Collecting Issue Management Data for Analysis with a Unified Model and API Descriptions

Otto Hylli, Anna-Liisa Mattila, and Kari Systä

Department of Pervasive computing, Tampere University of technology, Tampere, Finland
{otto.hylli,anna-liisa.mattila,kari.systa}@tut.fi

Abstract. Reuse of analysis methods and tools for data from different issue management systems is challenging because there are differences in how the data is accessed and represented. While various approaches for collecting and analysing software engineering data have been developed, they do not generally pay so much attention into how to actually get the data from various sources. This paper presents a combined model for issue management data that is based on an investigation of four issue management systems. It also presents a proof of concept tool that can collect issue management data from different services into our analysis and visualization framework using an API description language that defines how to get the issue management data and how to convert it into our model. The aim of this approach is to allow the addition of new data sources by simply providing their API descriptions.

Keywords: issue management, data model, software repository mining

1 Introduction

Issue management is an integral part of software development and management. Various tools have been developed for that purpose e.g. Jira and the issue tracking feature of GitHub. The information collected into the issue management system can be used to analyze the software project and it can give valuable insights, that can help in managing the project, e.g. how to automatically identify valid bug reports [12].

Many issue management systems offer an API that can be used to acquire data. A tool, that could fetch issue management data from multiple sources and offer the same interface and analysis features regardless what the data source is, would be useful for both its users and developers. There are some differences in how different issue management systems handle and represent issues and related concepts. Thus if the same analysis tools and notations are to be used to analyze issues from multiple sources, a common model for issue management has to be defined. Also a generic method for collecting issue data from different sources and converting it into this model has to be developed. This paper presents a combined data model based on four different issue management systems. It also

presents a tool that uses API descriptions in collecting and converting issue management data from different services.

This paper is organized as follows. Section 2 discusses the motivation and background of this research in more detail. Section 3 presents our investigation into different issue management systems. Section 4 describes the combined issue model that is based on the investigation. Section 5 describes the implementation of a issue collection tool that uses API descriptions in collecting issue management data according to the model. Section 6 presents discussion about our approach. Section 7 presents related work and section 8 presents conclusions.

2 Background and Motivation

An issue management system serves many purposes in an organization. It is a knowledge repository, a communication and collaboration hub and a communication channel for requests for new features, bug reports or any task that development team should perform [2]. Thus an issue management system contains different types of useful information for analysis. However, issue management systems are different in what data is stored and how the data is accessed. The organization's practices also affect how issues are used.

This research is related to previous research done in our department considering software engineering data analysis and visualization. In [9] we present a study where software engineering data from different data sources were combined and visualized to show realization of continuous deployment. This research has led to the development of a unified model for software engineering data and a framework for collecting, storing and accessing it [10]. This data can come from various sources and represent different domains such as issue management, version control or testing. To collect data from a specific domain an intermediate domain model can be used. This paper presents such a model for issue management data. Data can be first collected according to the intermediate model then converted into the higher level unified model.

In our previous work we have also developed a method for building Internet service compositions [7]. There we dealt with similar issues i.e. fetching conceptually similar data from different services, and using the data in a unified way to implement mash-ups. We developed the concept of generic data types that represented different concepts that many services handle like photo or status update. Then we added information about the generic data types to the service API descriptions so that they could be used in service compositions. In this work we want to use a similar approach for easily gathering issue management data from different services. We do not just want to write separate tools or plug-ins for fetching and converting data from different systems. Instead we want one generic tool that can be given descriptions of the APIs of the source systems. This would then make it much quicker to add different data sources.

3 Issue Management Data

To find a model for issue management data, we surveyed four web based systems that offer issue management. The systems were Jira[1], GitHub[2], GitLab[3] and Bitbucket[4]. Jira is a dedicated issue management system. GitHub, GitLab and Bitbucket offer code project hosting that include in addition to a issue tracker a code repository based on a version control system. We investigated what information a issue holds, what other concepts are related to issues and how the systems record changes and other activity related to the issues.

3.1 Properties

First we defined what information an issue contains, i.e., the properties of the issue. We listed the properties from each system and combined those that meant the same thing. Properties can be simple attributes or relations to separate entities, who have their own attributes. We found 26 different properties and 9 of these properties are common to all the systems. All issues have some kind of unique identifier, a title or summary and a longer explanation about the issue. The issue systems also record when an issue was created and when it was last updated. Issues also have a status or state that indicates the current phase in the workflow. Possible states in the workflow varies by system from just *opened* and *closed* offered by GitHub to the user customizable workflows of Jira.

All of the systems have authenticated users and they can be related to an issue as the creator. All systems support also assigning the issue to a user who then is responsible for progressing the issue's resolution. Issues can also be discussed in all systems with a commenting feature.

Two properties issue labeling and associating issues to milestones are shared with three of the systems. Seven properties are shared between two systems. In six cases those systems are Jira and Bitbucket. They let issues be categorized with types, offer possibility to associate issues with software versions and specific components. They also allow issues to be prioritized.

Jira is the most advanced of the systems. It offers eight properties that the other systems do not offer. It allows the type of the resolution to be recorded for example *fixed* or *cannot reproduce*. It also offers features for estimating and recording the amount of work for the issue. Issues in Jira can also be linked to related issues. In addition Jira is customizable offering a possibility to add custom fields.

3.2 Issue changes

All of the systems record changes to the issues such as changes in the issue state and properties. What changes are recorded and how they are accessed varies. In

[1] https://www.atlassian.com/software/jira
[2] https://github.com
[3] https://gitlab.com
[4] https://bitbucket.org

all of the systems the user who made the change and the time the change was made is recorded.

GitHub records issue changes as issue events. They can be accessed for the whole project or for a specific issue. These events have a type that indicates what kind of change the event represents for example *closed*, *opened*, *assigned* or *labeled*. The event contains also information about what the change was e.g. what label was added to the issue. GitLab has project specific events that include events about issues but have other events also. There are events only for issue opening and closing. Some other changes such as labeling or assigning are just saved as comments of the issue

Bitbucket has also project events. However the feature is limited since only 30 most recent are available. Their content is also quite limited. There are events for issue creation, commenting and updating but the update event does not have specific information about what was updated and the creation event does not tell what issue was created. In Jira each issue has a changelog. It records each change of the issue. A change record contains the property whose value was changed, its old value and the new value.

4 Data Model

This section presents the issue management data model that we developed based on our investigation of the issue management systems. It also shortly presents the unified software engineering data model used by our analysis and visualization framework, presented in [10], and its relation to the issue management model.

4.1 Issue Management Model

Our investigation shows that issues in the different systems have quite much in common. This enables the definition of a combined model for issues. It has eight different entities: *issue*, *user*, *comment*, *milestone*, *version*, *component*, *label* and *change event*.

An *issue* in our model has all of the properties presented in section 3.1. Most of them (18 out of 26) were shared at least with two of the systems and the rest are also useful. Table 1 lists the properties of an issue with their types and descriptions.

Label, *milestone*, *version* and *component* are similar simple entities. They have a name and a description and can be associated with issues. *Milestone* has also a due date, a creation time and a closing time. *User* represents a user of the issue management system. It can be associated with a *issue* as the issue creator and as an assignee. *Comments* consist of the comment message and the commenting time. They are associated with an *issue* and the *user* who posted the comment.

Each *issue* has a changelog. It consists of *change events* that record when the change was made, and optionally what property was changed and how i.e. what is the new value for the property. *Change event* is also associated with the *user* who made the change.

Table 1. The properties of an issue entity.

Property	Type	Description
id	string	An unique identifier for the issue in the management system
number	string	An unique identifier for the issue in a project that is not unique in the whole system
title	string	Describes the issue shortly
description	string	A longer explanation of the issue.
state	string	The current state of the issue in the issue workflow
author	user	User who created the issue
created	datetime	When was the issue created
updated	datetime	When was the issue last updated
assignee	user	The person who is responsible for the issue.
comments_count	integer	How many comments there are about the issue
comments	list of comments	Comments about the issue
change log	list of change events	Changes made to the issue
labels	list of labels	Tags that are used to categorize issues.
milestone	list of milestones	Used to categorize issues to be implemented in a specific version or sprint
priority	string	How important is the issue.
type	string	The type of the issue e.g. bug, feature
resolved	datetime	When was the issue resolved or closed.
affects version	version	The version the issue affects
fix version	version	The version in which the issue should be resolved.
component	component	The software component associated with the issue
watchers	integer	How many users are interested about the issue e.g. they get notified about issue changes
resolution	string	How was the issue resolved e.g. fixed, won't fix
environment	string	In what kind of environment the issue occurs
votes	integer	How many votes the issue has
due	datetime	When the issue should be resolved
estimate	integer	Original stimate of the time required to resolve the issue (minutes)
remaining	integer	Current time estimate (minutes)
logged	integer	How much work has been done for the issue (minutes)

4.2 Relation to Unified Software Engineering Data Model

The unified software engineering data model, mentioned in Section 2 and described in [10], consists of two concepts: *artifact* and *event*. *Events* present actions that happen in software engineering projects. They have an author, a type, the time the *event* happened and a duration. *Events* are related to *artifacts* the *event* happened to. *Artifacts* represent various aspects of software engineering that are interesting for visualization and analysis purposes. For example an *artifact* can be a file in version control and commits to that file are *events* related to it. *Artifacts* can also be related to each other. An *artifact* can have a state and it can be changed by an *event*.

The issue management model presented in the previous section works as a domain specific model for the unified model. It can be mapped to the unified model and so issue management data can be saved according to the unified model. Of the entities issue, milestone, label, component and version are *artifacts*. Change events and comments are *events*. Users can be modeled as *artifacts* or they can be just attributes for *events*. Additionally *events* can be generated from some of the entities' time based attributes. For example issues and milestones have an attribute that tells when they were created. From this attribute creation or opening *events* can be generated.

5 Implementation

This section presents the implementation of an issue collector tool that uses the issue model and the unified software engineering data analysis framework. First an overview of the tool is presented. Then its API description system is presented. Finally a usage example illustrates how the tool works.

5.1 Overview

Our web based data analysis and visualization framework [10] offers a database for the software engineering data and an HTTP API for storing and querying it. These APIs can be used by different data collection, analysis and visualization plug-ins.

To test the feasibility of the issue management data model and data collection approach presented in this paper a tool was developed that can fetch issue management data from different web based issue management systems according to the issue management model. The data collection involves making HTTP requests to various API endpoints like issues and milestones. The responses to these requests will contain lists of items in the JSON format that should be converted into various entities of the issue management model. After getting the data the tool then converts the issue management data into the format of the unified data model and sends it to the database.

The architecture of the tool is shown in Fig. 1. The tool consists of four components. *API descriptions* define how to get issue management data from

different sources. The *user interface* handles user input required for the data collection. The *API description* defines what input data is required. The *Collector* uses the user input to get data from a service described by an *API description*. The *Unifier* converts the data in to the unified model and sends it to the database.

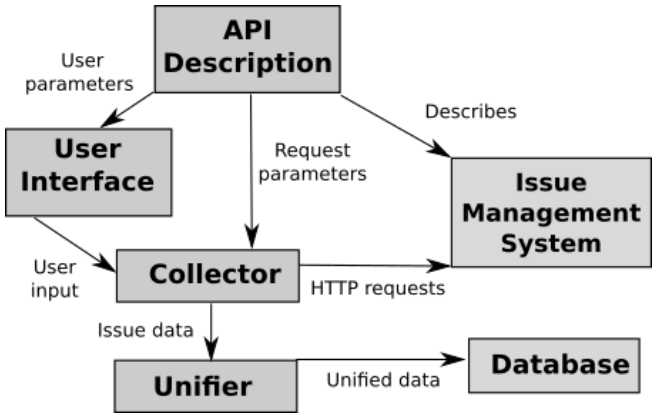

Fig. 1. The architecture of the issue collector tool.

The tool is implemented with Node.js. Currently the *user interface* is command line based. The current version of the tool does not yet cover the whole issue management model. It can process *issues*, *comments*, *milestones* and *change events*.

5.2 API Descriptions

An *API description* is a JavaScript object whose properties describe the API. The *API description* consists of general properties and resource descriptions. General properties describe general information about the API i.e. information that is common to all API calls. Resource descriptions describe information specific to API calls to one particular resource such as issues.

Table 2 lists the general properties. General properties for an API description include the common part of the API URL, possibly some HTTP headers and query parameters. Headers and query parameters are defined as simple objects containing key-value pairs where the key is the header or query parameter and the value its value. If the value is not static, the value undefined is used. This indicates that the value has to come from the user.

For defining what information is required from the user in the *user interface* the *API description* has an *userParams* property. This information is usually project specific information such as identification of project to be targeted. The value of *userParams* is a list of objects that contain the name of the parameter

Table 2. General API description properties.

Property	Type	Description
BaseUrl	string	The part of the URL that is common to all API requests
authentication	list of objects	A list of ways an user can authenticate to the service
pagination	string	How does the API handle pagination.
headers	object	HTTP headers required by every API call
query	object	Query parameters required by every API call
userParams	object	Defines what information is required from the user for issue management data collection
resources	resource description	Information about how to get and convert items from one API end point. Resources can be issues, milestones, changeEvents or comments.

and a description of the parameter that is shown to the user. The parameter name indicates where the value will be used. For example it can be used as a value for a header or query parameter that has the same name.

Issue management services can support multiple ways for their API users to authenticate. The *authentication* API description property lists the authentication methods that the service supports. A value in the list can be a string containing the name of an authentication method that the tool understands. Currently recognized methods are no authentication and HTTP basic. A value can also be an object that defines a custom authentication method which can contain additional headers, query parameters and user parameters.

Most API calls do not return everything at once. Instead they return up to a certain number of items and the client has to request more. The *pagination* property defines how this pagination in API calls is handled. Currently only pagination using a RFC 5988 link header, that has the pagination information, is supported.

The *API description* can contain multiple properties that have a resource description object as the value. The name of the property indicates what kind of entities the description describes. Currently supported values are issues, comments, milestones and changeEvents.

A resource description is an object whose properties describe a particular API resource, i.e., a concrete API end point that we want to make a HTTP call to. A resource could be for example the list of issues in a project or a list of one issue's comments. Table 3 lists the properties that a resource description can have.

The *path* property holds the rest of the HTTP request URL. The value is a RFC 6570 URI template whose variables have to be expanded before making the request. Values for these variables are found from the similarly named

Table 3. Properties of a resource description.

Property	Type	Description
path	string	The rest of the URL as an URI template.
query	object	Query parameters specific for this resource.
headers	object	HTTP headers specific for this resource
filter	function	Function used to decide if the current item will be processed.
item	item description object	How to convert one item from the resource into an entity in the issue management model
createOpeningEvents	bool	Create a change event from the created attribute of the new entity.
createUpdatingEvents	bool	Create a change event from the updated attribute of the new entity.
createClosingEvents	bool	Create a change event from the closed attribute of the new entity.
children	object	Contains resource descriptions of the current resource's child resources.
parentParams	object	Information required from a parent resource for getting a child resource.

user parameters. The *query* and *headers* properties are similar to the corresponding general properties but provide resource specific information. The *createOpeningEvents*, *createUpdatingEvents* and *createClosingEvents* properties indicate if additional change events should be created from the new entity's created, updated or closed properties.

The *filter* property can hold a function that is used to choose which items received from the issue management system are processed. The function is given a single item from the response like an issue and its boolean return value determines if that item should be processed.

The *children* property describes the current resource's child resources such as the comments of an issue. Its value is an object whose properties have resource descriptions as values similar to the general API description. The *parentParams* property is applicable only in child resource descriptions. Like user parameters its values can be used in the URI template, headers and query parameters but the source for the values is the child's parent entity.

The *item* property gives information on how to convert one item from the response in to an entity of the issue management model. The properties of an item description object correspond to the properties of the entity to be extracted. The value tells how to extract the value for the new entity's property. For describing how to extract the value we use JSONPath[5]. JSONPath expressions are used to select a specific part of a JSON document or JavaScript object. The value of an item description property can be a string or an object. The *path* property of that object holds the JSONPath expression. The *source* property tells where the

[5] http://goessner.net/articles/JsonPath/

Listing 1.1. A part of the GitHub API description. Most item descriptions are not shown and only part of the comment's item description is shown.

```
var api = {                                                                    1
   baseUrl: 'https://api.github.com/',                                         2
   authentication: [ 'no authentication', 'basic' ],                           3
   headers: { Accept: 'application/vnd.github.v3+json',                        4
         'User-Agent': 'ohylli/issue-collector' },                             5
   pagination: 'link_header',                                                  6
   userParams: [ { name: 'owner',                                              7
      description: 'The user name of the repository owner' },                  8
      { name: 'repo',                                                          9
      description: 'the repository name' } ],                                  10
   issues: {                                                                   11
      path: '/repos/{owner}/{repo}/issues',                                    12
      query: { state: 'all' },                                                 13
      filter: function ( item ) {                                              14
         return item.pull_request !== undefined; },                            15
      item: { ... },                                                           16
      createOpeningEvents: true,                                               17
      createUpdatingEvents: true,                                              18
      children: { comments: {                                                  19
            path: '/repos/{owner}/{repo}/issues/{number}/comments',            20
            parentParams: { number: '$.number' },                              21
            item: { id: '$.id',                                                22
            issue: { path: '$.id', source: 'parent' },                         23
            user: '$.user.login',                                              24
            message: '$.body', ... } } }, ... };                               25
```

value is to be extracted from. Possible values are item, which means the item received from the service, and parent, which means the parent entity of the new entity. The *mapping* property can be used to replace the extracted value with another value. If *source* is the item and there is no mapping information, the object can be replaced with a string containing the path information.

5.3 Usage example

As an example of the tool's usage we tested the method to collect issue management data from four public open source projects : grip[6], glutin[7], gfx[8] and webgl-noise[9]. The webgl-noise project is the smallest of the four projects containing 14 issues where as gfx is the largest containing 304 issues. Glutin project has 187 issues and Grip 107 issues. The projects use GitHub as a code repository and issue management system. Thus we require an *API description* of GitHub's API which is shown in listing 1.1.

When the issue collector is invoked, it first checks what *API descriptions* are present and asks the user which of these she wants to use. The issue collector loads the *API description* the user chose and first checks what authentication methods are available and lets the user choose the one she prefers. As can be seen on the line 3 of the Listing 1.1 GitHub issue collector can be used without authentication or with HTTP basic authentication.[10] If the user chooses basic authentication, the tool next asks the user for her username and password

[6] grip – https://github.com/joeyespo/grip/issues
[7] glutin – https://github.com/tomaka/glutin/issues
[8] gfx – https://github.com/gfx-rs/gfx/issues
[9] webgl-noise – https://github.com/ashima/webgl-noise/issues
[10] GitHub supports also OAuth2 authentication but our tool does not yet support it.

required by HTTP basic authentication. Next issue collector checks what additional API specific information is needed from the user. From the lines 7-10 of the Listing 1.1 we see that two user parameters named *owner* and *repo* are required. The issue collector queries inputs for these showing their descriptions to the user. Lastly the tool queries the user for some metadata required by the unified data model.

Next the *collector* can begin the actual data collection. It goes through every resource description, makes HTTP requests they define and converts the data received into the appropriate entities. For constructing the HTTP requests the *collector* gets the beginning of the URL from line 2. Lines 4 and 5 define that all HTTP requests have to contain two specific headers. If HTTP basic authentication was chosen the authentication information provided by the user is also added to the requests. Then, for example, from the resource description for issues the *collector* gets the rest of the URL from line 12. This URI template has two variables *owner* and *repo*. The *collector* gets values for these from the similarly named user parameters. The resource description also defines on line 13 that the URL has to include a query parameter named state with the value all. After making the request, the *collector* processes each item in the response. Since on lines 14-15 issues resource has a *filter* function, that is executed first and the item is processed only if it returns false. In this case the function is used to filter out pull requests which GitHub includes with the issues.

The *API description* defines on line 19 that issues have comments as children. This means that for each issue entity created its comments should be fetched as well. The *path* on line 20 is expanded with the owner and repo and also the number property of the parent issue. This is defined on line 21 with the *parentParams* property. The actual comment entity is constructed according to the information on lines 22-25. It defines for example that the message property of a comment can be found from the response item's property named body. It also defines that the id of the issue the comment is related to can be found from its parent entity's id property.

After each item in a response has been processed, the *collector* checks if the response contains a link header that has the URL for the next page of items, and if it does, it makes a request to it. This behaviour is specified on the line 6 of the Listing 1.1. After all entities are collected, the *collector* checks if additional change events have to be created. For example line 17 defines that from each issue entity a change event has to be created. This event's change type will be opened and the time the creation time of the issue.

After the *collector* is finished, the *unifier* converts the issue management data into unified model's artifacts and events which are then send to the database. After this the user can use the visualization framework's analysis and visualization features. Figure 2 has an example visualization from the grip project's data that shows each artifact's events and life spans on a timeline. From the visualization we can see for example how long different issues have been open and if the issue has been reopened. Also comment, label, reference and delabel times are visible

for each issue. This kind of view enables comparing issue lifespans to each other as well as finding similarities and patterns from issue events.

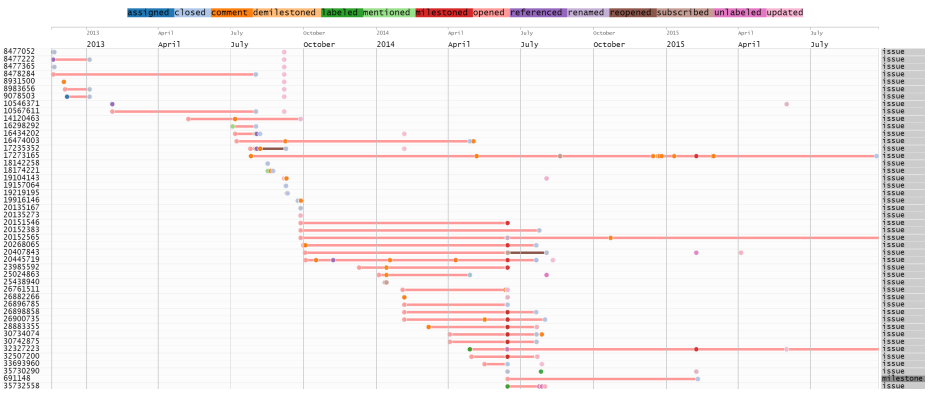

Fig. 2. Visualization of lifespans and events from Grip open source project. Each artifact has its own row to display events and lifespan. The lifespan starts when the artifact is opened and ends when it is closed. Lifespan is shown as a line in artifacts row. The dots mark different kinds of events. The dot colors are mapped to event types and those are explained in the top of the visualization. At the right of the artifact line the artifact type and at the left the artifact id are presented. All artifact rows are not visible in the figure as the figure is cropped to save space.

6 Discussion

Our issue management data model is based on a survey of four issue management systems. Although there are many more issue management systems we believe that our model covers the most important aspects of issue management. However in our future work we should verify our model by using it with systems that we did not survey and if the need arises to expand our model. Our model is quite simple and not as expressive as for example an ontology based approach. However, our aim was a light weight model for data storage and testing the API description approach, and for that purpose we believe our model is suitable.

The implementation of our issue collector tool shows the basic feasibility of the model. We used the model successfully with GitLab and GitHub for which we currently have API descriptions. The implementation and those API descriptions also shows the feasibility of our data collection approach. This approach has its strengths and weaknesses. The descriptions are declarative so a description author does not need to worry how the data is collected. When the APIs behave similarly such as GitHub and GitLab do when fetching issues and their comments, the approach works well. However, when there are differences in how things are done like with change events, the tool's implementation and API description have to take them in to account, which will cause complexity in the

implementation code and in the API description syntax. API rate limiting of the services can cause problems when fetching data from bigger projects and we must find ways to deal with them. Currently the tool is a proof of concept implementation and probably new issue sources such as Jira and Bitbucket could not be added just by adding their API descriptions since there are many things the implementation does not support yet. For example pagination is supported only with a link header which all services do not support so a custom pagination implementation would be required. More advanced data extraction features for more complex data structures are also required.

In our previous work on Internet service compositions [7] we used the Web Application Description Language (WADL) to describe the service APIs. Then we had to add additional metadata to describe the service and its data. In this work we wanted to try a different approach with our own JavaScript based API descriptions. It allowed us to combine the API description and the data description required in the data conversion. We could also add features that support common higher level tasks such as authentication and pagination. We can also add functionality to the API descriptions with JavaScript functions which we used in filtering the items. They could also be used for example with custom pagination implementations in the future. Though this approach requires the author of an API description to know JavaScript, the descriptions are quite simple and do not use advanced features of the language.

7 Related Work

On a high level this work can be seen to be related to research into extract, transform and load (ETL) processes used in data warehousing to integrate data from different sources for business reports. ETL research deals with similar problems as our research such as how to combine data from different schemas into a single schema and what is the workflow of the ETL process [11]. More precisely this work is a part of the research in to software repository mining where different tools for collecting and analyzing issue management data among other software engineering data have been developed. However these tools do not pay so much attention in making the data collection generic. Fischer et al. [3] developed a SQL based release history database for collecting and analyzing data for version control and issue management. The system does not include special features for data collection from different sources. Issue management data is just collected with custom scripts from Bugzilla.

Some approaches use semantic web technologies and define an ontology for software engineering data. Kiefer et al. [8] developed EvoOnt which focuses on software evolution. It includes models for the software, version history and bugs. The EvoOnt issue model is based on Bugzilla but it is similar to ours though there are some differences in what concepts of issue management are covered. The paper does not go much in to the details of the model like what properties and relations it supports or what if any change data is collected. This system also has no special consideration for data collection. Dhruv [1] offers semantically

enriched features for members of an open source community to work with issues and related information. Dhruv was developed for a particular open source community that uses particular tools though the developers point out that it could be made to work with other communities and tools, because its model should be general enough and its architecture supports expansion.

Evolizer [4] is a tool whose main focus is in analyzing code changes but its software metamodel includes issues and has an exporter for getting issue data from Bugzilla. It is an Eclipse plug-in and its extension including the addition of new issue data importers takes advantage of Eclipse's plug-in extension features. Goeminne and Mens [5] have developed a framework for analyzing and comparing the evolution of open source projects which is mainly focused on different metrics. It uses the FLOSSMetrics data base [6] which defines a schema for various data collectors including issue data collectors. The FLOSSMetrics issue data collector supports two issue management systems: Bugzilla and SourceForge.

8 Conclusions

Analysis of issue management data can give useful insights in to a software engineering project. In our previous work we had developed an unified software engineering data model and a framework for storing and accessing it. Collecting issue management data from various sources and converting it in to the unified format for analysis presents challenges. We tackled these challenges by first investigating four different web based issue management systems. Based on that we developed a combined issue management data model. It consists of eight entities such as issue, milestone and comment. These entities, their properties and relations cover the essential parts of issue management and allows data from various sources to be stored and analyzed.

We also developed a proof of concept issue data collection tool which collects issue data according to our issue management model and then converts the data into the unified data model's format. The tool uses declarative API descriptions which define how data is fetched and converted. The current version of our tool is limited but it proves the feasibility of our approach. The end goal of our approach is to allow new data sources to be added quickly just by providing an API description that can then be used to fetch data from different projects in that source. We believe that this approach can be expanded to cover different types of software engineering data such as version control data. In our future work we will explore the potential of this approach with other issue management systems and other kind of data. This approach might also have uses in other contexts.

Acknowledgments

The research has been supported by Tekes-funded Digile project Need for Speed[11] and by Foundation of Nokia Corporation[12].

[11] http://www.n4s.fi/en/
[12] http://www.nokiafoundation.com/

References

1. Ankolekar, A., Sycara, K., Herbsleb, J., Kraut, R., Welty, C.: Supporting Online Problem-solving Communities with the Semantic Web. In: Proceedings of the 15th International Conference on World Wide Web. pp. 575–584. WWW 2006, ACM, New York, NY, USA (2006)
2. Bertram, D., Voida, A., Greenberg, S., Walker, R.: Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In: Proceedings of the 2010 ACM conference on Computer supported cooperative work. pp. 291–300. ACM (2010)
3. Fischer, M., Pinzger, M., Gall, H.: Populating a Release History Database from version control and bug tracking systems. In: Proceedings of the International Conference on Software Maintenance. p. 23. ICSM 2003, IEEE, Washington, DC, USA (2003)
4. Gall, H.C., Fluri, M., Pinzger, M.: Change analysis with evolizer and changedistiller. IEEE Software 26(1), 575–584 (2009)
5. Goeminne, M., Mens, T.: A framework for analysing and visualising open source software ecosystems. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution. pp. 42–47. IWPSE-EVOL 2010, ACM, New York, NY, USA (2010)
6. Gonzalez-Barahona, J.M., Robles, G., Dueñas, S.: Collecting data about floss development: the flossmetrics experience. In: Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development. pp. 29–34. ACM (2010)
7. Hylli, O., Lahtinen, S., Ruokonen, A., Systä, K.: Resource description for end-user driven service compositions. In: IEEE 2nd International Workshop on Personalized Web Tasking (PWT 2014) (June 2014)
8. Kiefer, C., Bernstein, A., Tappolet, J.: Mining Software Repositories with iSPARQL and a Software Evolution Ontology. In: Proceedings of the 15th International Conference on World Wide Web. p. 10. MSR 2007, IEEE, Washington, DC, USA (2007)
9. Mattila, A.L., Lehtonen, T., Systä, K., Terho, H., Mikkonen, T.: Mashing Up Software Issue Management, Development, and Usage Data. In: Proceedings of RCoSE – 2nd International Workshop on Rapid Continuous Software Engineering (2015)
10. Mattila, A.L., Luoto, A., Terho, H., Hylli, O., Sievi-Korte, O., Systä, K.: Unified model for software engineering data. In: 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015) (September 2015)
11. Vassiliadis, P.: A survey of extract-transform-load technology. International Journal of Data Warehousing and Mining 5(3), 1–27 (2009)
12. Zanetti, M.S., Scholtes, I., Tessone, C.J., Schweitzer, F.: Categorizing bugs with social networks: a case study on four open source software communities. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 1032–1041. IEEE Press (2013)