite is a subset of OWL DL as a simple-to-use, simple-to-implement version of OWL.

The OWL format has four major concepts to store information and its associations.

• Classes are abstract definitions of a single concept. Classes define possible associations and properties they can have. A class itself does not store concrete data. It only acts as a container concept.

• Individuals (also called instances) are the concrete realizations of a class. They can only have associations and store data in the defined manner of their respective class.

• Object properties define the associations between two classes (abstract) or two individuals (concrete). Object properties are directed associations and always
belong to a specific domain (i.e. the starting point of an association) and a range (i.e. the endpoint). Domain and range can both be a list of multiple Classes.

- Data type properties can be, like object properties, considered as associations. Unlike object properties, the range is not a list of classes but rather a predefined data type. Typically the data types of XML Schema [W3C, 2004b] are used.

SPARQL is the W3C standard query language for semantic web OWL/RDF data. In order to retrieve data using SPARQL, a triple template is defined in the query. The core idea is to leave the subject or object of a triple blank variable and the query engine will try to find all the triples matching this template.

It provides facilities to:

- Extract information in the form of URIs, blank nodes, plain and typed literals.
- Extract RDF sub graphs.
- Construct new RDF graphs based on information in the queried graphs.

2.3.4. Ontology editing tools

Protégé release 3.4

Protégé is a free, open-source platform that provides a suite of tools to construct domain models and knowledge-based applications with ontologies. Protégé implements a rich set of knowledge-modeling structures and actions that support the creation, visualization, and manipulation of ontologies in various representation formats. Protégé can be customized to provide domain-friendly support for creating knowledge models and entering data. Furthermore, Protégé can be extended by way of plug-in architecture and a Java-based
Application Programming Interface (API) for building knowledge-based tools and applications [PRG09].

**TopBraid Composer (standard edition)**

TopBraid Composer is an enterprise-class modeling environment for developing Semantic Web ontologies and building semantic applications. TopBraid Composer is implemented as an Eclipse plug-in. TopBraid Composer is a professional development environment for W3C's Semantic Web standards RDF Schema, the OWL Web Ontology Language, the SPARQL Query Language and the Semantic Web Rule Language (SWRL). Composer can be used to edit RDFS/OWL files in various formats and also provide scalable database back ends (Jena, AllegroGraph, Oracle 11g and Sesame) as well as multi-user support.

Composer provides a comprehensive set of features to cover the whole life cycle of semantic application development. In addition to being a complete ontology editor with refactoring support, Composer also can be used as a run-time environment to execute rules, queries, and reasoners. Based on Eclipse, Composer can also be extended with custom Java plug-ins. This supports the rapid development of semantic applications in a single platform.

**2.3.5. Ontological Reasoners**

An ontological reasoner is a piece of software able to infer logical consequences from a set of asserted facts or axioms. The notion of a semantic reasoner generalizes that of an inference engine by providing a richer set of mechanisms to work with. The inference rules are commonly specified by means of an ontology language, and often a description
language. Many reasoners use first-order predicate logic to perform reasoning. The inference commonly proceeds by forward chaining and backward chaining [REA09].

**Pellet reasoner**

Pellet is an open source reasoner for OWL DL written in Java. It provides reasoning service for OWL ontologies. Pellet allows reasoning for semantically-enabled applications that need to represent and reason about information using OWL [PAL09]. Pellet is an OWL DL reasoner based on the tableaux algorithms developed for expressive Description Logics. It supports the full expressivity OWL DL including reasoning about nominal’s (enumerated classes). The core of the Pellet reasoner is the tableaux reasoner that checks the consistency of a KB, (i.e. a pair of an ABox and a TBox). The reasoner is coupled with a data type oracle that can check the consistency of conjunctions of (built-in or derived) XML Schema simple data types. The OWL ontologies are loaded to the reasoner after a step of species validation and ontology repair. This step ensures that all the resources have an appropriate type triple (a requirement for OWL DL but not OWL Full) and missing type declarations are added using some heuristics. During the loading phase, axioms about classes (subclass, equivalent class or disjointness axioms) are put into the TBox component and assertions about individuals (type and property assertions) are stored in the ABox component. TBox axioms go through the standard preprocessing of DL reasoners before they are fed to the tableaux reasoner. Figure 2-7 shows the architecture of a Pellet reasoner.
Figure 2-7: Architecture of a Pellet reasoner [PL09A]

**Jena Semantic Web Framework**

Jena [JEN01] is an open-source Semantic Web framework developed in Java language. Jena framework is used to create and populate RDF models, to persist them to a database, and to query these RDF models programmatically using SPARQL query language. Jena’s reasoning service capabilities can be used to infer knowledge about models from ontology.

Table 2-5 compares some popular reasoners available.
<table>
<thead>
<tr>
<th>OWL-DL Entailment</th>
<th>Pellet</th>
<th>KAON2</th>
<th>RacerPro</th>
<th>Jena</th>
<th>FaCT++</th>
<th>OWLIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported expressivity for reasoning</td>
<td>SROIQ(D)</td>
<td>SHIQ(D)</td>
<td>?</td>
<td>varies by reasoner (incomplete for nontrivial description logics)</td>
<td>SROIQ(D)</td>
<td>R-entailment</td>
</tr>
<tr>
<td>Reasoning algorithm</td>
<td>Tableau</td>
<td>Resolution &amp; Datalog</td>
<td>Tableau</td>
<td>Rule-based</td>
<td>Tableau</td>
<td>Rule-based</td>
</tr>
<tr>
<td>Consistency checking</td>
<td>Yes</td>
<td>?</td>
<td>Yes</td>
<td>Incomplete for OWL DL</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DIG Support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Rule Support</td>
<td>Yes (SWRL -- DL Safe Rules)</td>
<td>Yes (SWRL -- DL Safe Rules)</td>
<td>Yes (Own rule format)</td>
<td>No</td>
<td>Yes (Own format)</td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>2.0 RCS</td>
<td>Unknown</td>
<td>1.9.2</td>
<td>2.5.4</td>
<td>1.1.8</td>
<td>2. x/3. x</td>
</tr>
</tbody>
</table>

*Table 2-5: Comparison of available reasoners [REA09]*

### 2.3.6. Ontology Alignment

Aligning ontologies means “establishing links between two or more ontologies” and allowing the aligned ontologies to reuse information from one another [NFN07].

Aligning ontologies amounts to defining a distance between entities (which can be as reduced as an equality predicate) and computing the best match between ontologies, (i.e. the one that minimizes the total distance or maximizes a similarity measure) [JEP04].

Several methods are introduced by researchers to calculate distances between entities in ontologies [JEP04]:

- **Terminological** compares the labels of the entities.
• *String based* does the terminological matching through string structure dissimilarity (e.g., edit distance).

• *Internal structure comparison* comparing the internal structure of entities (e.g., the value range or cardinality of their attributes).

• *External structure comparison* compares the relations of the entities with other entities.

• *Taxonomical structure* compares the position of the entities within taxonomy.

• *Extensional comparison* compares the known extension of entities, i.e. the set of other entities that are attached to them (in general instances of classes).

• *Semantic comparison* compares the interpretations (or more exactly the models) of the entities.

The process of alignment creates a mapping between two input ontologies. The mapping is a set of anchors between the two ontologies (i.e. an edge connecting two elements of the ontologies).
3. Contribution

In this section we will first introduce the general motivation for our approach. We discuss the need for modeling and analysis of software repositories like bug trackers and version control systems. Next we will present the research hypothesis and the specific sub-goals which will be addressed as part of this thesis.

3.1. Motivation

According to Lehman, "a software system must evolve, or it becomes progressively less useful" [LEH97]. Software evolution involves both the comprehension and modification of existing software systems. Given the collaborative nature of software development, various software repositories like versioning systems and bug trackers are used to support the evolvability of the software system. When the software system evolves, changes made to source code and other documents are stored in software repositories. Software repositories contain valuable information about the development history of the software project. There is a great potential by mining and analyzing such historical information to support the evolution of software systems.

In recent years there has been a trend to use the information stored in these software repositories to provide the maintainers with additional support during the evolution of large software systems [CAC01, DTT05, GDM04, DMG04, TQG02, and HSK05].

For the detailed references we will discuss some efforts done in past to support software evolution.

As Antoniol [GIO04] states that, the software repositories like bug repositories and version control systems represent valuable sources of information to study software
evolution. Version control systems can be mined to gain insights about the evolvability of a system by analyzing other properties like the size, complexity, and the amount of changes stored in the repository. Bug reporting systems on the other hand can provide additional insights on the reliability of a system, as well as the management of defects (e.g., average defect fixing rate and statistics about defect severity) [GIO04].

[AEH06] Hassan et al. came in their work to a similar observation, stating that mining historical information from software repositories can support both developers and managers in their endeavors to build and maintain complex software systems.

As pointed out in [JIM07], by mining and analyzing software repositories it is possible to recover traceability links among different repositories to support the evolvability of the software systems. Some of the potential applications for these traceability links are the support for impact analysis, software comprehension, and requirements assurance of high quality systems [JIM07]. Software repositories have also been analyzed in [BJM03, GHH98, DMG04VAR04, VR04S, ZTT05 and ZT05A] to support the prediction of software change.

Software repositories (i.e. SVN repository and Bugzilla repository) typically use different types of persistent storage and schemas, which makes it inherently difficult to share and link information among these repositories. The information gathered from these software repositories is not structured. It is not easy to interconnect traces related to the same artifact in different sources.

Among the challenges faced by researchers, one of the key challenges in mining and analyzing software repositories is that they lack a common representation. The lack of
common representation does not allow for a semantic rich integration of these resources, and therefore limits the analysis support across the repository boundaries.

In order to address these challenges with respect to information integration and the analysis across repository boundaries, a common, semantic rich representation is needed to integrate the information from various software repositories. In this thesis we introduce a common ontological representation to support the mining and analysis of software repositories. The proposed common ontological representation will allow for efficient mining and analysis of software repositories (i.e. SVN and Bugzilla) to support software evolution.

**Research Hypothesis**

*A common ontological representation can be established among software repositories to support the evolution of software systems.*

3.2. Specific Contributions (Sub-goals) and Acceptance Criteria

The goal of our research is to provide a common ontological representation for software repositories for mining and analysis in order to support the evolution of software systems. We divide the general research goal into some more specific sub-goals to be addressed by this thesis:
Establishing a Common Ontological Representation

Software repositories, like versioning systems and defect tracking systems store valuable information for the evolution of a software project. The information stored in these repositories has a different format and representation. A user needs different tools to extract, integrate, and analyze the information stored in these repositories. For example, the SVN repository data can be retrieved through different client software available, such as Tortoise SVN. The Bugzilla repository information can be retrieved by the web interface tools provided by the Bugzilla installation. The information extracted from these repositories is in raw format. In order to process and analyze the raw data extracted from software repositories, as well as to identify the relationships and the dependencies between them, users require manual efforts and different types of analysis tools.

Nowadays, software development is a complex task; many large systems are interconnected with other systems. These systems produce a huge amount of information for software repositories. The process of extracting, combining, and analyzing such software repositories is more complex, time consuming, and requires much manual effort.

In order to deal with the stated problems, we propose a common ontological representation based on the Web Ontology Language (OWL) in order to integrate the information from different software repositories.

The Web Ontology Language (OWL) provides a semantic rich and meaningful way to store the information extracted from the software repositories. Standard OWL processing tools allow for immediate processing of the information in terms of visualization, editing, querying, and debugging.
By using the OWL standard tools, there is no need to write the code or to use the complicated command line tools. Compared to other formats and tools, OWL enables treating of data based on its semantics.

A common representation for the software repositories allows analysis across software repository boundaries. Additionally, it allows analysis of the relationships and dependencies among different artifacts.

**Automated Ontology Population**

The process of connection, data extraction, and parsing raw data extracted from the different software repositories is a complex and a time consuming job. Since the software repositories store information in different formats, they need different types of connectivity profiles, as well as the tools for extraction and parsing the raw data.

We introduce an approach which automates the process of the connection, extraction, and refinement (i.e. the transformation of raw data) of the ontology population. The proposed automated approach will save the time consumed in the manual efforts, as well as provide a safe and error-free way to populate the ontology.

**Implementation of an Automated Tool**

In addition to a common ontological representation, we introduce the SVN Ontologizer and Bugzilla Ontologizer tools that provide automation for both data extraction from remote repositories and automated ontology population.
Mining and Analysis across the Repository Boundaries, in order to Support the Evolution of a Software System

As discussed earlier, software researchers have recognized the benefits of the mining software repository data. The information stored in software repositories is a valuable source to support the evolution of a software system. By mining the software repositories’ information, we can improve the software design/reuse and recover the traceability links between different artifacts as well. The traceability links between the different artifacts can help to understand the relationships and dependencies among them.

As discussed in section 2, the one of the key aspects in software evolution is software comprehension. Our proposed common ontological representation supports bottom-up software comprehension. The bottom-up software comprehension approach is very useful for developers and maintainers, who have little or no knowledge of the existing software systems. The bottom-up approach helps the developers and the maintainers by gathering information from low level to abstract level. As mentioned in [BOT03], understanding is built from the bottom-up approach by reading the code and then mentally chunking or grouping these lines of code into higher-level abstractions. Analyzing software repositories across boundaries leads to better software comprehension.

The bottom-up approach to software comprehension primarily addresses situations where the developer or maintainer is unfamiliar with the domain. Several top-down models of software comprehension have been proposed to address the alternative situation, where the developer or maintainer has some previous domain exposure.
Analyzing the information stored in the software repositories also provides knowledge and understanding of:

- **Explicit concepts**, which are directly implemented in the source code as variables, executable code snippets, methods and classes.

- **Implicit concepts**, which are the assumptions that underlie parts of the code but are not directly implemented. For example, many applications assume that only one user is working with them; no specific code can be identified as the implementation of this single-user concept. If such an application is requested to support multiple users, programmers would have to change the implicit concept of the user to the explicit one, which requires substantial effort.

A common ontological representation allows for ease in the mining and analysis of the software repositories. Effective mining and analysis support the effective evolution of the software systems. Additionally, common ontological representation allows re-use of the information extracted from software repositories.

### 3.3. Acceptance Criteria

We expect our research hypothesis to hold if the following acceptance criteria can be validated:

- Establishing common ontological representation among software repositories
- Automated ontology population
- Implementation of SVN and Bugzilla Ontologizer tools
- Mining and analysis across the software repositories (i.e. SVN and Bugzilla repositories) in order to support the evolution of a software system.
4. Subversion Ontology

4.1. Subversion Ontology Design

For the design of the SVN ontology, we applied a three step development process. First, the existing schema of the SVN repository was extracted in order to identify and analyze the major concepts and their relationships modeled in the repository. Next, we applied a one-to-one mapping between the extracted relational SVN repository schema and an initial SVN ontology. In the last design step, we enriched and optimized our SVN ontology with new constraints and relations in order to be able to take advantage of ontology-specific modeling techniques and reasoning services.

4.1.1. SVN Repository Schema

Figure 4-1 provides an overview of the relational data schema extracted for the SVN repository. In what follows, we describe in more detail the modeled entities and relationships, since they are also going to be reflected in our ontological representation of the SVN repository.
Figure 4-1: SVN relational schema

**The File-Revision Relation**

The SVN repository manages both the directories (i.e. SVN branches/trunk) and the files that are committed to the repository. Within the relational data model, SVN does not distinguish between individual files or directories containing several files. Consequently, revisions, with revision number being the main attribute, are automatically associated through a many-to-one relation with the file entity. A file entity can have multiple revisions associated, whereby a particular revision belongs to a single file.
The Revision-Branch Relation

SVN supports the use of multiple parallel lines of development (branches). When a developer creates a branch in SVN, a new file is being created, yet the branch file remains invisible to the developer. Internally, SVN automatically creates a new sub-directory when a developer creates a branch. As a part of creating a branch, SVN creates first a transaction tree, then after a commit the transaction tree becomes a revision tree with the new branch as a sub-folder or file. The same procedure is applies for all commits performed.

The Revision-Transaction Relation

SVN defines what corresponds to a transaction as part of the relational schema. A transaction in SVN is used to distinguish uniquely a set of operations that lead to a new revision of a file. A transaction in SVN therefore represents a set of operations that apply to a file before the current revision number is updated.

The File-Modification-Info Relation

A modification report for a file in SVN can be extracted from the history log, which is available for each committed file. In SVN, a log corresponds to a listing of different modifications related to each revision of a file. SVN maintains the file and the modification information separately. As a result, the modification report log contains information about the particular action being performed (i.e. modified, added, deleted), the timestamps, the log message, the author, etc. for each revision.
4.1.2. Initial Mapping SVN Repository Schema to an Ontological Model

Description logic (DL) allows representing domain knowledge by defining relevant concepts called classes or TBox \[JRL06\]. As part of our ontological model for SVN, we define an initial TBox for our SVN ontology, which corresponds closely to a mapping of the existing SVN data repository schema. Figure 4-2 shows the resulting initial SVN ontology model. In order to define an initial TBox, we used Protégé release 3.4 ontology editor \[PRG09\]. The major entities are files, releases, and revisions. With every change to a file (commit) the revision-numbers increased to mark them as a new version. Older revisions of these files are still available and can be rolled back to. A revision represents the history of a specific file. A release combines a specific set of file revisions to a version that can be identified by its own unique name. For our initial ontological model we introduced three classes: revision, release, and file. We added a new concept, FileRevision, to our ontology model. The concept of a FileRevision is introduced to establish the relationship between File and Revision within a particular release.
Table 4-1 describes the main classes of our initial SVN ontology design.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Disjoint With</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Revision, Release, FileRevision.</td>
<td>File from the file system</td>
</tr>
<tr>
<td>Revision</td>
<td>File, FileRevision, Release.</td>
<td>Revision denotes version of the file</td>
</tr>
<tr>
<td>Release</td>
<td>Revision, File, Release.</td>
<td>Tag with multiple Revisions and Files</td>
</tr>
<tr>
<td>FileRevision</td>
<td>Release, File, Revision.</td>
<td>Combination of File and Revision-number</td>
</tr>
</tbody>
</table>

Table 4-1: Main classes of initial SVN ontology

Table 4-2 lists the various data properties modeled in the SVN ontology and a description of these properties. The data type properties allow the definition of the relations between instances of classes and RDF literals and XML Schema data types.
<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>String</td>
<td>File</td>
<td>The state describes the status of File in the Revision, (Added, Modified and Deleted)</td>
</tr>
<tr>
<td>creationDate</td>
<td>dateTime</td>
<td>Revision</td>
<td>Date when Revision was created</td>
</tr>
<tr>
<td>creationTime</td>
<td>dateTime</td>
<td>Revision</td>
<td>Time when Revision was created</td>
</tr>
<tr>
<td>fullPath</td>
<td>String</td>
<td>File</td>
<td>Full path of the File in SVN repository.</td>
</tr>
<tr>
<td>author</td>
<td>String</td>
<td>Revision</td>
<td>Name of the user who created this Revision</td>
</tr>
<tr>
<td>number</td>
<td>String</td>
<td>Revision</td>
<td>Revision-number</td>
</tr>
<tr>
<td>commitMessage</td>
<td>String</td>
<td>Revision</td>
<td>Text message entered at the time when Revision was created.</td>
</tr>
<tr>
<td>releaseDate</td>
<td>dateTime</td>
<td>Release</td>
<td>Time stamp for Release</td>
</tr>
</tbody>
</table>

*Table 4-2: Data type properties in SVN ontology*

An object property is a binary relation between instances of two classes. In order to restrict the relation of an object property, we specified the domain and range for an object property. Table 4-3 illustrates the object properties introduced in the SVN ontology.

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Range</th>
<th>Inverse Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasFile</td>
<td>FileRevision</td>
<td>File</td>
<td>isFileOf</td>
</tr>
<tr>
<td>hasMadeRelease</td>
<td>FileRevision</td>
<td>Release</td>
<td>isMadeupOf</td>
</tr>
<tr>
<td>hasRevision</td>
<td>File or Release</td>
<td>Revision</td>
<td>isRevisionOf</td>
</tr>
</tbody>
</table>

*Table 4-3: Object properties in initial SVN ontology design*
4.1.3. Enhanced SVN Ontology

As part of the ontological modeling approach, we further enriched and optimized our initial SVN ontology with additional constraints and relations in order to be able to take advantage of inference services provided by an ontological reasoner. The following enhancements to our initial SVN ontology were made: additional Object and Inverse Object Properties were introduced and we added new DL restrictions to existing concepts in order to allow us to take advantage of some reasoning services. Also, we added functional and transitive property types to the object properties and Inverse object properties.

Table 4-4 illustrates the additional object properties added in SVN ontology in order to take advantages of ontology reasoning services.

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
<th>Range</th>
<th>Inverse Property</th>
<th>Functional property</th>
<th>Transitive Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasLatestRevision</td>
<td>File</td>
<td>Revision</td>
<td>isLatestRevisionOf</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>hasNextRevision</td>
<td>Revision or File</td>
<td>Revision</td>
<td>hasPreviousRevision</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>hasFRevision</td>
<td>FileRevision</td>
<td>Revision</td>
<td>isFRRevisionof</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>hasPreviousRevision</td>
<td>File</td>
<td>Revision</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-4: Object properties added to enrich SVN ontology

Table 4-5 describes the main classes in SVN ontology and DL restriction applied on each class. The DL restrictions describe the relationships that must hold for members (individuals) of each class.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Restrictions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>hasLatestRevision some Revision, hasRevision some Revision.</td>
<td>Existential restriction on class File necessarily hasLatestRevision some Revision and hasRevision some Revision.</td>
</tr>
<tr>
<td>Release</td>
<td>isMadeUpOf some FileRevision.</td>
<td>Existential restriction on class Revision necessarily isMadeUpOf some FileRevision.</td>
</tr>
<tr>
<td>FileRevision</td>
<td>hasFile some File, hasFRevision some Revision.</td>
<td>Existential restriction on class FileRevision necessarily hasFile some File and hasFRevision some Revision.</td>
</tr>
</tbody>
</table>

Table 4-5: DL restrictions applied to classes

Figure 4-3 provides an overview of the refined SVN ontology model including classes, data type properties, object properties, inverse object properties and their relationships.

Figure 4-3: Overview of an enhanced SVN ontology
The DL restrictions on classes, the object and inverse object properties, and their types (i.e. functional property type and transitive property type) play a key role in reasoning, since some of the knowledge inference through the reasoner will be based on these object and inverse object properties. Our initial ontology design was almost one-to-one mapping with SVN repository schema. When we applied reasoning services to our initial ontology, the reasoner inferred no additional and/or interesting knowledge as shown in Figure 4-4.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revision16400</td>
<td>isRevisionOf</td>
<td>File192</td>
</tr>
<tr>
<td>Revision16601</td>
<td>isRevisionOf</td>
<td>File192</td>
</tr>
<tr>
<td>Revision16558</td>
<td>isRevisionOf</td>
<td>File192</td>
</tr>
<tr>
<td>Revision16117</td>
<td>isRevisionOf</td>
<td>File192</td>
</tr>
<tr>
<td>Revision15981</td>
<td>isRevisionOf</td>
<td>File192</td>
</tr>
<tr>
<td>File12</td>
<td>isFileOf</td>
<td>Revision15911</td>
</tr>
<tr>
<td>File14</td>
<td>isFileOf</td>
<td>Revision15911</td>
</tr>
<tr>
<td>File16</td>
<td>isFileOf</td>
<td>Revision15911</td>
</tr>
<tr>
<td>File17</td>
<td>isFileOf</td>
<td>Revision15911</td>
</tr>
<tr>
<td>File19</td>
<td>isFileOf</td>
<td>Revision15911</td>
</tr>
</tbody>
</table>

Figure 4-4: Reasoning services applied to initial SVN ontology model

After enriching and enhancing ontology models with new constraints, the reasoner inferred very useful knowledge like: links, relationship and dependencies of concepts as well as transitive relationships of the concepts. Figure 4-5 shows an example of inferred knowledge after enriching and enhancing initial SVN ontology model.
The SVN ontology also needs to define its own namespace. A distinct namespace is required in order to be able to uniquely identify the ontology. This mechanism is a main pillar of the Semantic Web. In order to have multiple ontologies defined within the same domain, a complete URL is used to specify the namespace. The namespace for our SVN ontology is http://aseg.cs.concordia.ca/svn
4.2. SVN-Ontologizer

The SVN-Ontologizer tool was developed to support:

1. The extraction of software version data from remote SVN repositories and
2. The automated ontology population of the extracted SVN data into a corresponding SVN ontology.

Figure 4-6 provides a general overview of the SVN-Ontologizer tool and the steps involved in the SVN extraction and ontology population process.

4.2.1. SVN Profile Setup

In order to establish a connection to a SVN repository, a user first has to set up a profile for the remote SVN repository (Figure 4-5). The profile includes the following information:
Repository location: The repository location that is specified using one of two protocols: “svn://, svn+ssh://” or “http://, https://”

User credentials: Required login and password information for the remote SVN repository server

Revision range: A specific range of revisions to be extracted (optional all revisions)

Version range: A specific range of versions to be extracted (optional all versions)

Ontology directory: The user can also specify ontology name and directory where the ontology is going to be created

---

Figure 4-7: SVN-Ontologizer main user interface
4.2.2. SVN Connection and Data Extraction

In the next step, access to the remote SVN repository is established by using the low level libraries provided by the SVNKit [SVK09]. The SVNKit provides an API to establish and manage remote access to a SVN repository. The SVN repository data can be accessed through the SVNKit using two different authentication protocols (shown in figure 4-8): (1) SVN specific protocols “svn/, svn+ssh” or (2) the standard “http and https” protocol [SV09A].

![Diagram](image)

**Figure 4-8: SVN connection process**

After a successful connection to the SVN server is established, the data extraction process for the *start revision* and *end revision* ranges specified in the profile is initiated. The raw data extracted from the SVN revision history consists of the following information:

- **revision:** Denotes a revision number of the committed data
**author:** Name of author or committer, who committed the revision

**date:** Revision date when it was committed or created

**log message:** Comments entered by author at the time of commit

**changed paths:** Includes information with respect to:

(a) The state of a file in the revision, denoted by the characters A, M or D, where “A” corresponds to Insertion, “M” to a Modification and “D” to a Deletion operation performed

(b) The full file path, which could be either a change path within same branch or copy path from different branch

The following is an example of raw data extracted from the ArgoUML [ARG09] SVN repository.

```
revision: 1014

author: shafique


log message: updated panel view and input view

changed paths:
M /trunk/src/UI/interface.java
M /trunk/src/UI/input.html
M /trunk/src/main/new/org/Status.java
A /trunk/src/main/new/org/broad.java
```

The extracted data is stored in two binary files. The revision file contains information related to each revision (i.e. **revision number, author, date and log message**). The path
file contains information related to the committed revision and actions performed on the file (i.e. *added, modified and deleted*), as well as the full path of the file in SVN repository.

### 4.2.3. Data Pre-processing

The extracted SVN data requires some pre-processing in order to support the automated population of the SVN ontology. The transformations are necessary to ensure that the extracted data can be represented in the OWL/RDF format.

**Serialization**

As part of the serialization process the following activities are performed:

- A unique identification is assigned to the paths associated with specific revisions.
- Revision numbers are serialized and duplicate entries of the same paths are eliminated.
- A memory model corresponding to the ontological representation is created to store serialized information.

**Elimination of invalid characters**

Some of the SVN data (in particular the SVN log messages) contains characters that are not supported by the OWL/RDF format; these invalid characters have to be removed. As part of the clean-up process, we replacing all invalid characters with characters supported by OWL/RDF format. Table 4-6 illustrates some of the substitutions that are performed as part of the data clean-up.
4.2.4. Ontology population

In the last step the pre-processed and normalized SVN data in the memory will be used to automatically populate our SVN ontology. In order to populate our SVN ontology, first we write SVN ontology TBox into the RDF/OWL file. In the second step we write ABox containing instances (i.e. loaded in memory models) as per TBox specifications in the form of RDF triples.

The RDF triples consist of two pieces of data that are linked by a named relationship. The RDF triple is a simple statement about the truth of some proposition. RDF distinguishes two kinds of elements that can appear in triples, literals and resources. A literal is a piece of data which can be an integer, a string, a floating-point number, or even an XML structure. A resource in RDF identifies something (or someone) about which we make semantically meaningful statements.

<table>
<thead>
<tr>
<th>Non-valid characters</th>
<th>Replaced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ü</td>
<td>U</td>
</tr>
<tr>
<td>Ý</td>
<td>Y</td>
</tr>
<tr>
<td>&lt;</td>
<td>-</td>
</tr>
<tr>
<td>&gt;</td>
<td>-</td>
</tr>
<tr>
<td>Ö</td>
<td>O</td>
</tr>
<tr>
<td>Å</td>
<td>a</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td>Î</td>
<td>I</td>
</tr>
<tr>
<td>Ñ</td>
<td>I</td>
</tr>
</tbody>
</table>

*Table 4-6: Substitution of invalid characters*
5. Bugzilla Ontology

5.1. Bugzilla Ontology Design

For the design of our Bugzilla ontology, again a 3 step ontology design approach was applied. First we analyzed the Bugzilla repository relational data schema in order to identify the major concepts stored in the repository and the relationships among them. Secondly we created an initial ontological model for the Bugzilla repository by mapping existing tables and relations found in the Bugzilla repository to their ontological equivalents. This mapping resulted in an almost one-to-one mapping between the relational Bugzilla schema and our initial Bugzilla ontology. In the last step we enriched the Bugzilla ontology with new constraints and relations in order to be able to take advantage of the ontological representation and reasoning services.

5.1.1. Bugzilla Repository Schema

Figure 5-1 provides an overview of the relational data schema extracted from the Bugzilla repository [BUG03]. In what follows we provide a more detailed description of the major entities and their relationships in the Bugzilla schema.

**Issue - Person (Many-to-Many Relationship)**

The issue entity has a relation with three types of persons (i.e. reporter, assignee, and cc person). The *reporter* is the person who submits the bug to the bug repository. The assignee is the person responsible for the submitted bug. It is however possible that an assignee also submits a bug, which makes him/her a reporter as well. A *cc person* is the
person to whom the bug was forwarded for resolution, review, or comment. The assignee, reporter and cc person types correspond to roles of the person, and can therefore be added to the entity person. The multiplicity of the relation between person and issue is many-to-many, with an issue having potentially multiple persons assigned to it. Likewise, a person can contribute to more than one issue.

**Person - Comment, Attachment and Activity Relation (Many-to-One Relationship)**

A person that contributes to a bug is a creator of a comment, attachment or an activity. The comment entity has a many-to-one relation with the person entity, since a person can write more than one comment, where as a comment can be written by only one person at a time. The same holds for the activity and attachment entities.

**Issue - Comment (One-to-Many Relationship)**

A comment entity contains information such as comment number, a time stamp, and the comment text. A comment provides additional information that is directly related to an issue. On the other hand, an issue can have multiple comments associated. The resulting relationship between issue and comment is therefore one-to-many.
Figure 5-1: Bugzilla repository schema

**Issue - Attachment (One-to-Many Relationship)**

An *attachment* (usually in the form of a file) provides additional information related to a particular issue. The attachment entity contains information such as the type of the attachment, the date of attachment, and a short description of the attached file. The
relation between an issue and attachment is one-to-many, where an issue can have more than one file attachment associated but an attachment can only be linked to a single issue.

**Issue - Activity (One-to-Many Relationship)**

Issues in Bugzilla are strictly bound to bug life cycle. As part of the bug life cycle, each reported issue is required to have an *activity* associated with it. An activity relates to changes that modify the status of an issue. The activity provides a detailed record of all the changes and contributions to an issue including comments added, status changes, etc. An activity therefore provides relevant information with respect to the history of an issue. The multiplicity of the relation between issue and activity is one-to-many. An issue can have multiple activities associated, whereas an instance of an activity can only be linked to a single issue.

**Issue - Dependency Relation**

Some *issues* can depend on or block one another. The dependency relation is normally two sided (i.e. “Depends on” and “Blocks”). In order to illustrate such an issue dependency, we consider three issues (issue1, issue2, and issue3) shown in Figure 5-2. In this scenario, issue1 depends on issue2 and issue3. In other words, the resolution of issue1 depends on the prior resolution of issue2 and issue3. On the other hand, in this scenario, issue3 blocks issue2 and issue1.
**Issue- Milestone Relation (One-to-Many Relationship)**

*Milestones* correspond to dates on which a developer plans to have a certain set of issue fixed. The multiplicity of the relation between issue and milestone is many-to-one, with a milestone typically involving more than one issue, whereas an issue has to be dealt with as part of a milestone.

**Issue-ComputerSystem Relation (Many-to-Many Relationship)**

The *ComputerSystem* contains hardware- and software-related information associated with a particular issue. The multiplicity of the relation is many-to-many; an issue can occur on different computer systems and a computer system can have different issues associated.

From the above schema description, one can identify that the issue entity plays a key role in the Bugzilla schema. A new instance of an issue is created each time, when a user reports a new bug or submits a feature request. The following information is always part of an issue to describe its details.
**Issue number:** A unique issue identification number.

**URL:** A location, where additional information about the bug can be found.

**Summary:** A short description of the issue.

**Priority:** Used by the assignee to prioritize the issue.

**Date opened:** Timestamp when the issue was submitted.

**Status:** Status of an issue in the bug lifecycle.

**Resolution:** Indicates what happened to particular issue in bug lifecycle.

### 5.1.2. Mapping Bugzilla Repository Schema to an Ontological Model

As part of our ontological model for the Bugzilla repository, we define an initial TBox by mapping the Bugzilla repository entities to an initial set of concepts in the Bugzilla ontology. Figure 5-3 shows the resulting Bugzilla ontology model, which consists of eight classes: issue, comments, activity, attachment, ComputerSystem, milestone, component and product. The *product* refers to a component as disused earlier, and *component* refers to a subsection of the product.
Table 5-1 describes the main classes and their usage.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>An Issue is an entity defining a certain topic concerning the development of a software system. An issue can be classified or discussed.</td>
</tr>
<tr>
<td>Activity</td>
<td>Activities form a certain kind of log, tracking the changes occurring to an Issue</td>
</tr>
<tr>
<td>Comment</td>
<td>A comment on a certain Issue</td>
</tr>
<tr>
<td>Attachment</td>
<td>Attachments are files sent in together with the Issue's text or a comment</td>
</tr>
<tr>
<td>Person</td>
<td>Person could be commenter, assignee of an issue, involved person in activity, cc person of an issue, reporter of an issue, an person attached file to issue</td>
</tr>
<tr>
<td>Component</td>
<td>A Component of the software system an Issue may refer to</td>
</tr>
<tr>
<td>Milestone</td>
<td>A Milestone refers to a planned version of a software system</td>
</tr>
<tr>
<td>Product</td>
<td>A Product is a functionally of a software system</td>
</tr>
<tr>
<td>Attachment</td>
<td>Attachments are files sent in together with the Issue's text or a comment</td>
</tr>
<tr>
<td>ComputerSystem</td>
<td>A ComputerSystem is the definition of an execution environment</td>
</tr>
</tbody>
</table>

*Table 5-1: Bugzilla ontology classes*
Table 5-2 provides a description of data-type properties modeled in the Bugzilla ontology, their type, and associated domain.

<table>
<thead>
<tr>
<th>Data Type Property</th>
<th>Type</th>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>what</td>
<td>String</td>
<td>Activity</td>
<td>Element affected in Activity</td>
</tr>
<tr>
<td>performed</td>
<td>dateTime</td>
<td>Activity</td>
<td>Date and time when this activity took place</td>
</tr>
<tr>
<td>removed</td>
<td>String</td>
<td>Activity</td>
<td>The part that was removed during this activity</td>
</tr>
<tr>
<td>added</td>
<td>String</td>
<td>Activity</td>
<td>The part that was added during this activity</td>
</tr>
<tr>
<td>fileName</td>
<td>String</td>
<td>Attachment</td>
<td>The filename of the attachment</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Attachment</td>
<td>The file type of this attachment. For example: gif, txt</td>
</tr>
<tr>
<td>text</td>
<td>String</td>
<td>Comment</td>
<td>Text part of comment</td>
</tr>
<tr>
<td>date</td>
<td>dateTime</td>
<td>Comment</td>
<td>The date when this comment was added</td>
</tr>
<tr>
<td>platform</td>
<td>String</td>
<td>ComputerSystem</td>
<td>A computer system's platform</td>
</tr>
<tr>
<td>OS</td>
<td>String</td>
<td>ComputerSystem</td>
<td>The operating system</td>
</tr>
<tr>
<td>status</td>
<td>String</td>
<td>Issue</td>
<td>The state of an issue</td>
</tr>
<tr>
<td>priority</td>
<td>String</td>
<td>Issue</td>
<td>The priority of the issues' fixing</td>
</tr>
<tr>
<td>dateOpened</td>
<td>dateTime</td>
<td>Issue</td>
<td>Date on which issue was reported</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>Issue</td>
<td>A description of the Issue</td>
</tr>
<tr>
<td>resolution</td>
<td>String</td>
<td>Issue</td>
<td>There can be different reasons why a bug is closed and therefore inactive</td>
</tr>
<tr>
<td>number</td>
<td>Integer</td>
<td>Issue, Comment</td>
<td>The number of an Issue an a comment describes its unique identifier</td>
</tr>
<tr>
<td>version</td>
<td>String</td>
<td>Product</td>
<td>Version of this Product</td>
</tr>
</tbody>
</table>

*Table 5-2: Data type properties in the Bugzilla ontology*
Table 5-3 describes the object properties modeled as part of the Bugzilla ontology. The description includes the concept, the supported domain and range.

<table>
<thead>
<tr>
<th>Object Property Name</th>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocks</td>
<td>Issue</td>
<td>Issue</td>
</tr>
<tr>
<td>hasActivity</td>
<td>Issue</td>
<td>Activity</td>
</tr>
<tr>
<td>hasAssignee</td>
<td>Issue</td>
<td>Person</td>
</tr>
<tr>
<td>hasAttachment</td>
<td>Issue</td>
<td>Attachment</td>
</tr>
<tr>
<td>hasCcPerson</td>
<td>Issue</td>
<td>Person</td>
</tr>
<tr>
<td>hasComment</td>
<td>Issue</td>
<td>Comment</td>
</tr>
<tr>
<td>hasCommentor</td>
<td>Comment</td>
<td>Person</td>
</tr>
<tr>
<td>hasComponent</td>
<td>Product</td>
<td>Component</td>
</tr>
<tr>
<td>hasComputerSystem</td>
<td>Issue</td>
<td>ComputerSystem</td>
</tr>
<tr>
<td>hasCreator</td>
<td>Attachment</td>
<td>Person</td>
</tr>
<tr>
<td></td>
<td>or Comment</td>
<td></td>
</tr>
<tr>
<td>hasInvolvedPerson</td>
<td>Activity</td>
<td>Person</td>
</tr>
<tr>
<td>hasMilestone</td>
<td>Issue</td>
<td>Milestone</td>
</tr>
<tr>
<td>hasReporter</td>
<td>Issue</td>
<td>Person</td>
</tr>
<tr>
<td>hasResolution</td>
<td>Issue</td>
<td>Revision (concept from Aligned SVN Ontology)</td>
</tr>
</tbody>
</table>

Table 5-3: Object properties of Bugzilla ontology

5.1.3. Refining the Bugzilla Ontology

In order to take advantage of the ontological model and reasoning services, the Bugzilla ontology structure was refined with additional constraints and relations. The following tables (Table 5-4 to 5-6) list some of the major modifications made to the initial Bugzilla ontology:
(1) An additional Inverse Object Properties were introduced.

(2) New DL restrictions to concepts in our initial design were introduced.

(3) We added a functional property and transitive property type to both the Object Properties and Inverse Object properties.

Table 5-4 describes the additional constraints added in order to enrich and enhance Bugzilla Ontology.

<table>
<thead>
<tr>
<th>Class</th>
<th>Disjoins with</th>
<th>DL Restrictions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>None</td>
<td>hasAssignee some Person hasComment some Comment hasReporter some Person</td>
<td>Existential restriction on class Issue necessarily hasAssignee some Person and hasComment some Person and hasComment some Comment.</td>
</tr>
<tr>
<td>Activity</td>
<td>Component, Product, Milestone, ComputerSystem, Attachment, Resolution, Comment</td>
<td>None</td>
<td>Disjoint classes</td>
</tr>
<tr>
<td>Comment</td>
<td>Component, Activity, Product, ComputerSystem, Milestone, Attachment, Resolution</td>
<td>hasCommentor some Person isCommentOf some Issue</td>
<td>Existential restriction on class Comment necessarily hasCommentor some Person and isCommentOf some Issue</td>
</tr>
<tr>
<td>Attachment</td>
<td>Component, ComputerSystem, Activity, Product, Milestone, Resolution, Comment</td>
<td>hasCreator some Person isAttachmentOf some Issue</td>
<td>Existential restriction on class Attachment necessarily hasCreator some Person and isAttachmentOf some Issue</td>
</tr>
<tr>
<td>Component</td>
<td>Activity, Product, Milestone, Attachment, ComputerSystem, Resolution, Comment</td>
<td>None</td>
<td>Disjoint classes</td>
</tr>
<tr>
<td>Milestone</td>
<td>Component, Activity, Product, ComputerSystem, Attachment, Resolution, Comment</td>
<td>None</td>
<td>Disjoint classes</td>
</tr>
<tr>
<td>Product</td>
<td>Component, Activity, Milestone, Attachment, Resolution</td>
<td>None</td>
<td>Disjoint classes</td>
</tr>
</tbody>
</table>

Table 5-4: Bugzilla ontology classes
Table 5-5 describes the inverse object properties introduced for the object properties and property types (i.e. functional and transitive property type) as part of the Bugzilla ontology. The description includes the concept, the supported range, and their inverse property. The object properties and their types (i.e. functional property type and transitive property type) play a key role in order to utilize reasoning services.

<table>
<thead>
<tr>
<th>Object Property Name</th>
<th>Property Type</th>
<th>Inverse Property</th>
<th>Property Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocks</td>
<td>Transitive Property</td>
<td>dependsOn</td>
<td>Functional Property</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Transitive Property</td>
</tr>
<tr>
<td>hasActivity</td>
<td>None</td>
<td>isActivityOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasAssignee</td>
<td>Functional Property</td>
<td>isAssigneeOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasAttachment</td>
<td>None</td>
<td>isAttachmentOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasCcPerson</td>
<td>None</td>
<td>isCcpersonOf</td>
<td></td>
</tr>
<tr>
<td>hasComment</td>
<td>Functional Property</td>
<td>isCommentOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasCommentor</td>
<td>Functional Property</td>
<td>isCommentorOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasComponent</td>
<td>Functional Property</td>
<td>isComponentOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasComputerSystem</td>
<td>None</td>
<td>isComputerSystemOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasCreator</td>
<td>Functional Property</td>
<td>isCreatorOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasInvolvedPerson</td>
<td>Functional Property</td>
<td>isInvolvedPerson</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasMilestone</td>
<td>Functional Property</td>
<td>isMilestoneOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasReporter</td>
<td>Functional Property</td>
<td>isReporterOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td>hasResolution</td>
<td>None</td>
<td>isResolutionOf</td>
<td>None</td>
</tr>
<tr>
<td>hasIssue</td>
<td>Functional Property</td>
<td>isIssueOf</td>
<td>Inverse Functional</td>
</tr>
<tr>
<td></td>
<td>Transitive Property</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5-5: Object properties of Bugzilla ontology
Figure 5-4 shows the example of reasoning services applied on initial Bugzilla ontology.

Figure 5-5 shows the knowledge inferred by the reasoner after enrichment and enhancement of an initial Bugzilla ontology design.

Figure 5-6 provides an overview of the refined Bugzilla ontology. In addition to classes, object and data type properties, the Bugzilla ontology needs to define its own namespace. A distinct namespace is required in order to be able to uniquely identify the ontology.
This mechanism is a main pillar of the Semantic Web. In order to have multiple ontologies defined in the same domain name, a whole URL can be used to define a namespace more specifically. Although a URL usually hosts a web page, this is not necessary for a namespace. The namespace for our Bugzilla ontology is http://aseg.cs.concordia.ca/bug. As part of the ontology population, we populated the ABox. In order to populate the Bugzilla ontology, we assert instances of concept and their roles.
Figure 5-6: Enriched Bugzilla ontology
5.2. Bugzilla-Ontologizer

In what follows, we discuss the initial Bugzilla-Ontologizer tool implementation in more detail. The Bugzilla-Ontologizer provides the follow functionalities:

1. Establishing a remote connection to a Bugzilla repository
2. Extracting and exporting raw data from a Bugzilla repository
3. Transforming the raw data and providing for an automated ontology population

The next sections will describe in more detail implementation details of the Bugzilla-Ontologizer tool.

5.2.1. Connection to Bugzilla and Data Extraction

Bugzilla provides a *Common Gateway Interface* (CGI) to access its components. The following components are accessible through the CGI interface:

- Administration of a Bugzilla Installation can be accessed through editcomponents.cgi, editgroups.cgi, editkeywords.cgi, editparams.cgi, editproducts.cgi, editusers.cgi, editversions.cgi, and sanitycheck.cgi.
- Creating, changing, and viewing bugs features can be accessed through enter_bug.cgi, post_bug.cgi, show_bug.cgi, and process_bug.cgi.
- Query.cgi / Buglist.cgi, searching for the bugs and viewing the bug list (i.e. query.cgi and buglist.cgi).
- Generating reports from the Bugzilla repository. (i.e. reports.cgi and duplicates.cgi.)
For the implementation of the Bugzilla-Ontologizer, the API provided by the CGI was used to establish the remote access to the Bugzilla repository. In particular, the following CGI components were used for the Bugzilla-Ontologizer tool implementation (Figure 5-7):

- The *urlbase* Java utility, which uses as a parameter the fully qualified domain name of the web server path that hosts the Bugzilla installation. Also, *show_bug.cgi* was used to search through the HTML file (provided by the Bugzilla remote installation) to find the bug identification number associated with a bug.

- The *buglist.cgi* component is used to extract the bug list based on a string matching query. Component returns an XML file containing the matches.

- Bug details are accessed through the *XML.cgi* component.

*Figure 5-7: Bug data extraction*
5.2.2. Data Pre-processing

In what follows, we describe some pre-processing steps that are necessary in order to transform the raw data into a format suitable for automated ontology population. Figure 5-8 provides a general overview of the transformation process and the various steps involved.

**Serializing**

The first pre-processing steps involve the serialization of the exported Bugzilla raw data. The processing performed as part of this step includes:

- Assign a unique identification numbers to both, issues and related entities (i.e. comments, activities and attachments).
- Create an in memory representation of the data which corresponds close to the ontological model.

![Diagram of pre-processing steps](image)

*Figure 5-8: Overview of pre-processing steps*
Elimination of invalid characters

In this pre-processing step a clean-up of the data is performed. In many cases the issue, activity, and text description of comments contain characters that are not supported by the OWL/RDF format. As part of the clean-up step, we replace invalid characters with characters supported by the OWL/RDF format. Table 5-6 provides some examples of the removal of non-valid characters and their replacements with valid ones.

<table>
<thead>
<tr>
<th>Non-valid characters</th>
<th>Replaced with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ū</td>
<td>U</td>
</tr>
<tr>
<td>&lt;</td>
<td>-</td>
</tr>
<tr>
<td>&gt;</td>
<td>-</td>
</tr>
<tr>
<td>Ö</td>
<td>O</td>
</tr>
<tr>
<td>Æ</td>
<td>a</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
</tr>
</tbody>
</table>

Table 5-6: Substitution of invalid characters

In the last step, the pre-processed and normalized SVN data in the memory will be used to automatically populate our SVN ontology. In order to populate our SVN ontology, first we write SVN ontology TBox into the RDF/OWL file. In the second step we write ABox containing instances (i.e. loaded in memory models) as per TBox specifications in the form of RDF triples.

5.2.3. Bugzilla-Ontologizer User Interface

Common to most bug tracking systems is the provision of a web-based query interface. Figure 5-9 provides an example of the web-based query interface associated with the ArgoUML [ARG09] Bugzilla issue tracking system. In order to extract bugs from the underlying Bugzilla repository, users can specify various properties in order to filter and select the scope of the bug information to be extracted from the repository.
Figure 5-9: ArgoUML bug tracking system query interface

In order to reduce the need of context switching among interfaces, we adopted a similar GUI as the one implemented in Bugzilla, for our Bugzilla-Ontologizer tool (Figure 5-10). The user interface is divided into three main parts: query, remote directory, and ontology destination directory panel.
**Figure 5-10: Bugzilla-Ontologizer user interface**

The query panel allows users to specify constraints in order to restrict the scope of the queries by filtering specific data. Among the supported filters are:

**Issue type:** Defines the type of issue the user wants to extract. Supported values are: DEFECT, ENHANCEMENT, FEATURE, TASK, and PATCH.

**Component:** Defines the product of a software project. The products are the broadest category in Bugzilla and tend to represent real-world
shipping products. For example, if a company makes computer games it should have one product per game, perhaps a "Common" product for units of technology used in multiple games, and maybe a few special products (Website, Administration, etc.).

**Subcomponent:** Defines the subsections of a component (product). For example, a company designing computer games may have a "UI" subcomponent, an "API" subcomponent, a "Sound System" subcomponent, and a "Plug-in" subcomponent, each overseen by a different programmer. It often makes sense to divide subcomponents in Bugzilla according to the natural divisions of responsibility within the component.

**Status:** Defines the status of an issue in the bug lifecycle. Supported values are: UNCONFIRMED, NEW, STRTED, REOPEND, RESOLVED, VARIFIED and CLOSED.

**Resolution:** Indicates that, what happened to a particular issue in the bug lifecycle. Supported values are: FIXED, WONTFIX, LATER, INVALID, REMIND and DUPLICATE.

**Priority:** Describes the importance and order in which a bug should be fixed. The priority field is used by the developers to prioritize their work. Supported values are: P1, P2, P3, P4, and P5 where P1 indicates the most important issue and P5 indicates the least important issue.
Platform: Defines the hardware platform context in which the bug occurred. Supported values are: All, Macintosh, PC, Sun and HP.


Version: Defines the release in which an issue or defect was found.

Target Milestone: Defines the project designated milestones, this field can also be used to associate issues with those milestones, such as version and releases.

Based on user-specified filters, a SQL query will be executed to extract the information from the remote Bugzilla repository. The remote repository location itself is specified by the user within the remote directory panel. The query panel allows users to specify the output directory and name of the Bugzilla ontology.
6. Initial Experimental Evaluation and Ontological Queries

6.1. Case Study

We have selected ArgoUML 0.28 release [ARG09] as a case study. ArgoUML is a medium size open source UML modeling tool and includes support for all standard UML 1.4 diagrams. It runs on any Java platform and is available in ten languages. ArgoUML 0.26 and 0.26.2 have been downloaded over 80,000 times and are in use all over the world. Table 6-1 shows some statistics of ArgoUML 0.28 release.

| Total number of attributes | 2460 |
| Total number of classes    | 18333 |
| Total number of methods    | 14059 |
| Total number of packages   | 144   |
| Total number of interfaces | 526   |
| Total number of static methods | 744   |
| Total number of static attributes | 1922 |
| Total number of line of code | 168516 |

Table 6-1: Some statistics about ArgoUML 0.28

Table 6-2 shows some statistics of an ArgoUML project retrieved from SVN ontology.

| Total number of concepts | 4 |
| Total number of Object properties | 12 |
| Total number of Data type properties | 9 |
| Total number of Instances | 195591 |
| Total number of Instances (reduced version) | 103343 |
| Total number of files used in different revision | 56920 |
| Total number of revisions | 16793 |
| Total number of Authors (developers/maintainers) | 50 |

Table 6-2: Statistics of an ArgoUML project from SVN ontology
Table 6-3 shows some statistics from an ArgoUML project retrieved from Bugzilla ontology.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of concepts</td>
<td>9</td>
</tr>
<tr>
<td>Total number of Object properties</td>
<td>28</td>
</tr>
<tr>
<td>Total number of Data type properties</td>
<td>25</td>
</tr>
<tr>
<td>Total number of Instances</td>
<td>871</td>
</tr>
<tr>
<td>Extracted releases</td>
<td>8</td>
</tr>
<tr>
<td>Total number of Person</td>
<td>27</td>
</tr>
<tr>
<td>Total number Issues (as of February 4, 2009)</td>
<td>71</td>
</tr>
<tr>
<td>Total number Activities of related to an Issue</td>
<td>33</td>
</tr>
<tr>
<td>Total number of Comments on Issues</td>
<td>366</td>
</tr>
<tr>
<td>Total number of computer systems</td>
<td>10</td>
</tr>
</tbody>
</table>

*Table 6- 3: Statistics of an ArgoUML project from Bugzilla ontology*

6.2. Ontological queries applied on SVN Ontology.

In this section, we present several SPARQL queries in order to illustrate information retrieval through the SVN ontology. The following queries discussed in more detail are applied to an SVN ontology that was populated with the SVN data extracted from the ArgoUML [ARG09] project. We will first identify the contribution of a developer/maintainer to the overall project. Next, we discuss the identification of releases and their commit dates. Finally, we treat the extraction of revisions and their associated files' information.
Contribution of a particular developer / maintainer to the overall project

In order to retrieve information about the contribution of a particular developer/maintainer, we defined the SPARQL query shown in Figure 6-1. The query identifies the overall contribution of a maintainer towards the project. We retrieve the number of revisions created by a specific author, in this case “bobtarling.” The SPARQL query returns the total number of commits performed by “bobtarling.” We can see that this author has made 2003 commits.

Figure 6-1: Results of SPARQL query

For the next query, we extend the query as shown in Figure 6-2 in order to provide some additional insights regarding a developer's contribution towards each revision. In this query we have retrieved three pieces of information: the revision numbers of the revisions created by the specific author, the files associated with each revision, and the action performed on the file during each revision (i.e. added “A”, modified “M” and deleted “D”). Figure 6-2 shows the results of this extended query. As a result, the query establishes a link between an author, revisions, files and actions performed by the author.
The information retrieved through the SPARQL query is useful to evaluate the developer’s contribution to a software project.

**Releases and their commit dates**

Release dates are stored within the SVN repository by creating release tags while committing a new revision. However, in SVN there is standard way of recording release information in the SVN repository. For example, releases in the ArgoUML SVN repository are stored as different branches. The branches are directories with no further information regarding the commits’ history. During SVN data extraction and pre-processing, we extracted these release tags from the commit information and stored then within the SVN ontology. In order to retrieve the releases and their creation date, we applied the query shown in Figure 6-3. As a result, the query returns releases and their creation dates as shown in Figure 6-3.
In the following example, we retrieve all files associated with a particular revision. The query in Figure 6-4 returns the following information:

- A specific revision and its associated files (i.e. the files, which are modified to create a specific revision)
- The latest revision of each file.
- The links of a file with other revisions, in this case instances of OtherRevision are inferred by the Pellet reasoner
- Furthermore, the query establishes a link between the revisions and the files.
6.3. Ontological Queries Applied to Bugzilla Ontology

In the following examples, we apply SPARQL queries to extract information from the populated Bugzilla ontology. Among the queries we discuss in more detail are queries that identify the contribution of a specific programmer towards the Bugzilla repository, provide some general Bugzilla repository statistics, and illustrate the use of knowledge inference through reasoning services.

Identifying the Contribution of a Particular Person

In order to retrieve information about the contribution of a particular developer/maintainer, we defined the query as shown in Figure 6-5. In this query, we identify the contribution of a particular maintainer within the Bugzilla repository. The query retrieves all the assigned issues and the activities in which a specific person is/was
involved. The query results can be used not only to identify the most active project members but can also analyze who worked on which issue in the past, etc.

\[
\text{select} \ ?\text{Person} \\
\ ?\text{AssigneeOf} \\
?\text{involved_In} \\
\text{where} \{ \\
\text{FILTER} \ (\text{regex}(\text{str}(?\text{Person}), \ \text{"linus"})). \\
?\text{Person} \text{ rdf:type} :\text{Person}. \\
?\text{Person} \ :\text{isAssigneeOf} \ ?\text{AssigneeOf}. \\
?\text{Person} \ :\text{involvedPerson} \ ?\text{involved_In}. \\
\}\n\]

Where:
\text{IsAssigneeOf} \text{ and } \text{isInvolvedPerson} \text{ are the Object Properties.}
\text{A "linus" is the person name.}

<table>
<thead>
<tr>
<th>Person</th>
<th>AssigneeOf</th>
<th>Involved_In</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person_linus</td>
<td>issue3059</td>
<td>issue3059activity1</td>
</tr>
<tr>
<td>Person_linus</td>
<td>issue4993</td>
<td>issue3059activity1</td>
</tr>
<tr>
<td>Person_linus</td>
<td>issue4668</td>
<td>issue3059activity1</td>
</tr>
<tr>
<td>Person_linus</td>
<td>issue5553</td>
<td>issue3059activity1</td>
</tr>
<tr>
<td>Person_linus</td>
<td>issue3059</td>
<td>issue3059activity2</td>
</tr>
<tr>
<td>Person_linus</td>
<td>issue4993</td>
<td>issue3059activity2</td>
</tr>
<tr>
<td>Person_linus</td>
<td>issue4668</td>
<td>issue3059activity2</td>
</tr>
</tbody>
</table>

Result of a query:
Person name, issue assigned to him/her.
Activities in which he/she involved.

Figure 6- 5: Query results showing the contribution of a specified person

**Inference knowledge by reasoning services**

In addition to information retrieval through the SPARQL queries, reasoning services provided by ontological reasoners, such as Pellet in our case, can be used to infer additional knowledge. In what follows we illustrate the use of inference services to infer missing knowledge that was not available at the time of ontology population. For example, at the time of the Bugzilla ontology population, we were only able to assert the instances of the object property DependsOn but did not populate the inverse property Blocks.
Figure 6-6: Reasoning example

Figure 6-6 illustrates an example of such a SPARQL query that takes advantage of Pellet's reasoning services. In this example, we consider four issues: issue4168, issue4766, issue5490, and issue5548. From the asserted knowledge, one can identify that issue4168 depends on issue4766, issue4766 depends on issue5490 and an issue5490 depends on issue5548. Through reasoning, we can also infer that, issue5548 is blocking issue5490, and issue5490 is blocking issue4766 (Figure 6-7).

Furthermore, the reasoning services also allow us to resolve the transitive closure between the issues (Figure 6-8). In this case the reasoner infers that issue4168 depends also on issue5490 and on issue5548. Furthermore, issue5548 blocks both issue4766 and issue4168. Figure 6-8 illustrates this example that takes advantage of both asserted and inferred knowledge. The results are displayed in Figure 6-9.
Figure 6-8: Example of inferred transitive closure
6.4. Linked SVN and Bugzilla Ontology Queries

In order to allow for queries to work across ontologies, our SVN and Bugzilla ontologies have to be linked.

Linking SVN and Bugzilla Ontologies

An interconnection between SVN and Bugzilla ontology is created through the entities sharing a common concept. The revision committed to an SVN repository may be referenced by its issue number. On the other hand, an issue reported in Bugzilla repository may be referenced by a revision number.

We linked our two ontologies through common shared instances with the help of the isResolutionOf and hasResolution object properties associations. As shown in Figure 6-
10, issue and revision concepts are linked through object properties isResolutionOf and hasResolution. The linking of the ontologies is bi-directional. The object property hasResolution contain revision numbers corresponding to the issue’s resolution history in SVN ontology. The object property isResolutionOf contains an instance issue number, which refers the particular revision to an issue in the Bugzilla repository. During the SVN and Bugzilla ontology population phase we extracted revision numbers from issue comments and issue numbers from revision commit messages and stored them as instances of isResolutionOf and hasResolution object properties. Figure 6-10 shows the linked Bugzilla and SVN ontologies.

**Evaluation of Links among SVN and Bugzilla Ontologies**

in order to validate our approach of linking SVN and Bugzilla ontologies, we applied a SPARQL query. The query searches for all the issues that have an instance of hasResolution in the Bugzilla ontology. After retrieving the instances (i.e. the revision number related to an issue), the query retrieves the information related to the revision number from the linked SVN ontology (i.e. commit date). Table 6-4 shows the evaluation of the SVN ontology (i.e. revision number) links found in the Bugzilla ontology.

| Total number of releases in Bugzilla Ontology (including Alpha-X and Beta X) | 8 |
| Total number of Bugs | 71 |
| Total Number of links to SVN repository | 37 |
| Total number of invalid links | 2 |

*Table 6-4: Evaluation of links found in the Bugzilla ontology*
In what follows, we present several SPARQL queries, which are applied to the linked SVN and Bugzilla ontology. Among the queries we discuss in more detail are the time spent to resolve an issue, the resolution history related to a particular issue, and the analysis of transitive relationships.

**Time Required in order to Resolving a Reported Issue**

In order to retrieve information related to the time required to resolve a reported issue, we define the query as shown in Figure 6-11, which retrieves the date and time an issue was first reported and the date and time of the commit corresponding to the resolution of the
same issue. The query retrieves information in two steps. First, the query searches for all the issues that have an instance of hasResolution in Bugzilla ontology. Then, after retrieving the instances (i.e. the revision number related to an issue), the query retrieves the related information to the revision number from the linked SVN ontology (i.e. commit date). Figure 6-11 shows the query and results obtained from this query.

![Query Editor](image)

**Information from Bugzilla ontology**

- issue5042 2008-04-29T06:06:05
- issue5256 2008-07-22T01:47:47
- issue5258 2008-07-22T11:19:18
- issue5260 2008-07-23T01:56:35
- issue5478 2008-10-29T06:19:46
- issue5482 2008-11-07T03:01:56
- issue5493 2008-11-07T04:00:56
- issue5497 2008-11-08T01:39:30
- issue5542 2008-12-04T03:28:42
- issue5548 2008-12-05T05:54:55
- issue5557 2008-12-11T10:05:31
- issue5581 2009-01-03T09:54:15
- issue5598 2009-01-06T12:05:32
- issue5602 2009-01-07T11:26:00

**Information from SVN Ontology**

- svn:Revision15911 2008-10-09T01:51:16
- svn:Revision15897 2008-10-07T04:44:59
- svn:Revision15814 2008-09-28T03:46:37
- svn:Revision15815 2008-09-28T03:51:54
- svn:Revision15911 2008-10-09T01:51:16
- svn:Revision15956 2008-11-03T06:39:05
- svn:Revision15972 2008-11-07T03:48:46
- svn:Revision15969 2008-11-06T02:18:49
- svn:Revision16135 2008-11-27T06:54:51
- svn:Revision16259 2008-12-08T01:59:23
- svn:Revision16270 2008-12-07T08:59:06
- svn:Revision291 2000-03-25T06:47:37
- svn:Revision16493 2009-01-03T02:51:05
- svn:Revision16541 2009-01-08T02:46:08
- svn:Revision16539 2009-01-08T02:31:25

*Figure 6-11: Date/time between issue reporting and resolution*

**Resolution history related to an issue**

Figure 6-12 illustrates an example of how to mine the linked Bugzilla and SVN ontologies in order to retrieve information related to the resolution of a reported issue. In our approach, the resolution history can be retrieved by applying the query across the linked Bugzilla and SVN ontologies.
Figure 6-12: Bug resolution history in SVN ontology

Figure 6-13 illustrates a detailed query and the results obtained from this query. After retrieving the revision related to a particular issue (through the Has Resolution property) the following additional revision information can be retrieved: the files modified in specific revision to resolve an issue, the full path of a specific file in SVN repository, and a traceability link between the issue and its related resolution information stored in the SVN ontology.
Transitive relationships between entities

The query example in Figure 6-14 illustrates a case where inferred knowledge from the reasoner is combined with transitive relationships of entities to mine the two sub-ontologies.

Figure 6-14: Transitive relationships.
We note that revisions associated with a particular issue are identified. Also, files modified as part of a particular revision are retrieved. In addition, all previous revisions of a particular file are extracted using inferred knowledge (Figure 6-15).

<table>
<thead>
<tr>
<th>Query Editor</th>
<th>Query Library</th>
</tr>
</thead>
</table>

Figure 6-15: Query results showing inferred knowledge

6.5. Discussion

The presented case study and the results obtained through queries illustrate the applicability of our approach in order to support various aspects of software evolution. The approach presented here is implemented as a part of the SE-Advisor framework. Section 6.5.1 introduces SE-Advisor framework functionalities and architecture.

6.5.1. SE-Advisor Framework

A common ontological representation (i.e. SVN and Bugzilla ontologies) and automated tools (i.e. SVN and Bugzilla Ontologizer tools) are designed and implemented as a part of SE-Advisor framework.

SE-Advisor provides a pro-active, ambient, knowledge-based environment that integrates users, tasks, tools and resources, as well as processes and history-specific information.
SE/Process Advisor provides an ambient semantic software maintenance environment. Its goal is to support developers throughout maintenance tasks by providing a context-sensitive knowledge base that can be queried either directly by a user or indirectly through supporting tools. The SE-Advisor framework supports maintainers by managing two knowledge-intensive aspects of the software evolution:

1. Collecting and maintaining semantic links, i.e., traceability links, between software artifacts, in particular those at different abstraction levels like source code and its associated documentation
2. Maintaining knowledge about available tools, software evolution processes, users, and their history of solving tasks with the available artifacts, to provide contextual guidance during complex maintenance tasks

SE-Advisor integrates available knowledge resources such as emails, wikis, bug trackers, source code, etc. This information is further automatically and/or semi-automatically analyzed and linked. The process of building knowledge repository ontology is also called ontologizing. Gathered information is presented to a maintainer in the form of a context-sensitive advisor tool which provides the ability to look beyond document boundaries while working on a process. Being aware of the process definition also allows one to guide users through a process. In a feedback loop, newly gained knowledge resources are used to constantly enrich the ontology. Figure 6-16 provides an overview of SE-Advisor framework.
Figure 6-16: Overview of SE-Advisor framework
7. Related Work and Limitations

In this section we will discuss and compare our work with existing approaches introduced by several researchers, which are closely related to our approach. Later we will discuss the limitations and challenges of our approach.

7.1. Related Work

The work most related to ours is by Kiefer et al [KAI07]. It also provided the foundation for the ontological models we used in this thesis. They introduced the iSPARQL query engine which is based on the SPARQL query language. They conducted four sets of experiments. The first was the measurement of code evolution code by visualizing changes between different releases. Secondly, they conducted refactoring experiments by the evaluation of the applicability of the iSPARQL framework to detect bad code smells. Their third experiment was a metrics experiment, performed by the evaluation of the ability to calculate software design metrics. Fourth and finally was their use of ontological reasoning as part of their software ontology models. However, the main focus of their work is on the source code model. In our approach, we enhanced and enriched the ontologies introduced by [KAI07] with two additional concepts: object and data type properties. We also applied DL restriction to our concepts to take advantage of reasoning services. Furthermore, we introduced the SVN and Bugzilla-Ontologizer tools to automate the process of extraction and ontology population.

Happel et al. [HAP06] presented their KOntoR project in which they focus on storing and querying meta-data about software artifacts to foster software reuse. The software
components are stored in a repository and they present various ontologies for providing background knowledge about the components, such as the programming language and licensing models. Compared to our approach, their focus was mainly on conceptualization of the software domain, rather than on the analysis of specific artifacts, as in our case.

Antoniol et al. [GIA04] proposed a multi-level concept navigation framework that represents source code entities using the FAMIX meta-model compliant Rigi Standard Format (RSF). In their approach, the release history information from Release History and Bug Databases (RHDBs) were extracted using a set of different tools. The extracted information was stored as RSF files for further processing and analysis. Compared to the approach by [GIA04], we use an OWL/RDF format in order to provide a uniform and semantic rich ontological representation that allows us to take advantage of inference services provided by ontological reasoners. Another approach, presented by D'Ambros et al. in [IMB06], introduced a visualization technique to uncover the relationships between data from a versioning and bug tracking. In their approach they use a version of the Release History Database.

German [DMG04] proposed a tool called softChange. The main function of softChange is the extraction, enhancement and visualization of software repositories (i.e. CVS). SoftChange consist of three different sub tools: (1) the trails extractor, for retrieving the raw software trails from the CVS repositories. The extracted data is stored within a relation database. (2) The fact enhancer analyzes the raw data in the database in order to generate the new facts. (3) The visualize tool provides a visual representation of the extracted facts.
Main differences between the approach in [DMG04] and ours is that it focuses only on the analysis and visualization of CVS repositories. In our approach we are not limited to analyzing CVS but also include the bug tracking systems. Furthermore, within our approach we also promote the integration of various artifacts and cross-artifact analysis.

Rysselberghe [VR04S] introduced another visualization approach to visualize the changed frequency of files, using different charts.

Hyland-Wood [HLW06] introduced an ontology model for software code based on Java called SEC. SEC allows the recording and tracking of changes made to metadata. Our approach is similar to [HLW06] in the sense of an ontological format representation. However, SEC does not include the information from a versioning or bug tracking system. Our presented approach allows integration of the information from the versioning system and the bug tracking system and uses inference services across sub-ontologies.

Other research in mining software repositories (i.e. [AEH06, JIM07, GCL05]) have also been focused on various types of analysis like impact analysis, traceability links, or guiding software development process. Our approach not only supports similar types of analysis, but also promotes the use of a common, semantic-rich ontological representation which allows for analysis across multiple repositories.
7.2. Limitations

During the evaluation phase, we found the following limitations of our presented approach.

Scalability

In order to evaluate our proposed approach with respect to reasoning services across multiple ontologies (i.e. SVN and Bugzilla Ontology), we used Protégé 3.4 ontology editor which provides a plug-in for the Pellet reasoner and SPARQL query editor. During the reasoning process, we experienced the memory overflow errors due to the size of our ontology. The initial size of the SVN ontology was 44 megabytes, causing memory overflow errors. We reduced our initial ontology size to 22 megabytes and were able to apply reasoning services and to apply SPARQL queries. The reason behind the memory overflow error was the consumption of memory by both the reasoner (Pellet Reasoner) and the ontology editor (Protégé 3.4). Furthermore, the Protégé 3.4 and Pellet Reasoner use Java virtual machine in the background. During the reasoning process, these tools generate in-memory models too complex and too large to process ontologies in order to apply reasoning services. Another limitation is of the Java virtual machine, which only supports memory size up to 1 gigabyte.

Bug extraction

In order to extract all the bug information, we found the limitation of remote Bugzilla installation which does not allow for extraction of all the bug information at once. Due to this limitation we were only able to extract bug information of eight releases. This
limitation meant that linking SVN and Bugzilla ontology (i.e. instances of hasResolution object property) is one-sided, from the Bugzilla ontology to the SVN ontology. In order to recover links from SVN to Bugzilla (i.e. instances isResolutionOf object property), we needed all of the issues to be stored in Bugzilla ontology.

**Persistent Storage**

Currently, our SVN and Bugzilla ontology is stored as plain RDF/XML format which has size (these are very large ontologies with many instances), performance, and management issues. In order to deal with size, performance, and management issues, there is a need to use database technology in order to provide persistence to the knowledge described by the ontologies, and scalability to the queries and reasoning on this knowledge.

**Consistency (i.e. incremental updates)**

One of the challenges and potential future work of this project is to manage incremental updates to software repositories such as SVN and Bugzilla ontology. Currently SVN and Bugzilla Ontologizer tools do not support incremental updates to SVN and Bugzilla ontologies.
8. Conclusion and Future Work

In this thesis, we discussed the importance of software repositories in supporting the evolution of software systems. We also discussed some of the challenges associated with extracting and modeling the information extracted from the software repositories.

In order to model the extracted information, we introduced a common ontological representation (based on the OWL/RDF format) to store the information extracted from SVN and Bugzilla repositories. In order to support the extraction process, we implemented two tools (SVN and Bugzilla-Ontologizer) to automate the data extraction and ontology population process. The two tools support the establishment of a connection between the Eclipse IDE and the software repositories, the extraction of raw data from the software repositories (namely SVN and Bugzilla), and the transformation and normalization of the extracted raw data in order to support automated ontology population. The approach we have presented is implemented as a part of our SE-Advisor framework. We presented a case study to evaluate our ontological model and its ability to mine and analyze data from these repositories to support the evolution of a software system. The case study was performed on ArgoUML [ARG09]. We used SPARQL queries to demonstrate how our ontological representation can support software evolution by mining and analyzing software repositories. The evaluation (case study) also shows the applicability of our presented approach as a part of SE-Advisor framework. SE-Advisor supports various aspects of the software evolution; our contribution of a common semantic rich ontological representation fulfills the key requirement of SE-Advisor
framework. Furthermore, through the ontological queries, we also were able to illustrate some of the benefits of using an ontological reasoner.

As part of the future work, the SVN Ontology should be extended to include additional entities to support the modeling of file content differences (i.e. the difference of the file contents modified in each revision).

Furthermore, there is a need for additional analysis and evaluation of our approach. Also, additional data mining techniques can be applied to provide further insights and analysis of these repositories.
References


livadas, p. e., small, d. t., “understanding code containing preprocessor constructs”, in proceedings of the 3rd workshop on program comprehension, ieee computer society press, los alamitos, ca, 1994, 89-97.


mcguinness, d. l., fikes, r., rice, j. and wilder, s., “an environment for merging and testing large ontologies”, principles of knowledge representation and reasoning, in proceedings of the seventh international conference (kr2000).


muller, h., “understanding software systems using reverse engineering technology”, in proceedings of colloquium on object orientation in databases and software engineering; the 62nd congress of l’association canadienne francaise pour l’avancement des sciences (acfa’94), montréal.


Appendices

Appendix-I

Following Table (AP1) provides the details of the removal of non-valid characters and their replacements with valid ones in SVN and Bugzilla Ontology.

<table>
<thead>
<tr>
<th>Non-valid character</th>
<th>Replacing character</th>
<th>Non-valid character</th>
<th>Replacing character</th>
<th>Non-valid character</th>
<th>Replacing character</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td></td>
<td>&quot;</td>
<td>&quot;</td>
<td>Ò</td>
<td>o</td>
</tr>
<tr>
<td>!</td>
<td>Space</td>
<td>.</td>
<td>&quot;</td>
<td>Ò</td>
<td>o</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td>»</td>
<td>»</td>
<td>Ò</td>
<td>o</td>
</tr>
<tr>
<td>%</td>
<td>Percent</td>
<td>A</td>
<td>A</td>
<td>Ò</td>
<td>o</td>
</tr>
<tr>
<td>(</td>
<td>-</td>
<td>A</td>
<td>A</td>
<td>Ù</td>
<td>u</td>
</tr>
<tr>
<td>)</td>
<td>-</td>
<td>A</td>
<td>A</td>
<td>Ù</td>
<td>u</td>
</tr>
<tr>
<td>*</td>
<td>-</td>
<td>A</td>
<td>A</td>
<td>Ù</td>
<td>u</td>
</tr>
<tr>
<td>&lt;</td>
<td>-</td>
<td>A</td>
<td>A</td>
<td>Ù</td>
<td>u</td>
</tr>
<tr>
<td>&gt;</td>
<td>-</td>
<td>Ç</td>
<td>Ç</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>@</td>
<td>-</td>
<td>E</td>
<td>E</td>
<td>&lt;</td>
<td>-</td>
</tr>
<tr>
<td>^</td>
<td>-</td>
<td>É</td>
<td>É</td>
<td>&gt;</td>
<td>-</td>
</tr>
<tr>
<td>.</td>
<td>Space</td>
<td>É</td>
<td>É</td>
<td>Ò</td>
<td>O</td>
</tr>
<tr>
<td>{</td>
<td>-</td>
<td>É</td>
<td>É</td>
<td>Æ</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>&amp;</td>
</tr>
<tr>
<td>}</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>i</td>
<td>-</td>
</tr>
<tr>
<td>~</td>
<td>Space</td>
<td>i</td>
<td>i</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>☐</td>
<td>Space</td>
<td>N</td>
<td>N</td>
<td>Ù</td>
<td>Ù</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>Ò</td>
<td>Ò</td>
<td>Ù</td>
<td>Ù</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>Ò</td>
<td>Ò</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>Space</td>
<td>Ò</td>
<td>Ò</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Space</td>
<td>Ò</td>
<td>Ò</td>
<td></td>
<td></td>
</tr>
<tr>
<td>†</td>
<td>+</td>
<td>Ò</td>
<td>Ò</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>-</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>Space</td>
<td>Ù</td>
<td>Ù</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>S</td>
<td>Ù</td>
<td>Ù</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-valid character</td>
<td>Replacing character</td>
<td>Non-valid character</td>
<td>Replacing character</td>
<td>Non-valid character</td>
<td>Replacing character</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>---------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>‘</td>
<td>Space</td>
<td>‘</td>
<td>Space</td>
<td>‘</td>
<td>Space</td>
</tr>
<tr>
<td>‘</td>
<td>Y</td>
<td>‘</td>
<td>Y</td>
<td>‘</td>
<td>Y</td>
</tr>
<tr>
<td>‘</td>
<td>B</td>
<td>‘</td>
<td>B</td>
<td>‘</td>
<td>B</td>
</tr>
<tr>
<td>‘</td>
<td>A</td>
<td>‘</td>
<td>A</td>
<td>‘</td>
<td>A</td>
</tr>
<tr>
<td>‘</td>
<td>A</td>
<td>‘</td>
<td>A</td>
<td>‘</td>
<td>A</td>
</tr>
<tr>
<td>£</td>
<td>S</td>
<td>£</td>
<td>S</td>
<td>£</td>
<td>S</td>
</tr>
<tr>
<td>£</td>
<td>Å</td>
<td>£</td>
<td>Å</td>
<td>£</td>
<td>Å</td>
</tr>
<tr>
<td>£</td>
<td>Å</td>
<td>£</td>
<td>Å</td>
<td>£</td>
<td>Å</td>
</tr>
<tr>
<td>¥</td>
<td>Y</td>
<td>¥</td>
<td>Y</td>
<td>¥</td>
<td>Y</td>
</tr>
<tr>
<td>¥</td>
<td>Å</td>
<td>¥</td>
<td>Å</td>
<td>¥</td>
<td>Å</td>
</tr>
<tr>
<td>¥</td>
<td>Å</td>
<td>¥</td>
<td>Å</td>
<td>¥</td>
<td>Å</td>
</tr>
<tr>
<td>©</td>
<td>Copy-rights</td>
<td>©</td>
<td>Copy-rights</td>
<td>©</td>
<td>Copy-rights</td>
</tr>
<tr>
<td>©</td>
<td>i</td>
<td>©</td>
<td>i</td>
<td>©</td>
<td>i</td>
</tr>
<tr>
<td>©</td>
<td>i</td>
<td>©</td>
<td>i</td>
<td>©</td>
<td>i</td>
</tr>
<tr>
<td>©</td>
<td>i</td>
<td>©</td>
<td>i</td>
<td>©</td>
<td>i</td>
</tr>
<tr>
<td>©</td>
<td>o</td>
<td>©</td>
<td>o</td>
<td>©</td>
<td>o</td>
</tr>
</tbody>
</table>

Continued Table (AP1).
Appendix-II

Following are the SPARQL quires applied to retrieve results from SVN and Bugzilla Ontology by simple string matching.

Retrieve a text comment of an issue containing string "revision".
Retrieve a commit message of the revisions containing string "bug".

<table>
<thead>
<tr>
<th>Revision</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>10910</td>
<td>Fixed a bug in the new UMLTextField - when inserting a string inside the document, the cursor won’t jump to the end.</td>
</tr>
<tr>
<td>269</td>
<td>Adjusted ProjectMember name handling to fix issue #455. Issue number: 455.</td>
</tr>
<tr>
<td>343</td>
<td>Patch to fix issue 4558, as supplied by Christian.</td>
</tr>
<tr>
<td>463</td>
<td>Changes to critics for associations and wizard for aggregation to fix issue #19.</td>
</tr>
<tr>
<td>540</td>
<td>Fix issue 1777 where the Action &quot;remove from diagram&quot; would actually remove the element and put it in Trash when the element is moved.</td>
</tr>
<tr>
<td>251</td>
<td>Move multiplicity out of fig group to fix issue 1215. This undoes change introduced in 1.1.3 which was to fix issue 1007. Issue 251 fixes this.</td>
</tr>
<tr>
<td>474</td>
<td>Buggfix issue 1993.</td>
</tr>
<tr>
<td>252</td>
<td>Rollback changes to fix issue 291, which caused the higher priority issue 2574. This undoes the changes in vers 1.7.9 of FigModel.</td>
</tr>
<tr>
<td>711</td>
<td>Revert initial values field to use old deprecated class to fix issue 1378.</td>
</tr>
<tr>
<td>7464</td>
<td>First step to fix issue 1396 - model used as namespace in generation.</td>
</tr>
<tr>
<td>874</td>
<td>This modification should fix issue 225. It sets the name of the figure to the name of a passed Interface node, so you can access it.</td>
</tr>
<tr>
<td>8745</td>
<td>Next attempt to fix issue 2079: &quot;Edges do not stick to package bounds.&quot; Besides for a Package, also solved for Choice, MInterface.</td>
</tr>
<tr>
<td>8889</td>
<td>Corrected the failing test since my latest commits to the DetailsPane class to fix issue 1354.</td>
</tr>
</tbody>
</table>
Retrieve a commit message of the revisions containing string “Issue number”.

Revision 5089
- Issue number: 1832
- Fixed bug in LGPL CVS.

Revision 5072
- Issue number: 2542
- Fixed bug implemented in the critical path detect “circular” association classes.

Revision 50139
- Issue number: 1894
- Fixed bug in the name of the “unnamed” CVS.

Revision 50204
- Issue number: 3547
- Fixed bug in calling the ant.bat script provided with ant 1.6.2 CVS.

Revision 50259
- Issue number: 3652
- First implementation of CodeGenerator CVS.

Revision 50131
- Issue number: 3773
- Obtained from CVS.

Revision 50135
- Issue number: 3782
- Submitted by Thierry Lach.

Revision 50141
- Issue number: 3793
- Obtained from CVS.

Revision 50130
- Issue number: 3804
- Submitted by Thierry Lach.

Revision 50125
- Issue number: 3814
- Added new features.

Revision 50122
- Issue number: 3824
- Updated some features.

Revision 50121
- Issue number: 3834
- Added a call to targetSet after targetRemoved had been called.

Revision 50120
- Issue number: 3844
- Added new methods.

Revision 50119
- Issue number: 3854
- Added package documentation.

Revision 50118
- Issue number: 3864
- Added source for namespace package CVS.

Revision 50117
- Issue number: 3874
- Avoid NPE in removeObsoletedFeatures CVS.

Revision 50116
- Issue number: 3884
- Updated some features.

Revision 50115
- Issue number: 3894
- Added source for namespace package CVS.

Revision 50114
- Issue number: 3904
- Added source for namespace package CVS.

Revision 50113
- Issue number: 3914
- Added source for namespace package CVS.

Revision 50112
- Issue number: 3924
- Added source for namespace package CVS.

Revision 50111
- Issue number: 3934
- Added source for namespace package CVS.

Revision 50110
- Issue number: 3944
- Added source for namespace package CVS.
In what follows, several quires applied to retrieved information from the SVN and Bugzilla Ontology.

<table>
<thead>
<tr>
<th>Query Editor</th>
<th>Query Library</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT DISTINCT ?Author WHERE { revision:author ?Author }</code></td>
<td>![Query applied to retrieve all author names from a SVN Ontology]</td>
</tr>
</tbody>
</table>

Query Editor ([Query Library](#))
### Query Editor

#### Query applied to retrieve an issue number, its assignee and the name of author in SVN.

```sql
SELECT Issue, Revision, SVN_author, Issue_Assignee
WHERE
  Issue.hasResolution = Revision
  AND Issue.hasAssignee = Issue_Assignee
AND Revision.svn_author = SVN_author
```

### Query applied to compare a issue description with SVN commit message.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Revision</th>
<th>SVN_author</th>
<th>Issue_Assignee</th>
</tr>
</thead>
<tbody>
<tr>
<td>issue5042</td>
<td>svn:Revision15911</td>
<td>maurelio1234</td>
<td>Person_maurelio1234</td>
</tr>
<tr>
<td>issue5235</td>
<td>svn:Revision15897</td>
<td>mvw</td>
<td>Person_mvw</td>
</tr>
<tr>
<td>issue5256</td>
<td>svn:Revision15814</td>
<td>mvw</td>
<td>Person_mvw</td>
</tr>
<tr>
<td>issue5258</td>
<td>svn:Revision15815</td>
<td>mvw</td>
<td>Person_mvw</td>
</tr>
<tr>
<td>issue5260</td>
<td>svn:Revision15911</td>
<td>maurelio1234</td>
<td>Person_maurelio1234</td>
</tr>
<tr>
<td>issue5478</td>
<td>svn:Revision15956</td>
<td>maurelio1234</td>
<td>Person_maurelio1234</td>
</tr>
<tr>
<td>issue5482</td>
<td>svn:Revision15972</td>
<td>maurelio1234</td>
<td>Person_maurelio1234</td>
</tr>
<tr>
<td>issue5493</td>
<td>svn:Revision15969</td>
<td>maurelio1234</td>
<td>Person_maurelio1234</td>
</tr>
<tr>
<td>issue5497</td>
<td>svn:Revision16135</td>
<td>mvw</td>
<td>Person_mvw</td>
</tr>
<tr>
<td>issue5542</td>
<td>svn:Revision16289</td>
<td>mvw</td>
<td>Person_mvw</td>
</tr>
<tr>
<td>issue5548</td>
<td>svn:Revision16270</td>
<td>mvw</td>
<td>Person_mvw</td>
</tr>
<tr>
<td>issue5557</td>
<td>svn:Revision291</td>
<td>jrobbins</td>
<td>Person_jrobbins</td>
</tr>
<tr>
<td>issue5581</td>
<td>svn:Revision16493</td>
<td>jrobbins</td>
<td>Person_jrobbins</td>
</tr>
<tr>
<td>issue5598</td>
<td>svn:Revision16541</td>
<td>jrobbins</td>
<td>Person_jrobbins</td>
</tr>
<tr>
<td>issue5602</td>
<td>svn:Revision16539</td>
<td>jrobbins</td>
<td>Person_jrobbins</td>
</tr>
<tr>
<td>issue5657</td>
<td>svn:Revision16596</td>
<td>jrobbins</td>
<td>Person_jrobbins</td>
</tr>
<tr>
<td>issue5670</td>
<td>svn:Revision16692</td>
<td>jrobbins</td>
<td>Person_jrobbins</td>
</tr>
<tr>
<td>issue5681</td>
<td>svn:Revision16670</td>
<td>jrobbins</td>
<td>Person_jrobbins</td>
</tr>
<tr>
<td>issue5685</td>
<td>svn:Revision16763</td>
<td>jrobbins</td>
<td>Person_jrobbins</td>
</tr>
</tbody>
</table>

### Table: Resolved in

<table>
<thead>
<tr>
<th>Issue</th>
<th>Description</th>
<th>Resolved in</th>
<th>commit_message</th>
</tr>
</thead>
<tbody>
<tr>
<td>issue5235</td>
<td>UML1.4 notation doesn't parse frozen issues</td>
<td>svn:Revision15897</td>
<td>Fix for issue 5235: UML1.4 notation doesn't parse frozen issues</td>
</tr>
<tr>
<td>issue5256</td>
<td>diagrams appear in profile configuration</td>
<td>svn:Revision15814</td>
<td>Fix for issue 5256: diagrams appear in profile configuration</td>
</tr>
<tr>
<td>issue5258</td>
<td>2 perspective rules with same name: ClassifierRole</td>
<td>svn:Revision15815</td>
<td>Fix for issue 5258: 2 perspective rules with same name: ClassifierRole</td>
</tr>
<tr>
<td>issue5260</td>
<td>[WFRs] Reorganize WFRs</td>
<td>svn:Revision15911</td>
<td>Fix for issue 5260: [WFRs] Reorganize WFRs</td>
</tr>
<tr>
<td>issue5478</td>
<td>Critiques aren't being generated</td>
<td>svn:Revision15969</td>
<td>Fixing issue 5478: Critiques aren't being generated</td>
</tr>
<tr>
<td>issue5493</td>
<td>Apply Stereotypes in diagram popup menu</td>
<td>svn:Revision15972</td>
<td>Fix for issue 5493: Apply Stereotypes in diagram popup menu</td>
</tr>
<tr>
<td>issue5497</td>
<td>Class that shows stereotype grows on no association</td>
<td>svn:Revision15969</td>
<td>Fix for issue 5497: Class that shows stereotype grows on no association</td>
</tr>
<tr>
<td>issue5534</td>
<td>Association end label position is further from association</td>
<td>svn:Revision15815</td>
<td>Remove unnecessary code. This fixes issue 5534: Association end label position is further from association</td>
</tr>
<tr>
<td>issue5548</td>
<td>Can't save project referencing user-defined profile</td>
<td>svn:Revision16739</td>
<td>Fix for issue 5548: Can't save project referencing user-defined profile</td>
</tr>
<tr>
<td>issue5557</td>
<td>Exception when double clicking on an association</td>
<td>svn:Revision16739</td>
<td>Fix for issue 5557: Exception when double clicking on an association</td>
</tr>
<tr>
<td>issue5598</td>
<td>FillColor applied to stereotype figs</td>
<td>svn:Revision16739</td>
<td>Fix for issue 5598: FillColor applied to stereotype figs</td>
</tr>
<tr>
<td>issue5602</td>
<td>Notification ignores Association to self when activating</td>
<td>svn:Revision16593</td>
<td>Fix for issue 5602: Notification ignores Association to self when activating</td>
</tr>
<tr>
<td>issue5669</td>
<td>Activations have no border</td>
<td>svn:Revision16692</td>
<td>Fix for issue 5669: Activations have no border</td>
</tr>
<tr>
<td>issue5667</td>
<td>Java RE not creating figs for Packages</td>
<td>svn:Revision16692</td>
<td>Fix for issue 5667: Java RE not creating figs for Packages</td>
</tr>
<tr>
<td>issue5670</td>
<td>java source import uses profile, but classfigs</td>
<td>svn:Revision16739</td>
<td>Fix for issue 5670: java source import uses profile, but classfigs</td>
</tr>
<tr>
<td>issue5681</td>
<td>Selection indication on attribute remains</td>
<td>svn:Revision16739</td>
<td>Fix for issue 5681: Selection indication on attribute remains</td>
</tr>
<tr>
<td>issue5685</td>
<td>Dropping a component state does not take effect</td>
<td>svn:Revision16739</td>
<td>Fix for issue 5685: Dropping a component state does not take effect</td>
</tr>
</tbody>
</table>

### Exception when double clicking on an association

```sql
SELECT Issue, Revision, SVN_author, Issue_Assignee
WHERE
  Issue.hasResolution = Revision
  AND Issue.hasAssignee = Issue_Assignee
AND Revision.svn_author = SVN_author
```

### Fix for issue 5557: Exception when double clicking on an association

Fixing issue 5557: Exception when double clicking on an association.

### Fix for issue 5598: FillColor applied to stereotype figs

Fixing issue 5598: FillColor applied to stereotype figs.

### Fix for issue 5602: Notification ignores Association to self when activating

Fixing issue 5602: Notification ignores Association to self when activating.

### Fix for issue 5669: Activations have no border

Fixing issue 5669: Activations have no border.

### Fix for issue 5667: Java RE not creating figs for Packages

Fixing issue 5667: Java RE not creating figs for Packages.

### Fix for issue 5670: java source import uses profile, but classfigs

Fixing issue 5670: java source import uses profile, but classfigs.

### Fix for issue 5681: Selection indication on attribute remains

Fixing issue 5681: Selection indication on attribute remains.

### Fix for issue 5685: Dropping a component state does not take effect

Fixing issue 5685: Dropping a component state does not take effect.

---

116
In order to retrieve some statistics of an ArgoUML project from the Bugzilla Ontology, we applied several queries on the Bugzilla Ontology. As a result, the queries retrieved following statistics. (1) The total number of persons involved in the Bugzilla Ontology (i.e. Person in the role of Assignee, Commenter, Creator of an Attachment, and Involved Person). (2) Total number of the issues reported in the current release. (3) Total number of activities related to the issues. (4) Total number of Comments. (5) Total number of computer systems (i.e. combination of an operating system and platform). Following table shows some statistics obtained from Bugzilla Ontology.
In order to retrieve some statistics of an ArgoUML project, we applied several queries on the SVN ontology. As a result, the queries retrieved following statistics. (1) Total number of the files in SVN repository. (2) Total number of revisions. (3) Total number of authors (i.e. developers / maintainers). (4) Total number of releases tagged in SVN repository.

Following table shows some statistics obtained from SVN ontology.

| Total number of files used in different revision | 56920 |
| Total number of revisions | 16793 |
| Total number of Authors (developers /maintainers) | 50 |
| Total number of Releases tagged through revisions (i.e. including tags “ALPHA_1 to ALPHA_n”, and “BETA_1 to BETA_n”). | 156 |

Some Statistics of SVN Ontology.