Crawling Bug Tracker for Semantic Bug Search

Ha Manh Tran, Georgi Chulkov, and Jürgen Schönwälder

Computer Science, Jacobs University Bremen, Germany
`{h.tran,g.chulkov,j.schoenwaelder}@jacobs-university.de`

Abstract. The Web has become an important knowledge source for resolving system installation problems and for working around software bugs. In particular, web-based bug tracking systems offer large archives of useful troubleshooting advice. However, searching bug tracking systems can be time consuming since generic search engines do not take advantage of the semi-structured knowledge recorded in bug tracking systems. We present work towards a semantics-based bug search system which tries to take advantage of the semi-structured data found in many widely used bug tracking systems. We present a study of bug tracking systems and we describe how to crawl them in order to extract semi-structured data. We describe a unified data model to store bug tracking data. The model has been derived from the analysis of the most popular systems. Finally, we describe how the crawled data can be fed into a semantic search engine to facilitate semantic search.

Key words: Bug tracking system, Bug crawler, Semantic search

1 Introduction

Trouble ticket systems and bug tracking systems are widely deployed in the information technology industry. Software and hardware companies use bug tracking systems during the development cycle to track bugs and design issues, or during later phases of the product lifecycle to keep track of defect reports and to obtain quality indicators. Almost all large open source projects maintain online bug tracking systems. In addition, there are many bug tracking systems supporting people who package open source software components for various software distributions. Some companies provide special online support forums (also known as communities or knowledge bases) for their products that often resemble bug tracking systems.

The fast growing amount of online information that can be used to resolve problems has led to a situation where system administrators and network operators often use search engines in order to find hints how to resolve a specific problem. However, it is our experience that searching in this way is not as efficient as we would like it to be; generic search engines do not seem to take advantage of the data found in trouble ticket or bug tracking systems.

Trouble ticket systems and bug tracking systems contain semi-structured data. Predefined fields are used to keep track of the status and metadata associated with a problem report while textual descriptions are used to describe the problem and to document the process for resolving the problem. Exploiting this semi-structured data has been considered a challenge. An early study in [1] suggests avoiding textual descriptions as much as possible since they cause difficulties in processing trouble tickets automatically. While this recommendation makes sense from a programmer's perspective, it clearly does not match the requirements of users who prefer to write down free-form text. Other studies [2, 3] exploit only predefined fields that only use binary, numeric or symbolic values, an approach that has several limitations. These works have experimented with using trouble tickets as a basis for case-based reasoning (CBR) systems.

The semi-structured data contained in bug tracking systems is a valuable resource. It can be used to construct specialized search systems that have access to and knowledge of metadata out of reach to general text-based search engines. It can also be used to build automated reasoning systems that help to diagnose problems based on past experience. The goal of the work presented in this paper is to build a semantics-based bug search system and a bug dataset that can be used by a distributed case-based reasoning system which we are developing [4, 5]. The aim of our system is to find relevant information quickly and to cope with the fact that the relevance of bug records changes quickly, due the short lifecycles of today's software products and services. Our contribution in this paper is fourfold:

1. We present a study of popular bug tracking systems and their features that can be used by semantic bug search or automated reasoning systems.
2. We present two methods to crawl bug tracking systems and to extract data.
3. Based on an analysis of the data models used by existing bug tracking systems, we develop a unified data model to store bug tracking data.
4. Finally, we apply a multi-vector representation (MVR) [5] to bug reports in order to enable semi-structured bug data search on the bug database.

The rest of the paper is structured as follows: In Section 2, we provide a study of some popular bug tracking systems. Section 3 explains how bug data can be extracted from these systems and Section 4 describes the unified data model we have developed to store the extracted bug data. Section 5 explains how the data are used for finding similar bugs and provides an experimental evaluation. We discuss related work in Section 6 before we conclude the paper in Section 7.

2 Bug Tracking Systems

Trouble ticket systems (TTS) have been widely used by network operators in order to assure the quality of communication services. A bug tracking system (BTS) is a special trouble ticket system used to keep track of software bugs. BTSs in general aim to improve the quality of software products. They do so by keeping track of current problems, and maintaining historical records of previously experienced issues. They also establish an expert system that allows to search for similar past problems, and provide reports and statistics for performance evaluation of the services [6].

Table 1. Overview of bug tracking systems and some of their features (as of October 2007). Items marked with * are optional for each site. † reads dependencies

Tracker	**License**	**Access**	**Updates**	**Schema**	**Dep.**†	**Search**
Bugzilla	MPL	HTML,XML-RPC*	SMTP, RSS*	textual	optional	filter,keywords
Mantis	GPL	HTML, SOAP*	SMTP, RSS	graphical	yes	filter
Trac	BSD	HTML	SMTP, RSS*	graphical	no	filter,keywords
Debian BTS	GPL	HTML, SMTP	SMTP	unknown	optional	filter
phpBugTracker	GPL	HTML	SMTP	textual	yes	filter,keywords
Flyspray	LGPL	HTML	SMTP,RSS,XMPP	unknown	yes	filter,keywords

We explore the features of several BTSs, focusing on several properties that are important for obtaining data from them. For each BTS we checked whether the BTS supports the functionality in question, and if so, which bug sites (i.e., specific installations of a BTS) support that function. A small sample of our results for BTSs and sites is given in Table 1 and Table 2 respectively. An explanation of each column in these tables is given below.

For each BTS, we looked at how the system is licensed for use, how its collection of bugs can be accessed, whether it is possible to easily receive notification of updated data, whether the database schema used by the BTS was available, whether the BTS keeps track of bug dependencies, and whether the BTS can be searched for bugs with a given property.

The license of a BTS affects its popularity. Generally, large free software projects tend to prefer a BTS licensed as free software itself. These projects also tend to be the ones that make their BTS sites public. For this reason, we excluded proprietary BTSs from our study.

The most important function of a BTS is retrieving bug reports in their most current states. All systems must at a minimum have an HTML-based web interface, but convenient automated retrieval requires a formalized programmatic interface, either based on XML-RPC or SOAP. While all BTSs support e-mail (via SMTP) as an update notification mechanism, e-mail is non-trivial to use for a web application. For instance, to receive notifications for all bug reports in a BTS by e-mail, an application would need to have its own e-mail address, register an account with that e-mail address in the BTS, and subscribe for every bug report of interest. Some systems support RSS or Atom feeds, which are significantly easier to use by a program.

In order to understand the structure of the information stored in a BTS, we investigated whether the underlying data model is documented. Some systems provide this information in a textual format while others provide graphical representations, usually in ad-hoc notations. Some systems do not provide a clear description of the data model underlying the BTS and it is necessary to reverse engineer the data model by looking at concrete bug reports.

Tracking any dependency relations between bug reports is useful, because it helps to correlate bugs. Generally, a bug is dependent on another bug if it cannot be resolved or acted upon, until the dependency is itself resolved or acted upon.

Some systems allow full keyword search for their reports, while others only support searching via a set of predefined filters applied on the entire bug database.

The former is more useful for an automated system that aims to provide keyword search capabilities itself.

Table 2. Some popular bug tracking sites (as of October 2007). A plus indicates that we were unable to get precise numbers and our numbers present a lower bound. † reads dependencies

Site	System	Version	Bugs	Activity	Custom	RPC	RSS	Dep.†
bugs.debian.org	Debian BTS	N/A	349346	1036	N/A	no	no	no
bugs.kde.org	Bugzilla	unknown	9655+	24+	light	no	no	no
bugs.eclipse.org	Bugzilla	unknown	204600	746	heavy	yes	yes	yes
bugs.gentoo.org	Bugzilla	unknown	183365	538	none	no	yes	yes
bugzilla.mozilla.org	Bugzilla	3.0.1+	173885	721	none	yes	yes	yes
bugzilla.redhat.com	Bugzilla	2.18-rh	177724	unknown	light	yes	yes	yes
qa.netbeans.org	Bugzilla	unknown	116639+	unknown	heavy	no	no	yes
bugs.digium.com	Mantis	unknown	10765	63	none	no	yes	yes
bugs.scribus.net	Mantis	1.0.7	6142	24	none	no	yes	yes
bugtrack.alsa-project.org	Mantis	1.0.6	3430	22	none	no	no	yes
dev.rubyonrails.org	Trac	0.10.5dev	11493+	unknown	none	no	yes	no
trac.edgewall.org	Trac	unknown	5948	unknown	none	no	yes	no
bugs.icu-project.org	Trac	0.10.4	5845	unknown	none	no	yes	no

Based on popularity and available documentation, we chose to focus on Bugzilla, Mantis, Trac and Debian BTS. With the exception of the Debian BTS, which is only used for the Debian operating system, the BTSs we chose all publish lists of known public sites that use them for bug tracking. Starting from these lists, we investigated all sites that are accessible and did not require authentication to browse their repositories (about 85 sites). Table 2 lists the sites with the largest number of bugs for each BTS. For each site, Table 2 shows which version of what BTS is used, how many bugs are stored there in total and how many have been added in one week, indicating the activity of the site. The table also specifies whether the site has been customized from its base BTS, whether it supports a programmatic XML-RPC interface or RSS feeds, and whether the site supports bug dependency relations.

The version of the underlying BTS largely impacts the set of available features. For example, Bugzilla only supports RSS feeds as of version 2.20, and XML-RPC as of version 3.0. Note that some sites hide this version number, possibly because this information may be sensitive with respect to security exploits in the BTS source code. The number of stored bugs and the rate of opening new bugs indicate the popularity and activity of a site. The margin between the most popular Bugzilla sites and the most popular sites using other BTSs is very large. We believe that the reason is that Bugzilla was the first widely-known open-source BTS when it was released in 1998. Mantis was only started in late 2000, and Trac is even newer.

Some sites customize their BTS in order to provide better integration of the bug tracker into the rest of their web site. While some sites only change the visual appearance of the BTS (marked as “light” customization in Table 2), others also modify the functionality of the BTS (marked as “heavy” customization). Customized sites pose a problem for automated bug retrieval: a system that is designed to derive structured data from presentational HTML (see Section 3.2) will generally fail to handle a significant change of a site's appearance. In addition, customizing a BTS naturally makes upgrading the site to a newer version of the BTS much more difficult; therefore customized sites tend to lag behind in version number, and consequently lack features such as RSS feeds or XML-RPC support.

Programmatic interfaces provided by protocols like XML-RPC or SOAP can be used by programs to directly query a bug tracker for structured data, without having to guess the value of any fields presented in human-readable form (HTML, SMTP). While this greatly simplifies interfacing to that bug tracker, no BTS currently makes such an interface a default option. It is an optional feature of the BTS at best, and not supported at all at worst. Only very few sites actually deploy and enable such programmatic interfaces, and clearly relying on their availability is not sufficient. RSS, on the other hand, is much more widely supported. RSS feeds allow programs to query a bug tracker for any updated bug reports, and while they are not as useful as XML-RPC interfaces, they still provide a better alternative to SMTP update notification.

3 Retrieving Data from BTSs

This section describes ways to retrieve semi-structured data from BTSs. First, we describe how we can exploit application programming interfaces (APIs) provided by the BTSs themselves. Since such APIs are only available on some sites, we have also implemented a web crawler specialized for bug tracking systems.

3.1 Exploiting APIs

The Bugzilla BTS provides an XML-RPC web service interface for users to access and modify bug reports. Users can write an external tool to interact with Bugzilla through several web service modules: The *User* module allows applications to create user accounts and to log in/out using an existing account. The *Bug* module can be used to file a new bug in Bugzilla, or to get information about already filed bugs. The *Product* module allows applications to list the available Products and to get information about them. Products are Bugzilla's top-level categories for bugs. Finally, the *Bugzilla* module provides functions to retrieve some general information about a Bugzilla installation.

The Bugzilla XML-RPC API in addition allows programs to retrieve a large number of bug reports in a single request using an array of bug identifiers. This saves much time when downloading many bug reports from a single Bugzilla website that supports XML-RPC.

We have implemented a crawler that uses several methods provided by the Bugzilla web service interface. It acts like an XML-RPC client that submits a bug identifier to a Bugzilla server and obtains the details of the bug (i.e., a list of field-value pairs) from the Bugzilla server. Note that the XML-RPC API is rarely available on production BTS installations (see Section 2) and is incomplete: it

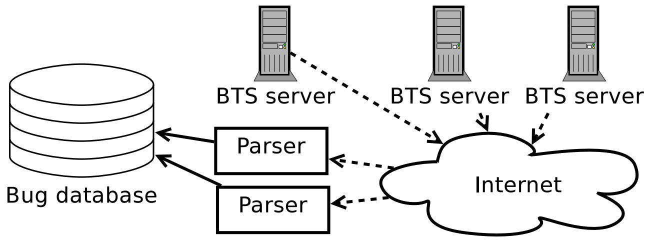

Fig. 1. Architecture of the Buglook crawler

restricts users from retrieving any attachments from a bug report, as well as the bug description or any related discussion entries. We access this information via another tool (described below).

Unlike all other BTSs, the Debian BTS allows users to access the raw bug data directly. Users can use the *rsync* utility to copy the whole bug database from bugs-mirror.debian.org. Debian's database is split into three sections: *bts-spool-db* for the active bug report spool, *bts-spool-archive* for bug reports that have been closed for a while and thus archived, and *bts-spool-index* for the bug index files. Each bug report is stored in four different files whose names consist of the bug number and either .log, .report, .status, or .summary as an extension.

3.2 Crawling with Buglook

While some BTSs provide a machine-readable web service interface to their bug data, most do not. In all systems where such an interface is supported, it is an optional feature, and because optional features require additional effort from an administrator to be set up, they are rarely available. In addition, a web service interface often provides much less data than the human-readable web interface that is most commonly used. Clearly, relying on the availability of a web service API is unrealistic. To solve this problem, we created Buglook [7], a tool which attempts to directly use the presentational HTML-based web interface in order to get as much access to information as ordinary users.

The problem with presentational HTML pages is that the same structure can be presented in vastly different ways. As an example, consider an algorithm that must detect the end of a bug report comment and the beginning of the next one. In HTML, this boundary could be encoded as a closing `<div>` tag and the opening of another `<div>`, or as a new paragraph (`<p>`), or why not a sequence of newlines (`<br>`)? There is nothing preventing the same elements from being used in another context, while being rendered differently (dictated by CSS tags). No consistency can be expected.

To tackle this problem, we note the following: (i) Because bug report pages are generated from a template, all bug reports within a single BTS site have the same structure. An algorithm that can parse one bug report can parse all bugs in that site. (ii) BTS sites that use the same software have similar bug structure, and

Table 3. Severity of bugs

Unified model	Bugzilla	Trac	Mantis	Debian
critical	blocker, critical	blocker, critical	block, crash	critical, grave, serious
normal	major	major	major	important, normal
minor	minor, trivial	minor, trivial	minor, tweak, text, trivial	minor
feature	enhancement	-	feature	wishlist

often similar appearance. The underlying BTS software determines the structure of the data it can work with, and only allows presentational customization of the displayed HTML pages. (iii) For each BTS, there is a canonical appearance. In general, most sites do not find it necessary to customize their appearance, and use the one that the BTS provides by default.

Buglook (Figure 1) uses a small set of parsers defined for each BTS's canonical appearance, together with specialized parsers for the most important customized sites. These parsers, called "site modules", can provide a very high degree of coverage of all BTS sites. The set of sites covered by a site module is its "sitetype". Sitetypes are essentially equivalence classes of BTS sites, with respect to parsing.

Buglook's bug extraction component consists of two essential parts - the set of site modules, and a common component independent of all of them. The common component is responsible for generic tasks such as downloading web pages, parsing HTML, etc. The site modules encapsulate all logic unique to a given sitetype. This distinction allows more sitetypes to be supported with minimal duplication of effort. To support a sitetype, a site module must implement a fixed interface to the common component.

4 Unified Data Model

In order to integrate bug data from different BTSs into a single bug database, we define a unified data model for bugs. This model must be simple and easy to use for our purpose; it is not necessary to be able to represent all available details of all systems. Our investigation of the database schemas of the four BTSs we considered exposes several interesting observations. The bug formats of Bugzilla, Trac and Mantis share many similar fields that can be classified in two main groups:

1. The administrative metadata associated with a bug is often represented as field-value pairs with very precise semantics, such as *id*, *severity*, *reporter*, *summary*, among others.
2. The descriptions detailing the bug and any followup discussion or actions are typically represented as free-form (i.e., non-formal) textual attachments.

Because the unified data model is used to support semantic search, we aim to extract fields from bug reports in such a way as to minimize the loss of bug information. We introduce new fields that establish the relationships between bugs or provide for more sophisticated classification. The values of these fields

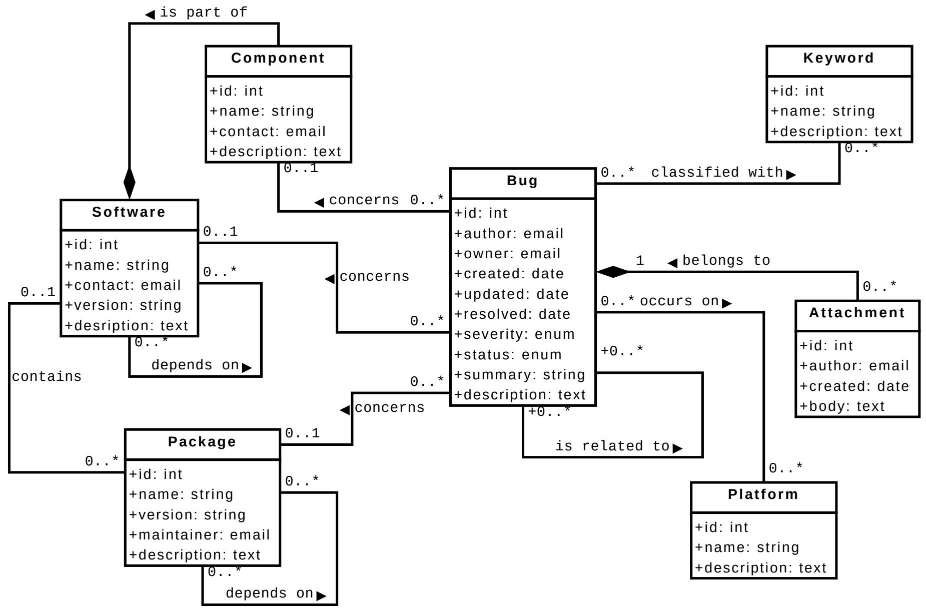

Fig. 2. Unified bug data model represented as a UML diagram

can be derived differently for each BTS: Mantis users can manually specify the relation of a bug to other bugs when reporting it; Debian users can indicate which package a bug relates to; and Bugzilla and Trac offer a keyword field that enables the classification of bugs. To exploit data from a bug's description and its attachments, we use several text processing techniques [8, 9].

Figure 2 shows our unified bug data model in the form of a UML class diagram. The central class is the `Bug` class. The `id` attribute of a `Bug` instance uniquely identifies a bug. We use the URL where a bug can be retrieved as its identifier. Most of the attributes of a `Bug` instance can be easily extracted from the retrieved data. Our `severity` attribute is probably the most interesting to fill correctly, because BTSs have very different severity classifications for bugs. Table 3 shows how we map the severity values of the BTSs into our data model, which only distinguishes the severity values *critical*, *normal*, *minor*, and *feature*. The `status` attribute of a `Bug` instance only has two values: the value *open* represents what BTSs call *unconfirmed*, *new*, *assigned*, *reopened* bugs while the value *fixed* represents what BTSs call *resolved*, *verified*, and *closed* bugs.

Free-form textual descriptions are modelled as `Attachment` objects. Every `Attachment` belongs to exactly one `Bug` object. Some BTSs provide information about the platforms affected by a bug. We represent platforms (such as "Windows2000" or "MacOS X") as `Platform` objects. The `Keyword` class represents keywords used to describe and classify bugs.

The left part of Figure 2 models what piece of software a bug is concerned with. While some BTSs are only concerned with bugs in a specific piece of software, software in larger projects is split into components and bugs can be related to specific components. The classes `Software` and `Component` model this structure. The Debian BTS is somewhat different from the other BTSs as it is primarily used to track issues related to software "packages", that is software components packaged for end user deployment. Since there is a large amount of meta information available for Debian software packages (dependency, maintainer and version information), we have introduced a separate `Package` class to represent packaged software.

5 Semi-Structured Bug Data Search

BTSs only support keyword search and restricted meta-data search by predefined fields and values (as discussed in Section 2); e.g., searching for bugs with the *resolved* status or bugs with the *critical* severity. This section presents the performance of different search algorithms on semi-structured bug data from the unified dataset.

We have previously evaluated the combination of fulltext search and meta-data search, namely `ft-md` search, on the CISI and MED bibliographic datasets whose documents contain semi-structured data [5]. With the bug dataset, meta-data search exploits significant keywords extracted from bug contents, such as type of problems, scope of problems, typical symptoms, error messages and distinct terms. A set of keywords is represented by a field-value vector. The similarity of two field-value vectors is estimated by the sum of weight values of matched keywords. Fulltext search involves indexing terms from bug contents using text processing techniques [8, 9]. Each bug is converted to a term vector which is then transformed to a real number vector (or a semantic vector) using algebraic computation. The similarity of two semantic vectors is evaluated by the cosine of these vectors.

We consider keyword search as baseline search that evaluates the similarity between a bug and a query by simply matching keywords from the query to the bug content without considering the significance of keywords. We use a *matching rate* metric to compare the performance of the combination of search algorithms: `ft-md` combining fulltext search and meta-data search, `ft-bl` combining fulltext search and keyword search, and `md-bl` combining meta-data search and keyword search. The matching rate r is the ratio of the number of the identical bugs obtained by two search algorithms to the minimum number of bugs obtained by these algorithms for a query:

$$r = \frac{|S_x \cap S_y|}{min(N_x, N_y)} \tag{1}$$

where S_x and S_y are the resulting set of search algorithms x and y per query, $|S|$ is the size of set S, and parameters N_x and N_y are the total number of bugs obtained by search algorithms x and y per query. Intuitively, if two search algorithms are good, the probability of a large number of identical bugs obtained by these algorithms is high. This metric is more feasible and flexible than the recall rate or precision rate metrics that require knowledge of the correct number of relevant bugs per query. Note that it is difficult to obtain this number from a new and large dataset.

The evaluation of a semantic bug search engine on a large dataset of several hundred thousand bugs will be reported in another study. In these experiments, the dataset contains 11.077 bugs, and the number of obtained bugs N_{ft}, N_{md} and N_{bl} are set to 100 by sorting the resulting sets according to the similarity value and selecting only the top N elements. A set of 50 queries include pieces of textual descriptions, distinct keywords, typical symptoms and error messages that are extracted from bug contents. Terms or keywords from bug contents are stemmed by the Porter stemming algorithm [10] and weighted by the term frequency-inverse document frequency (`tf-idf`). Semantic vectors are generated by computing singular value decomposition using the single-vector Lanczos algorithm [11] implemented in `svdlibc`. The experiments were performed on an x86_64 GNU/Linux machine with two dual-core AMD Opteron(tm) processors running at 2 GHz with 4 GB RAM.

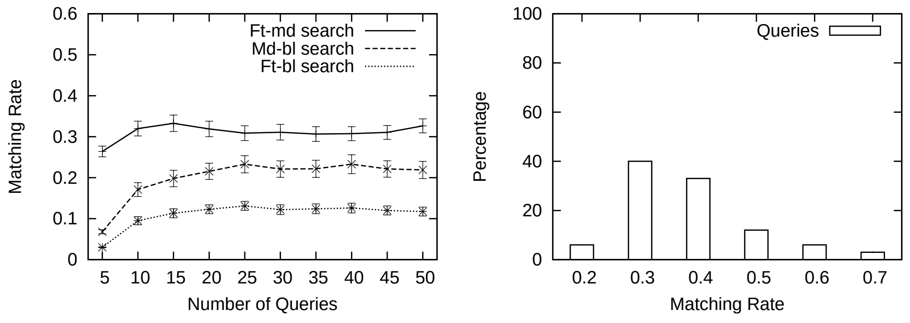

Fig. 3. Average matching rate by number of queries for `ft-md`, `md-bl` and `ft-bl` (left). Query distribution by matching rates for `ft-md` (right).

The left plot shown in Figure 3 reports the average matching rate of `ft-md`, `md-bl` and `ft-bl` over an increasing number of queries. The `ft-md` line stays at at a matching rate of 0.3 on average and reaches 0.33 finally, whereas the `md-bl` and `ft-bl` lines start lowly and reach 0.22 and 0.12, respectively. `Ft-md` found more identical bugs than other combinations. Furthermore, `md-bl` obtained better results than `ft-bl`. The results of the last two combinations are relatively different, illustrating that `bl` search obtains less consistent and reliable results. `Bl` search combines better with `md` search than with `ft` search because `md` and `bl` both use keyword comparison to estimate similarity.

The query distribution for `ft-md` shown in the right plot in Figure 3 indicates that more than 50% of the queries receives a matching rate higher than 0.3. These queries tend to focus on specific distinct characteristics of bugs, whereas the other queries tend to be more general or vague, resulting in a large number of improper bugs obtained.

We relax the number of obtained bugs for only `bl` search, namely `rlbl`; N_{rlbl} is set to 500. The left plot shown in Figure 4 shows that `md-rlbl` improves the matching rate to 0.25 on average, whereare `ft-rlbl` remains unchanged. `Rlbl` search and `md` search finds more identical bugs than `rlbl` search and `ft` search. While `md` search is similar to `rlbl` search, it is much different from `ft` search in the way of measuring similar bugs. Combining `ft` search and `md` search, therefore, works well for semi-structured bug data.

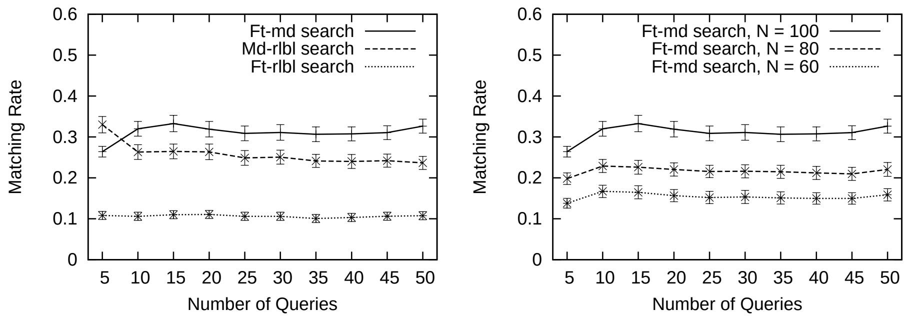

Fig. 4. Average matching rate by number of queries for `ft-md`, `md-rlbl` and `ft-rlbl` with a large number of bugs obtained by baseline search (left). Average matching rate by number of queries with various numbers of bugs obtained by `ft-md` (right).

We investigate further `ft-md` search by restricting the number of obtained bugs; N_{ft} and N_{md} are both set to 100, 80 and 60. The right plot in Figure 4 indicates that the matching rate reduces when the number of obtained bugs reduces. Note that bugs are chosen by their ranking. The ranking of obtained bugs are different between `ft` search and `md` search. This is caused by three reasons: first, while BTSs may contain duplicated bugs, the number of these bugs is small, thus the number of truly similar bugs is also small; second, the ranking values of obtained bugs contain errors from indexing and ranking bugs, especially when a query is general, ranking many similar bugs is affected by these errors; last, as described above, ineffective queries make results inaccurate. These reasons also cause the low matching rate. However, since the bug dataset is wide and diverse in scope, we believe that `ft-md` search achieving an average matching rate of 0.3 is reasonable.

6 Related Work

The X.790 recommendation from ITU-T [12], the RFC 1297 from the IETF [13] and the NMF501 and NMF601 documents from the TMF [14, 15] define terminology and basic functions for reporting and managing trouble tickets. NetTrouble [16] introduces advanced features for trouble ticket systems (TTSs) that include a distributed database concept for geographic dissemination, an administrative domain concept for the multi-organizational characteristics of network management environments, and an administrative model for hierarchical decomposition. The work in [17] proposes a generic interface and a generic data structure, namely customer service management trouble report, to support inter-domain problem management between customer and service provider. Our work towards a unified data model is to some extend related to these efforts. However, instead of designing a feature rich data model from scratch, we took the opposite approach to extract the common core from the data models used by existing systems.

Since the problem-solving knowledge in TTSs can be exploited to search for similar problems or infer typical solutions, several studies discussed in [1] have used TTSs associated with artificial intelligence techniques for finding and resolving similar problems. A study in [2] has proposed a CBR system to resolve network problems by retrieving similar problems and adapting solutions to novel problems. Trouble tickets obtained by a TTS are used as cases for evaluating the system. The DUMBO system [3] also takes advantage of TTSs to propose solutions for network problems. This system provides six types of features to represent trouble tickets, and employs similarity and reliability measurement for proposing solutions. The main limitations of these systems, however, contain the inexpressive representation of trouble tickets and the lack of trouble ticket sources [1]. We consider these issues in this work by using the unified data model that allows various bug reports to be collected in one bug database, and by applying MVR to bug reports to enable semantic search on the bug database.

7 Conclusions

We have provided a study of existing BTSs with a specific focus on the four most popular open source systems, namely Bugzilla, Trac, Mantis and the Debian bug tracking system. Widely used public BTSs based on these software systems contain a large number of bug reports that can be used for building bug datasets. Such datasets are invaluable for evaluating systems such as case-based reasoning engines or semantic search engines. We have used web service APIs and a special purpose web crawler (Buglook) to obtain a large number of bug reports from several large BTSs. In order to store the data in an effective way, we have developed a unified bug data model that is able to capture the most important aspects of the data maintained by the various BTSs we have analyzed. Our model enables interoperable aggregation of data from different sources, useful for various purposes ranging from efficient wide-scale search to automated reasoning systems.

The multi-vector representation method (MVR) [5] has been used to perform semantic search experiments on the unified bug dataset. MVR exploits semi-structured bug data to search for similar bugs with salient features. The experimental results indicate that (i) the combination of fulltext search and meta-data search (using MVR) outperforms the other combinations of fulltext search and baseline search or of meta-data search and baseline search, (ii) baseline search provides less consistent and reliable results, and (iii) the bug dataset is wide and diverse in scope.

We are currently implementing a complete online semantic search system that will accept user queries so that a larger number of users can test and evaluate our system. This system also allows us to evaluate search latency and bug synchronization on a large bug dataset. Future work involves extending the unified data model to support another semantic bug search system, where bug reports are represented in the resource description framework (RDF). In addition, refined and unified datasets are used to evaluate the problem-solving capability of our distributed case-based reasoning system. Such systems will certainly be a practical tool for anyone who needs to troubleshoot a software system with a public bug tracking system.

Acknowledgments The work reported in this paper is supported by the EC IST-EMANICS Network of Excellence (#26854).

References

1. L. Lewis and G. Dreo. Extending Trouble Ticket Systems to Fault Diagnostics. *IEEE Network Special Issue on Integrated Network Management*, 7(6):44–51, 1993.
2. L. Lewis. A Case-Based Reasoning Approach to the Resolution of Faults in Communication Networks. In *Proc. 3rd International Symposium on Integrated Network Management (IM '93)*, pages 671–682. North-Holland, 1993.
3. C. Melchiors and L. Tarouco. Fault Management in Computer Networks Using Case-Based Reasoning: DUMBO System. In *Proc. 3rd International Conference on Case-Based Reasoning and Development (ICCBR '99)*, pages 510–524. Springer-Verlag, 1999.
4. H. M. Tran and J. Schönwälder. Distributed Case-Based Reasoning for Fault Management. In *Proc. 1st International Conference on Autonomous Infrastructure, Management and Security (AIMS '07)*, pages 200–203. Springer-Verlag, 2007.
5. H. M. Tran and J. Schönwälder. Fault Representation in Case-Based Reasoning. In *Proc. 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '07)*, pages 50–61. Springer-Verlag, 2007.
6. D. Bloom. Selection Criterion and Implementation of a Trouble Tracking System: What's in a Paradigm? In *Proc. 22nd Annual ACM SIGUCCS Conference on User Services (SIGUCCS '94)*, pages 201–203. ACM Press, 1994.
7. G. Chulkov. Buglook: a search engine for bug reports. Seminar Report. Jacobs University Bremen, May 2007.
8. S. Deerwester, S. Dumais, T. Landauer, G. Furnas, and R. Harshman. Indexing by Latent Semantic Analysis. *Journal of the Society for Information Science*, 41(6):391–407, 1990.

9. M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, Vector Spaces, and Information Retrieval. *SIAM Review*, 41(2):335–362, 1999.
10. M. F. Porter. An algorithm for suffix stripping. *Readings in Information Retrieval*, pages 313–316, 1997.
11. G. H. Golub and R. Underwood. The block lanczos method for computing eigenvalues. *Mathematical Software III*, pages 361–377, 1977.
12. ITU-T. Trouble Management Function for ITU-T Applications. X.790 Recommendation, 1995.
13. D. Johnson. NOC Internal Integrated Trouble Ticket System Functional Specification Wishlist. RFC 1297, 1992.
14. TMF. Customer to Service Provider Trouble Administration Business Agreement. NMF 501, Issue 1.0, 1996.
15. TMF. Customer to Service Provider Trouble Administration Information Agreement. NMF 601, Issue 1.0, 1997.
16. L. Santos, P. Costa, and P. Simes. NetTrouble: A TTS for Network Management. In *Proc. SBT/IEEE International Telecommunications Symposium (ITS '98)*, pages 480–485. IEEE Computer Society, 1998.
17. M. Langer and M. Nerb. Defining a Trouble Report Format for the Seamless Integration of Problem Management into Customer Service Management. In *Proc. 6th Workshop of the OpenView University Association (OVUA'99)*, 1999.