Process mining software repositories

Wouter Poncin, Alexander Serebrenik, Mark van den Brand
Eindhoven University of Technology
P.O. Box 513, 5600 MB
Eindhoven, The Netherlands
w.poncin@alumnus.tue.nl, {a.serebrenik, m.g.j.v.d.brand}@tue.nl

***Abstract*—Software developers' activities are in general recorded in software repositories such as version control systems, bug trackers and mail archives. While abundant information is usually present in such repositories, successful information extraction is often challenged by the necessity to simultaneously analyze different repositories and to combine the information obtained.**

We propose to apply process mining techniques, originally developed for business process analysis, to address this challenge. However, in order for process mining to become applicable, different software repositories should be combined, and "related" software development events should be matched: e.g., mails sent about a file, modifications of the file and bug reports that can be traced back to it.

The combination and matching of events has been implemented in FRASR (FRamework for Analyzing Software Repositories), augmenting the process mining framework ProM. FRASR has been successfully applied in a series of case studies addressing such aspects of the development process as roles of different developers and the way bug reports are handled.

***Keywords*-Process mining, software repositories**

I. Introduction

Modern software development processes often involve multiple developers and development teams, sometimes residing at different continents and time-zones. Communication and coordination in such projects necessitates support by means of various kinds of software repositories, including mail archives, bug trackers and version control systems. A plentitude of data available in such repositories triggered an extensive research effort on mining software repositories [1], [2], [3], [4], [5], [6], i.e., automatic analysis of the development process based on such data.

In this paper we advocate applying process mining to analysis of data from multiple software repositories. Process mining [7], [8] aims at extracting information from event logs produced by an information system, in order to capture the business process supported by the information system. It has already been demonstrated to be a valuable technique for analyzing business processes in various domains [8], [9]. Process mining makes a clear separation of the event log preparation [10] from the event log analysis [7]. Therefore, once data from multiple software repositories has been translated to an event log suited for process mining, a wide range of process mining techniques becomes applicable. This sharply contrasts with many existing repository mining applications [4], [5] that tightly couple the preprocessor to the analyzer, i.e., the analyzers cannot be reused or combined.

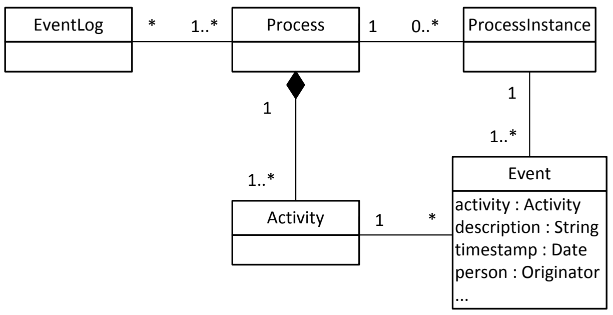

Figure 1. Process mining meta model, modified from [11].

In order to be amenable for process mining, the event logs should conform to the process mining event log meta model [11] (see Figure 1[1]). An *event log* contains data from a number of *processes* (usually one). Each process has a number of *process instances* that can be uniquely identified. Furthermore, each process has a number of *activities*, and each *process instance*, also known as *case instance*—a number of *events*, consisting of an activity being executed at a certain moment in time and associated with certain data. For instance, a log of an insurance company might contain information about a billing process and a refund process. A refund process has a number of process instances uniquely identified by the claim number. Activities that should be executed in the refund process may include registering the claim, and checking the insurance policy. An example of an event is "On Thursday September 23, 2010 Alice checks the insurance policy of the persons involved in claim 478-12". Process mining aims, therefore, at discovering the information about a process, based on the information about different process instances. Based on a log conforming to the meta model, process mining techniques can, for instance, derive abstract representations of the process control flow, detect relations between the individuals involved in the business process and their tasks, and infer data dependencies between different process activities.

The application of process mining techniques to repository mining, requires the information from the software repositories to be preprocessed first. The preprocessing

[1]As process mining terminology was evolving, different papers use different names for notions as process, event, etc. In our choice of names in Figure 1, we have tried to achieve clarity and consistency with terminology used in the research on mining software repositories.

step should not only combine the data from different sources, but also produce a log conforming to the process mining meta model (Figure 1). To create such a log, the following challenges have to be addressed:

- While for many business processes the appropriate notion of case is the "natural way" to associate different events, e.g., customer number associates all events pertaining to the same customer, this is not necessarily the case for software artefacts.
- While traditional logs state the events explicitly, this is not always the case for software repositories: e.g., one has to analyze the contents of an e-mail to decide whether it represents an event relevant for a given file.
- Finally, multiplicity of software repositories leads to different representations of the same information in different repositories: e.g., user names' of the bug tracker might be very different from those found in the mailing archive. Therefore, such representations should be matched: for instance, developer names should be matched and bug reports in the bug tracker should be ideally linked to code modifications caused by resolving bugs, as reflected in a version control system. While modern systems like Subclipse (`http://subclipse.tigris.org`) allow to link bug reports and code modifications, most of the time these links are not available [12].

Once the preprocessing step has been completed, the resulting log can be analyzed using existing process mining application, such as ProM [7].

The remainder of the paper is organized as follows. After discussing the challenges identified above in Section II, we present our prototype implementation, called FRASR `http://www.frasr.org`, in Section III. Application of the tool in a series of case studies is discussed in Section IV. Discussion of the related work in Section V and conclusions in Section VI close the paper.

II. Applying process mining to software repositories

In this section we discuss how the three challenges identified in the introduction can be addressed. We start by presenting different ways in which the case can be defined, proceed to event extraction and conclude with how different representations of the same information can be matched. As the running example in this section, we consider information from a Subversion version control system. One should note however, that similar case definitions and event extraction techniques can be applied to other software repositories as well. Additional examples and ways of defining cases and event extraction techniques, are discussed in [13].

A. Defining case

The main challenge pertaining to the application of process mining to software repositories is related to the notion of case: many software repositories do not express this notion explicitly. This means that the application of process mining to a software repository requires an explicit preprocessing step allowing to identify related events within or across different repositories, i.e., to form a process instance. In our running example, process instances can be formed by all commits in the Subversion log, commits performed by a *developer* or commits affecting a software *component*. Alternatively, a process instance can be formed by commits that took place on a certain *date* or during a certain *period*. It should be noted that the way the case is defined greatly influences which analysis techniques are applicable. Having, for example, the component as the process instance allows the user to combine the data from different software repositories pertaining to this component; having the developer as the process instance allows to combine the data from different software repositories pertaining to this developer, etc.

B. Extracting events

As opposed to logs used for traditional process mining, logs produced by software repositories often do not specify the events explicitly. Therefore, we have to introduce *event bindings* as a mean to specify the way the events should be extracted from the data of the software repositories. For every event extracted, we need to determine to which activity the event belongs, when the event took place and what data is associated with the event.

Basic event bindings: Software repositories can be seen as lists of uniform entities, e.g., commits in the running example. Basic event bindings retrieve these entities from the software repository and create a separate event for each one of them. For all the events extracted, the corresponding activity is derived from a property of the repository. In this way we can distinguish between events originating from different categories of software repositories (e.g., version control system vs. a bug tracker), different types of repositories belonging to the same category (e.g., a Subversion repository and a Git repository) and different repositories of the same type (e.g., a Subversion repository for system version 5 and a Subversion repository for system version 6).

Detailed event binding: While basic event bindings provide means to associate events with coarse-grained properties of software repositories, sometimes a more fine-grained labeling is desired. The detailed event binding allows a user to specify for every node in the hierarchy of the data fields of the software repository, whether to use it, and how to derive the activities. In the Subversion data field hierarchy, a repository consists of a list of commits. Each commit contains the commit-level information such as 'commit-id', 'author', 'timestamp' and 'commit message', and a list of files added, deleted, modified or renamed. Each modified file has a list of modified text fragments associated. Hence, the data field hierarchy contains three levels: commits, modified files and modified text fragments. The detailed event binding allows one to indicate for each level whether to use the corresponding information to extract the events. Moreover, if the information from a certain level is used, one should be able to select the data fields required to identify the activity. For instance, the detailed event binding can be used to get for each file in a Subversion repository (used as a process instance) the modifications that were performed on it (as events).

Type-specific event bindings: While the preceding groups of event bindings can be defined for any type of software repositories, type-specific event bindings are specific for a given repository type. For example, an event binding specific for Subversion might label the commits as predominantly additions (*VCS: A*), deletions (*VCS: D*), modifications (*VCS: M*) and renamings (*VCS: R*).

Category-specific component event bindings: An other type of event bindings specific for repositories of a given category are component event bindings. These event bindings determine the activities based on the activities associated with software repositories of a given category, e.g., for version control systems, components can be derived by applying a regular expression to the file path: */trunk/src/com/company/application/component/Interface-Implementation.java* is matched to *component*.

C. Matching different representations

Different software repositories might involve different representations of the same entity. For instance, developers might use different names in the mail archive and in the version control system; while specific versions of a document stored in the version control system are often encoded as a part of the document name (SRS-1.0), mail messages are likely to refer to them using their generic name (SRS). Moreover, developers might be identified by their real names, aliases or mail addresses. Therefore, one should be able to indicate cross-repository links: either manually or automatically, e.g., as suggested in [14], [15].

III. FRASR

We have implemented the preprocessing techniques discussed in Section II in a prototype called FRASR (FRamework for Analyzing Software Repositories). FRASR is a quite small application consisting of 173 Java files amounting to 20,301 source lines of code (SLOC) and 20 more form files implementing the user interface. A simplified view on the architecture of FRASR is shown in Figure 2. FRASR is aware of different *data sources* that can be associated with projects: e.g., the aMSN case study discussed in Section IV-B has associated a version control system, mail archives and bug trackers. In general, FRASR supports 12 commonly used data sources: Subversion, Bugzilla, Trac tickets, JIRA bugs, SourceForge bugs, mbox archives, SourceForge mails, Piper mail archives, MARC mail archives, Tigris mail archives, SourceForge forums and Trac wiki articles. Each data source might contain more information than needed to answer a specific software engineering question: to this end *filters* can be applied. Moreover, each data source mentions *developer aliases* that can be associated with a single *developer*. Using simple heuristics, FRASR can automatically construct the developer matching, an important form of cross-repository link specification, based on the developer aliases from the data sources in the project. The heuristics gives more weight if it is more likely that two aliases refer to the same developer. For example, the weight corresponding to the complete mail address such as *psmith@example.com* is at least as high as the weight corresponding to the username in an other software repository, such as *psmith*. As the automatically constructed developer matching may contain false positives and false negatives, the user can manually correct this matching. Finally, to create (export) the event log, *case* and *event bindings* have to be defined.

In Figure 2, abstract classes (typeset in italics) indicate extension points: FRASR can be extended to add new cases, event bindings, data sources and export types.

As indicated, answering different questions with respect to the same software repositories might require redefinition of the case and/or event bindings, and therefore, recalculation of the event log. In order to speed up this process as well as to support incremental analysis, FRASR also includes a cache. Caching significantly reduces the time needed to preprocess the log: for instance, preprocessing 7223 version control system revisions, 3349 bug reports and 4256 mails of the WinMerge project (`http://www.winmerge.org`), FRASR needs 10 minutes instead of 71 minutes on a 32bit Windows 7 machine with an Intel Core2 Quad CPU 2.40 GHz with 3GB of memory. Similar improvement figures were obtained for other projects such as PhpMyAdmin (`http://www.phpmyadmin.net`) with 13465 revisions, 7694 bugs, 49696 mails and 10065 topics from the SourceForge forum: on the same machine the analysis time was reduced from 588 minutes to 45 minutes [13]. Further time reduction can be achieved by filtering data from the software repositories. FRASR allows the user to specify that, e.g., only bugs reported after January 17, 2010 should be included in the resulting log.

The log produced by FRASR can be exported either as a comma-separated list or in the MXML format, supported by the ProM process mining workbench (`http://www.processmining.org`). ProM contains more than 170 mining, analysis, monitoring and conversion plugins. Mining plugins aim at discovering a process model for a given log: e.g., the Alpha algorithm detects a control flow, while the Fuzzy Miner, used in Section IV-C, is suited for mining less-structured, flexible processes. Analysis plugins aim at providing insights in correctness or performance of the process reflected in the log. For instance, Dotted Chart Analysis, used in Section IV-B, shows a spread of events of an event log over time. The added value of FRASR consists, therefore, in providing the user with a wide palet of successful techniques for repository mining. In the next section we will see how the combination of FRASR and ProM can be applied in practice.

IV. Evaluation

In this section we report on the application of FRASR in a number of real-life cases. We start by presenting a

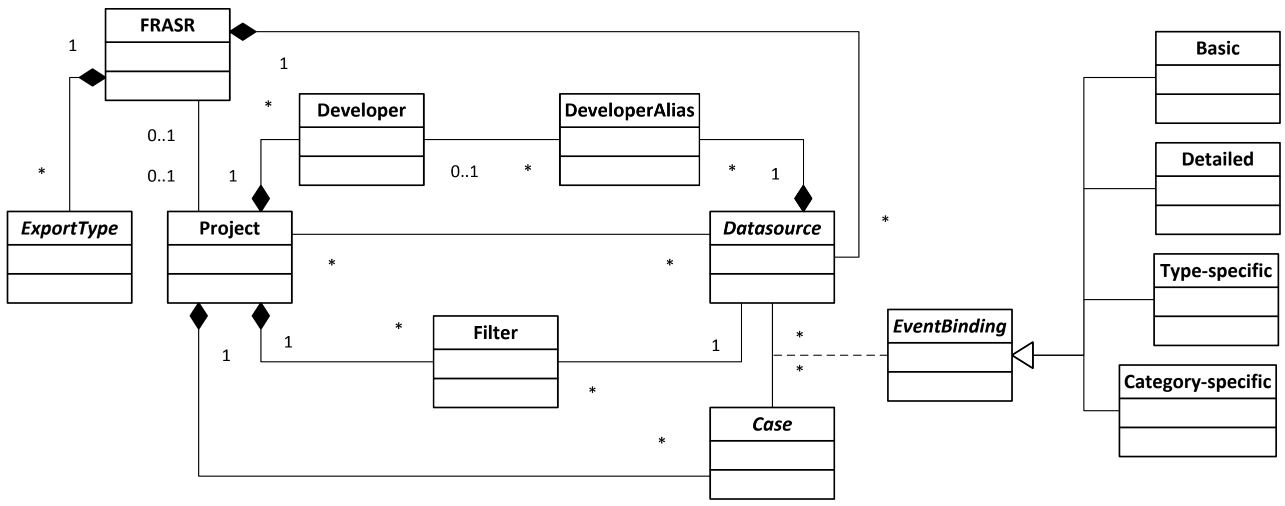

Figure 2. Architecture of FRASR

methodology, following the general guidelines of [16], and then discuss two case specific case studies.

A. Methodology

The objective of the case studies is to demonstrate how software engineering questions can be answered using FRASR. While the software system and the software engineering question considered in the case studies are different, the same procedure presented in Figure 3 has been followed. We started by *defining data sources*, i.e., providing FRASR with information about software repositories. In our studies we have considered open-source projects: an instant messaging application aMSN, and the GNU compiler collection (gcc). In both cases appropriate repositories were available on-line, so data source definition consisted of entering the URL and authentication information. To ensure no ethical issues were involved, we have opted for software systems with which none of the authors has previous experience. Next, a *case mapping* and *event bindings* have been defined. To match developers across different repositories FRASR calculates the *developer matching*, and finally, it *exports* the data as an event log. The log has been imported to ProM. Finally, ProM-assisted analysis has been carried out providing an answer to the initial software engineering question.

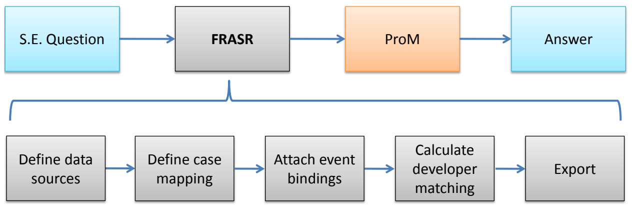

Figure 3. Using FRASR for answering software engineering questions.

B. Case study 1: developer roles

The goal of this case study is to demonstrate the added value of using the combination of multiple software repositories compared to using multiple data sources separately. We show that when using the 'combined' data from multiple software repositories, we can get insights which cannot be obtained by using only data from individual software repositories in isolation. As case study we have chosen the problem of classifying developers in open source software projects to roles. Such a classification is necessary to analyze the social relationships between the community members and the relationships between the roles.

1) Classification rules: We follow the classification of Nakakoji *et al.* [17]. While alternative classifications have been proposed, e.g., in [18], Nakakoji *et al.* provide a detailed description of the roles.

- **Project leader** is, as defined in [17], often involved from the beginning of the project. She is responsible for vision and overall direction of the project. Therefore, we require a project leader to be involved from the beginning of the project and to contribute to its core base, i.e., she should have added or modified files in the version control system. A complementary approach to project leader identification proposed in [19] is based on the number of *inbound* mail messages. This approach is, however, no longer applicable when mailing lists are used as all messages are being addressed to all list subscribers. The number of *outbound* responses might reflect helpfulness or technical expertise rather than actual leadership.
- **Core member** is a developer which has been involved in the project for a relatively long period of time. Nakakoji *et al.* state that core members should have made "significant contributions to the development and evolution of the system" [17]. As we have studied the repositories from February 26, 2002 until July 9, 2010, we have opted for a continuous period of at least thirty-six months as the threshold for "relatively long period of time". Moreover, core member's contribution to the development

and evolution of the system should be significant. Therefore, we require such a developer to have more revisions in the version control system than average (of all the developers having events in the version control system). Moreover, core members must have added files to and modified files in the version control system.

- **Active developers** regularly contribute new features and fix bugs, i.e., during their activity period they should have *Ticket-closed*, *VCS: A* and *VCS: M* events. We interpret "regularly" as at least one event every month. Furthermore, to distinguish active developers and core members we require the continuous activity period of active developers to be shorter than thirty-six months.
- **Peripheral developer** are those with sporadic or irregular contribution to the system functionality. To characterize these developers we require them to have both *VCS: A* and *VCS: M* events, but no longer require the events to be present every month.
- **Bug fixer** is a developer that fixes bugs discovered either by herself or by a different developer. This means that the bug fixers should have *Ticket-closed* and *VCS: M* events, and should not belong to the other categories.
- **Bug reporter** is an open-source counterpart of a tester. Bug reporters discover bugs and report them, but do not modify the code. Bug reporting occurs either using a bug tracker or a mailing list: samples of the mailing list messages in a number of projects have been inspected, confirming that the mailing lists are indeed often used for reporting bugs. Hence, bug reporters have *Ticket-created* events or *Mail thread created*, but no events related to version control.
- **Reader** goes beyond using the system and inspects the code to understand how the system works.
- **Passive user** is attracted by the functionality of an open source system, but does not attempt to contribute to it.

As activities of readers and passive users are often not reflected in software repositories, we do not consider these classes further. Indeed, the only information available about the behavior of passive users is the number of times the executables were downloaded, but as registration is often not a prerequisite for downloading we cannot distinguish between different passive users. Similarly, registration-free access to the source code does not allow to distinguish between different readers. Therefore, we only consider the categories: bug reporter, bug fixer, peripheral developer, active developer, core member and project leader.

2) System under investigation: We have chosen to study aMSN, a free and open source instant messaging application, clone of Windows Live Messenger. Unlike Windows Live Messenger, aMSN supports Macintosh and UNIX/Linux in addition to the Windows platform. At the moment of writing aMSN has been downloaded more than 38 million times, making it 20th most popular SourceForge project of all times. To analyse aMSN we have considered seven bug repositories (bugs, feature requests, patches, plugins, skins, support requests and translations), three mail archives (commits, devel and lang) and one Subversion repository located at `https://amsn.svn.sourceforge.net/svnroot/amsn/`. We have focused on the period from February 26, 2002 until July 09, 2010. In total, the repositories contained 3137 bug reports, 34947 mail messages and 12062 revisions. The aMSN project also has a discussion forum. However, the data of this forum cannot be used in the current implementation of FRASR.

The data from the software repositories has been exported using the developer case and the data source specific binding for each data source. The developer matching has been calculated automatically using the simple heuristics mentioned in Section II-C. Furthermore, we assume that the time stamps are synchronous, i.e., when time stamps from both repositories are equal, the points in 'real time' they were recorded, do not differ significantly.

3) Results: Using the exported log in combination with the ProM Dotted Chart visualization and a spreadsheet application, the developers were assigned to one of the available roles. Figure 4 presents a part of a Dotted Chart visualization, used in the classification. Green dots correspond to mail events such as *Mail thread created* and *Mail reply*, black to *Ticket-created*, red to other bug tracker events, blue to addition of files in the version control system, and finally white to other events of the version control system (modifications, deletions and renames). The size of dots represents a number of events occurring in the same week and color mixture corresponds to events of different kinds occurring in the same week.

By inspecting Figure 4 we clearly see that the developer in the first line is represented by a sequence of overlapping white, red and blue dots, starting at the very beginning of the project. According to the classification rules above this developer (*Alvaro J. Iradier Muro/airadier*[2]) will be classified as the project leader. Furthermore, we observe that some developers are represented by long sequences of overlapping dots, as is, for instance, the case for *Alaoui Youness/kakaroto* and *Boris Faure/billiob*. These developers are core members of the project. Shorter sequences represent active developers, such as *Arieh Schneier/lio_lion* and *Tom Jenkins/bluetit*. Finally, disconnected dots are characteristic for the sporadic activity of peripheral developers. This is, for instance, the case for *Harry Vennik/thaven*.

Visual inspection of Figure 4 provides for qualitative results. Additional quantitative results have been obtained by exporting the relation between the developers and activities, expressed by a so called ProM "originator by activity matrix", to the spreadsheet application and performing simple counting. In this way, out of 1725 developers we have identified 1443 bug reporters, 3 bug

[2]The developer names are derived from the information of the associated developer aliases. This includes for example a username and a name associated to an email address.

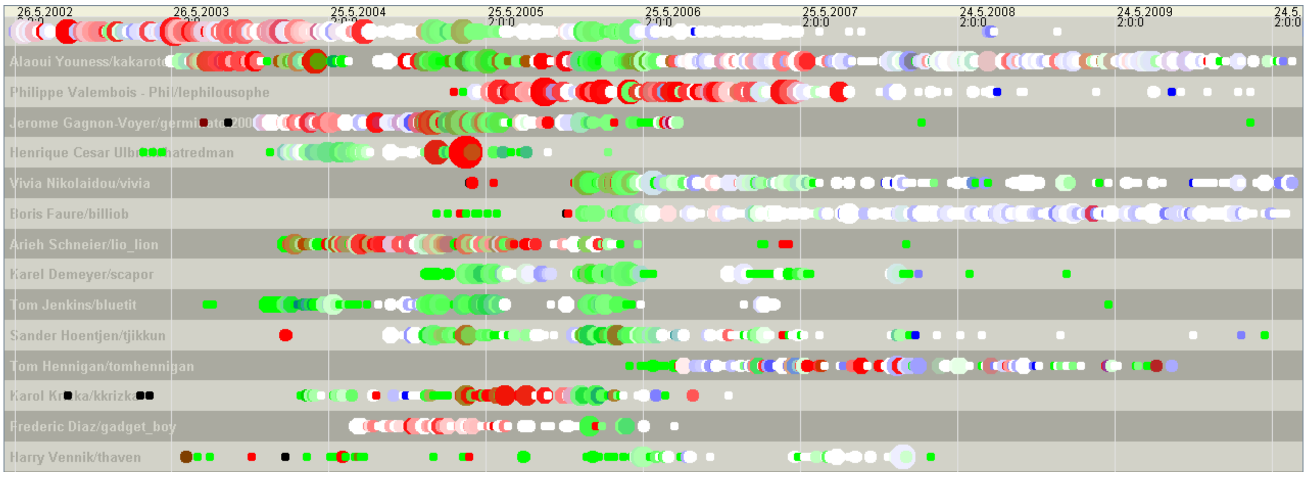

Figure 4. Dotted Chart visualization of the top-15 developers of aMSN, sorted by number of events. Color legend: Green: *Mail thread created* and *Mail reply*. Black: *Ticket-created*. Red: *Ticket-closed*, *Ticket-commented*, *Ticket-reopened*. Blue: *VCS: A* (file added). White: *VCS: M* (file modified), *VCS: D* (file deleted) and *VCS: R* (file renamed)

fixers, 29 periferal developers, 6 active developers, 7 core members and 3 project leaders. Moreover, 234 developers cannot be classified according the rules above: these are, e.g., developers having only *Ticket-commented* or *Mail reply* events (6 max), anonymous developers and the SourceForge robot commenting on and closing tickets.

4) Conclusions: The case study has revealed a number of shortcomings of the classification of [17]. First of all, the classification considers the entire development process as a whole, while developers clearly interleave periods of high and low activity (see, e.g., the behavior of *Karel Demeyer/scapor* in Figure 4). To address this problem, the classification can be refined to take only shorter periods of time into account. The refined classification can be further supported by the filtering capabilities of FRASR (Section III). Furthermore, we have observed that such classification criteria as "responsibility for vision and overall direction of the project" are difficult to formalize, and therefore, to assess objectively.

Compared to the general structure of open source software communities (as presented in [17]), our results exhibit a relatively low number of bug fixers. We conjecture that access to the Subversion repository is granted only to relatively active project members, i.e., bugs are fixed by project members classified as peripheral developers.

From the methodological point of view this case study illustrates that software engineering questions that cannot be split to sub-questions pertaining to individual repositories can be answered using FRASR and ProM combination due to the ability of FRASR to combine information coming from different software repositories. Identification of bug fixers provides an example of such a software engineering question.

Validity of the conclusions above could have been affected by a number of threats. Construction validity pertains to the way the study has been conducted: we have, for instance, not considered the discussion forum of aMSN that might have been used for bug reporting. In this way we could have misclassified some developers. To compensate for this we did not distinguish between different kinds of mail archives and considered, e.g., the "translations" mail archive as relevant for bug reporting. As we do not aim at generalizing our conclusions to additional developer classification approaches or software systems, no threats to external validity are present.

C. Case study 2: bug life cycle

The purpose of this case study is demonstration of more advanced mining features becoming applicable to software repositories. We discuss the life cycle of a software artefact and show that FRASR enables a process mining technique well-suited to identify popular and exceptional sequences of events as reflected in the log. As an example we consider the life cycle of bug reports in Bugzilla.

1) Life cycle of bugs in Bugzilla: According to the Bugzilla Guide [20] the bug reports follow the flow presented in Figure 5. However, bug reports typically do not 'visit' each possible state (e.g., bug reports marked as a duplicate are immediately removed). We investigate whether the actual bug reports life cycle corresponds to Figure 5.

2) System under investigation: To study the bug reports life cycle we have chosen GCC, the GNU Compiler Collection, containing front ends as well as libraries for such programming languages as C, C++, Ada, Java and Fortran. As GCC is well-known for a big and active users community, we expected numerous bug reports to be filled in, and hence, the bug reports life cycle constructed based on the GCC data will also show less frequent transitions between the bug report states.

In our study, we focused on bugs recorded in the GCC Bugzilla, located at `http://gcc.gnu.org/bugzilla`, and reported from January 1, 1999 till January 31, 2010. During this period 42373 bugs have been reported. The number of comments and updates corresponding to these bugs equal 495321. In total 13944 users have been involved in bug reporting and discussion.

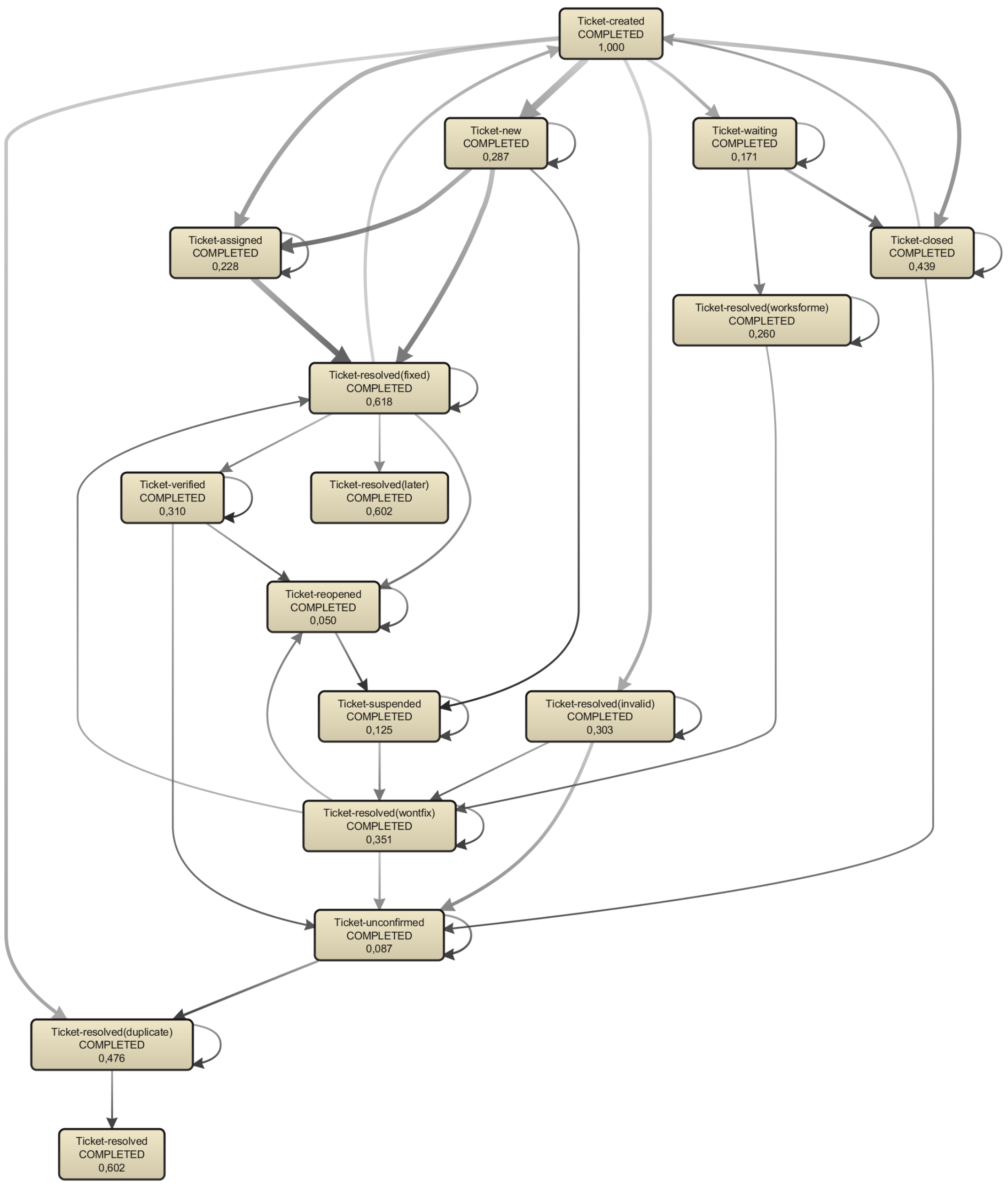
Ticket-created COMPLETED 1,000
Ticket-new COMPLETED 0,287
Ticket-waiting COMPLETED 0,171
Ticket-assigned COMPLETED 0,228
Ticket-closed COMPLETED 0,439
Ticket-resolved(worksforme) COMPLETED 0,260
Ticket-resolved(fixed) COMPLETED 0,618
Ticket-verified COMPLETED 0,310
Ticket-resolved(later) COMPLETED 0,602
Ticket-reopened COMPLETED 0,050
Ticket-suspended COMPLETED 0,125
Ticket-resolved(invalid) COMPLETED 0,303
Ticket-resolved(wontfix) COMPLETED 0,351
Ticket-unconfirmed COMPLETED 0,087
Ticket-resolved(duplicate) COMPLETED 0,476
Ticket-resolved COMPLETED 0,602

Figure 6. Bug life cycle Fuzzy Graph, extracted from the GCC Bugzilla repository.

To get insights in the actual transitions between states of the bug reports as opposed to the prescribed ones, we have used FRASR in combination with the ProM Fuzzy Miner plugin. The event log was produced by FRASR using the data field case on the bug-id's, in combination with the data source specific (Bugzilla) binding. The events named *'Ticket-commented'* and *'Ticket-updated'* have been filtered out, as these are not present in Figure 5 and make the figure unnecessarily complex.

3) Results: Figure 6 presents the Fuzzy Graph corresponding to the bug reports life cycle in the GCC Bugzilla repository. We observe that Figure 6 distinguishes between different kinds of ticket resolution with resolution "later" being absent from Figure 5. Moreover, Figure 6 introduces two states *'Ticket-waiting'* and *'Ticket-suspended'* absent from Figure 5.

The thickness of an arrow in Figure 6 represents the percentage of process instances (bug reports) making the transition to that state. Figure 6 shows that most of the bug reports are either immediately resolved (*'Ticket-created'*, *'Ticket-new'*, *'Ticket-resolved(fixed)'*) or successfully resolved after one or more assignments (*'Ticket-created'*, *'Ticket-new'*, *'Ticket-assigned'*, *'Ticket-resolved(fixed)'*). Several other paths however are present in the graph, including an unexpected recreation of tickets, indicated by arrows entering *'Ticket-created'*, as opposed to *'Ticket-reopened'*.

4) Conclusions: Summarizing the results above, we observe that the official bug reports life cycle as presented in Figure 5 provides a simplified view on the way bug reports

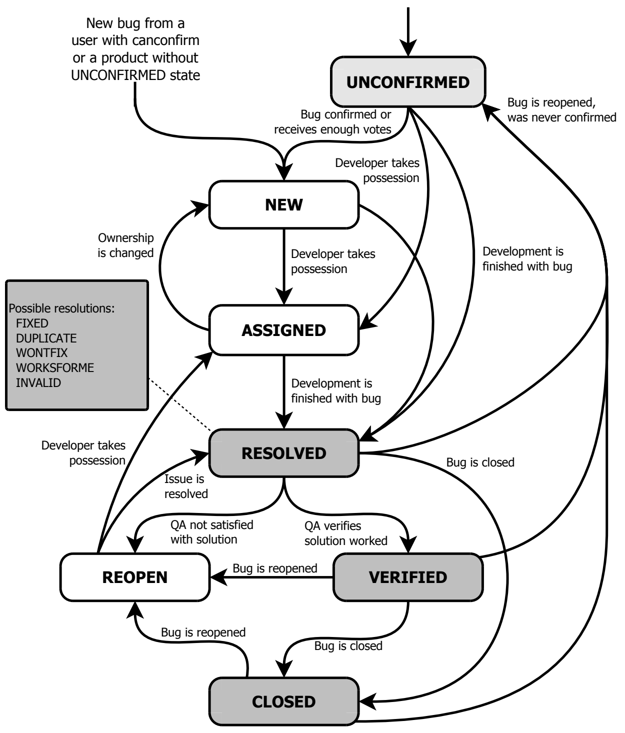
New bug from a user with canconfirm or a product without UNCONFIRMED state
UNCONFIRMED
Bug confirmed or receives enough votes
Bug is reopened, was never confirmed
Developer takes possession
NEW
Ownership is changed
Developer takes possession
Development is finished with bug
Possible resolutions: FIXED DUPLICATE WONTFIX WORKSFORME INVALID
ASSIGNED
Development is finished with bug
Developer takes possession
RESOLVED
Bug is closed
Issue is resolved
QA not satisfied with solution
QA verifies solution worked
REOPEN
Bug is reopened
VERIFIED
Bug is reopened
Bug is closed
CLOSED

Figure 5. Bugzilla bug life cycle according to [20].

are handled in practice. We expect that less frequent states or transitions might be absent in smaller open source projects, but are likely to be present in the larger ones, comparable with GCC in size and popularity.

Validity of the conclusions above might have been affected by our choice of the case study: however, we have explicitly chosen a larger open source project in order to highlight less frequent scenarios as well. Unlike traditional approaches we do not limit our attention to "completed cases", i.e., cases that end with one of the activities denoting bug report being closed. By doing so we increase the collection of process instances to include open bug reports, and observe transitions that might have been absent in completed cases. Furthermore, relations between the expected bug report life cycles, as reflected in diagrams of other bug trackers, and the way these bug trackers are used in practice should be a subject of a separate study.

V. Related Work

Application of process mining to software repositories was considered in [3]. This approach focusses on deriving process models from data in software repositories. While the component case/binding in FRASR is similar to the one used in [3], FRASR also supports means to map data from, e.g., mail archives to activities. ProM Import [10] is a preprocessor similar to FRASR. Unlike FRASR however, ProM Import cannot combine information from different sources. Moreover, it has been designed for business processes rather than software repositories. Therefore, it misses essential functionality such as matching developers and configuring the case definition.

Beyond process mining, mining software repositories has attracted significant attention from the research community: CVSgrab [1] supports gathering and visualization data from CVS repositories. Using CVSgrab, a user can answer questions like 'What is / was the development process?' and 'What are the main contributors and their responsibilities?'. Other tools like the eROSE [2] and ProjectWatcher [6] plugins in Eclipse, assist developers in finding related artefacts. Hipikat [5] is similar to ProjectWachter, but uses data from multiple software repositories (e.g., Subversion, Bugzilla). Using Hipikat, a developer new to a project can quickly become familiar with the project group memory. However, as these tools are geared towards a single developer, they are not very well suited for analyzing the development process of the project. Similarly, Alitheia Core [4] can calculate metric values based on multiple software repositories, while softChange is a fact enhancer and a visualizer supporting data import from mail archives, Bugzilla repositories and CVS repositories. All these tools gear the analysis towards specific visualizations. Unlike them, our approach separates the preprocessing step carried out by FRASR and the analysis step, and hence makes multiplicity of mining and analysis techniques readily available. In the case studies above, we have used ProM to carry out the analysis but commercial process mining tools such as Futura Reflect (`http://www.futuratech.nl`) could have been used for this purpose as well.

Matching identities from various sources is a nontrivial task, as the identities can be usernames, e-mail addresses, real names, etc. In [15], Robles and Gonzalez-Barahona present an approach for matching developer aliases from various software repositories. In this approach, identities are constructed by using information from the sources (like an e-mail address or a username). For example, the name and surname can be extracted from an e-mail address like *name.surname@example.com*; a technique also applied in FRASR. They also use GPG keys (which contains a list of e-mail addresses a developer may use for encryption and authentication purposes) and other information related to the developers. Bird *et al.* propose a technique more specific to matching e-mail addresses [14]. In this approach they use the Levenshtein edit distance between (parts of) e-mail addresses to determine the similarity.

VI. Conclusions

In this paper we advocate applying process mining techniques to mining software repositories. We have identified the challenges that should be addressed to enable this application, discussed how they can be addressed and presented FRASR, our prototype implementation. Unlike existing approaches to repository mining, the approach proposed makes clear separation between the preprocessing step and the analysis step, fostering reuse of analysis techniques.

Case studies (coming from different domains), have shown that process mining in combination with FRASR leverages valuable insights in software development processes. The flexibility provided by FRASR allows to combine information from software repositories depending on the question the analyst tries to answer: for aMSN cases were based on the developer names and for Bugzilla on bug identifiers.

As *future work* we consider developing new analysis techniques, tool extension and empirical studies. We plan to study integration of process mining techniques with software metrics, specifically recent approaches to metrics aggregation [21]. Furthermore, we will investigate what kind of process mining techniques are required to address additional software engineering questions. As a tool, FRASR should be extended with components allowing it to parse information from decentralized version control systems such as Mercurial and GIT, forums such as phpBB Forum as well as from micro-blog message systems such as Twitter. We also plan to integrate in FRASR more advanced artefact and deloper matching techniques such as [15]. To support a more interactive form of analysis such as visual analytics [22], FRASR should be extended to support quick navigation to elements in the data sources, such as a single bug report or an email message, where the events originated from. As empirical studies we further consider application of FRASR + ProM combination to a controlled study of student projects, on the one hand, and on a larger scale to such open source projects as Debian and KDE, on the other. Moreover, we intend to apply FRASR + ProM to social network analysis using the corresponding ProM mining plugin and single point of failure detection using the so called "originator by task matrix".

REFERENCES

[1] L. Voinea and A. Telea, "Mining software repositories with CVSgrab," in *Int. Workshop on Mining Softw. Repositories*. ACM, 2006, pp. 167–168.

[2] T. Zimmermann, V. Dallmeier, K. Halachev, and A. Zeller, "eROSE: guiding programmers in Eclipse," in *ACM SIGPLAN Conf. on OO programming, systems, languages, and applications*. ACM, 2005, pp. 186–187.

[3] V. Rubin, C. W. Günther, W. M. P. van der Aalst, E. Kindler, B. F. van Dongen, and W. Schäfer, "Process mining framework for software processes," in *Softw. Process Dynamics and Agility*, ser. LNCS, vol. 4470. Springer, 2007, pp. 169–181.

[4] G. Gousios and D. Spinellis, "Alitheia core: An extensible software quality monitoring platform," *Softw. Eng., Int. Conf. on*, pp. 579–582, 2009.

[5] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 446–465, 2005.

[6] C. Gutwin, R. Penner, and K. Schneider, "Group awareness in distributed software development," in *ACM Conf. on Computer supported cooperative work*, 2004, pp. 72–81.

[7] B. F. van Dongen, A. K. Alves de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst, "The ProM framework: A new era in process mining tool support," in *Int. Conf. on App. and Theory of Petri Nets*, ser. LNCS, vol. 3536, 2005, pp. 444–454.

[8] W. M. P. van der Aalst, H. A. Reijers, A. J. M. M. Weijters, B. F. van Dongen, A. K. Alves de Medeiros, M. Song, and H. M. W. Verbeek, "Business process mining: An industrial application," *Inf. Syst.*, vol. 32, no. 5, pp. 713–732, 2007.

[9] A. Rozinat, I. S. M. de Jong, C. W. Günther, and W. M. P. van der Aalst, "Process mining applied to the test process of wafer scanners in ASML," *Trans. Sys. Man Cyber Part C*, vol. 39, no. 4, pp. 474–479, 2009.

[10] C. W. Günther and W. M. P. van der Aalst, "A generic import framework for process event logs," in *Workshop on Business Process Intelligence (BPI 2006)*, ser. LNCS, vol. 4103. Springer, 2006.

[11] B. F. van Dongen and W. M. P. van der Aalst, "A meta model for process mining data," in *Conf. on Advanced Information Systems Engineering*, vol. 161, 2005.

[12] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Int. Conf. on Softw. Maintenance*. IEEE, 2003, p. 23.

[13] W. Poncin, "Process mining software repositories," Master's thesis, Eindhoven University of Technology, aug 2010. [Online]. Available: http://alexandria.tue.nl/extra1/afstversl/wsk-i/poncin2010.pdf

[14] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Int. Workshop on Mining Softw. Repositories*. ACM, 2006, pp. 137–143.

[15] G. Robles and J. M. Gonzalez-Barahona, "Developer identification methods for integrated data from various sources," in *Int. Workshop on Mining Softw. Repositories*. ACM, 2005, pp. 1–5.

[16] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, pp. 131–164, 2009.

[17] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *Int. Workshop on Principles of Softw. Evolution*. ACM, 2002, pp. 76–85.

[18] K. Crowston and J. Howison, "The social structure of free and open source software development," *First Monday*, vol. 10, no. 2, 2005.

[19] M. Weiss, G. Moroiu, and P. Zhao, "Evolution of open source communities," in *OSS*, ser. IFIP, E. Damiani, B. Fitzgerald, W. Scacchi, M. Scotto, and G. Succi, Eds., vol. 203. Springer, 2006, pp. 21–32.

[20] "The Bugzilla guide—2.18.6 release," oct 2006, "6.4. Life Cycle of a Bug". [Online]. Available: http://www.bugzilla.org/docs/2.18/html/lifecycle.html

[21] A. Serebrenik and M. G. J. van den Brand, "Theil index for aggregation of software metrics values," in *Int. Conf. on Softw. Maintenance*. IEEE Computer Society, 2010.

[22] M. G. J. van den Brand, S. A. Roubtsov, and A. Serebrenik, "SQuAVisiT: A flexible tool for visual software analytics," in *CSMR*, A. Winter, R. Ferenc, and J. Knodel, Eds. IEEE, 2009, pp. 331–332.