Software Process Recovery using Recovered Unified Process Views

Abram Hindle, Michael W. Godfrey and Richard C. Holt
David Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
{ahindle,migod,holt}@swag.uwaterloo.ca

***Abstract*—The development process for a given software system is a combination of an idealized, prescribed model and a messy set of ad hoc practices. To some degree, process compliance can be enforced by supporting tools that require various steps be followed in order; however, this approach is often perceived as heavyweight and inflexible by developers, who generally prefer that tools support their desired work habits rather than limit their choices. An alternative approach to monitoring process compliance is to instrument the various tools and repositories that developers use — such as source control systems, bug-trackers, and mailing-list archives — and to build models of the de facto development process through observation, analysis, and inference. In this paper, we present a technique for recovering a project's software development processes from a variety of existing artifacts. We first apply unsupervised and supervised techniques — including word-bags, topic analysis, summary statistics, and Bayesian classifiers — to annotate software artifacts by related topics, maintenance types, and non-functional requirements. We map the analysis results onto a time-line based view of the Unified Process development model, which we call Recovered Unified Process Views. We demonstrate our approach for extracting these process views on two case studies: FreeBSD and SQLite.**

I. Introduction

A custom more honour'd in the breach than the observance.

– Hamlet, I:iv

In principle, software development processes offer a means to ensure reliable and predictable results when creating and modifying software systems. While there are a number of well known processes and process frameworks — and the research literature on this topic is extensive [1], [2], [3] — companies commonly follow a "home-brew" development process that mixes broad and well known process practices with narrower and more specialized ones that are dictated by the particular demands of their business model.

Anecdotal evidence suggests that even these customized process models are often not followed very closely in practice. Consequently, if a manager wants to carefully track what actual process steps her developers have been following, she must either interview the developers or analyze the observable results of their efforts.

The advantages of interviewing are numerous: one can ask both general, wide-ranging questions as well as directed questions about specific areas of interest; one can adapt the line of enquiry as new information is revealed; and the replies are usually rich in detail, often providing additional context to the interviewer that she may have been unaware of. However, interviewing also has many disadvantages: it is labour intensive and time consuming for both the interviewer and interviewee; it is hard to automatically extract and organize the large amounts of details that typically result; developers may view the interview as adversarial, and may be unwilling to cooperate truthfully; and developers may have a mistaken view of what process steps they actually followed. Additionally, stakeholders other than team managers may wish to be able to examine the development process of a software system. For example, new team members might desire a way to learn about a project and its development culture without bothering their colleagues; also, regulating agencies and potential corporate partners may wish to be able to scrutinize development practices as part of a due diligence effort. In these cases, the stakeholders may have limited or no direct access to the developers in question, and semi-automated approaches may be the only practical option.

The advantages of process recovery through semi-automated artifact analysis include that it is largely automated, it is relatively easy to gather the needed data — assuming that it is available in the first place — and that, apart from validation, it does not require access to the developers. However, there are also disadvantages: it can be hard to meaningfully link data from disparate sources; there are often large semantic gaps in the available knowledge; and even if results of one study seem promising, it is unclear how generalizable they may be.

This paper explores the extent to which semi-automatic analysis of development artifacts can be used to extract the processes used by the developers. Our goal is to be able to map artifacts to process-oriented activity models such as the UP time-line diagram shown in Figure 1. The artifacts of interest we consider in this paper include changes to source code and documentation, bug tracker reports and events, and mailing-list messages. We analyze these artifacts to extract indicators of use and behaviour. We also perform signal similarity analysis to determine when behaviour within a repository significantly changes. And finally we perform a topic analysis of the artifacts; that is, we attempt to characterize the intent or purpose of the artifacts by comparing the natural language used within them against a benchmark glossary of general software engineering concepts, such as non-functional requirements. By analysis and presentation of the underlying focus, topics, and behaviour we provide a historically accurate view of the software development processes, as illustrated in Figure 3.

Our contributions include:

- a proposal of a methodology for process recovery,
- a proposal of a high-level process visualization called the Recovered Unified Process Views (Figure 3), that is based on UP time-lines (Figure 1), and
- a case study of FreeBSD and SQLite using these techniques.

A. Motivation

Figure 1 depicts the time-line of development activities of an idealized project through the lens of the Unified Process (UP). We call this kind of model an UP diagram; it is a well known and elegant visualization of the multi-dimensional nature of iterative software development. In this diagram, we can see a mixture of simultaneous behaviours and workflows that span different development disciplines. Since a diagram like this is effective at communicating a prescribed process, we conjecture that it will be useful to describe the actual underlying and observed process. In our approach to semi-automatic recovery of a project's process, we create diagrams that are similar to these UP diagrams.

A persistent problem with this kind of approach is that the activities within the various UP disciplines may not be easily observable by simply monitoring available development artifacts. For example, project requirements may be managed by a group external to the development team, and key but undocumented design decisions may be made informally between developers in face-to-face meetings. These holes in the record present a challenge to meaningful analysis: if we cannot observe all activities of the ongoing development effort, how reliable are our results likely to be, and what can we do with them? One thing we can do is to use the events that we *can* observe to try to predict when undetected events might have occurred. For example, discussions about APIs and the modification of an API might be a result of the design and analysis discipline. Even then there are disciplines within the UP that may not have directly or indirectly observable events. With our approach we model what we can observe and infer, including activities that the process designers may not have considered explicitly. For example, we can focus quality assurance (QA) and the non-functional requirements related to those qualities, such as reliability or portability. These concerns are not an explicit part of the UP, yet may be of interest to stakeholders.

II. Previous work

Our work leverages research primarily from the mining software repositories (MSR) community [5], which often focuses its mining efforts on version control systems, bug-trackers, and mailing-lists.

The focus of our work concerns software development processes. Unfortunately, the term "process" is overloaded in this research field, so we must take care to distinguish development processes from *stochastic* processes and *business* processes.[1] Stochastic processes and time-series have been used to explore the laws of software evolution [6], [7]. For example, Herraiz et al. [8] studied these processes in depth by mining many software projects and studying their software metrics; they found that metrics used to measure growth usually followed Pareto distributions.

Business processes are closer to software development processes because they concern sequences of related tasks, activities, and methods that are combined to achieve some business goal, such as the handling of product returns at a retail store. Van der Aalst et al. [9] describes business process mining as the extraction of business processes at run-time from actual business activities. Van der Aalst deferred to Cook and Wolf [10] when it came to applying process mining on software projects. Software development is a kind of information work, like research, and thus is not easy to model so formally.

Software development processes are meant to ensure quality and provide a reliable framework for the development of software projects. Some processes address a specific part of software development, such as maintenance [11]. The software development life-cycle (SDLC) [3] describes how software is often built, maintained, supported, and managed. Most software development processes relate to some if not all of the various aspects of the SDLC. Software development processes are often posed as a methodology related to development, such as the waterfall model [1] and the spiral model [2]. More recent software development processes include the Unified Process [4], (see Figure 1 for a diagram of the Unified Process disciplines over time), Extreme Programming (XP) [12], SCRUM [13], and many methodologies related to Agile development [14]. Most of the recent processes focus on incremental development and smaller iterations, so that design and requirements can be updated as they become clearer over time. Software development processes and life cycles seek to manage the creation and maintenance of software. Meta-processes, such as the Capability Maturity Model (CMM) [15], attempt to model the processes used to create software, much like ISO standard 9000 [16] attempts to model and document how processes are executed, tracked, modelled and documented. The CMM concerns modelling, tracking, and ranking software process adherence.

Our work on non-functional requirements [17] that we extract from software artifacts is based on Ernst et al.'s [18] work and is also related to Cleland-Huang et al.'s [19] work on mining requirements documents for non-functional requirements (NFR). Cleland-Huang used keywords mined from NFR catalogues [20]. With respect to the MSR community Mockus and Votta [21] leveraged WordNet and word-bag approaches to discover commits that dealt with security topics. Treude et al. [22] produced ConcernLines, a visualization and

[1] In this paper, unless otherwise indicated the reader should assume that the term "process" refers to a software development process.

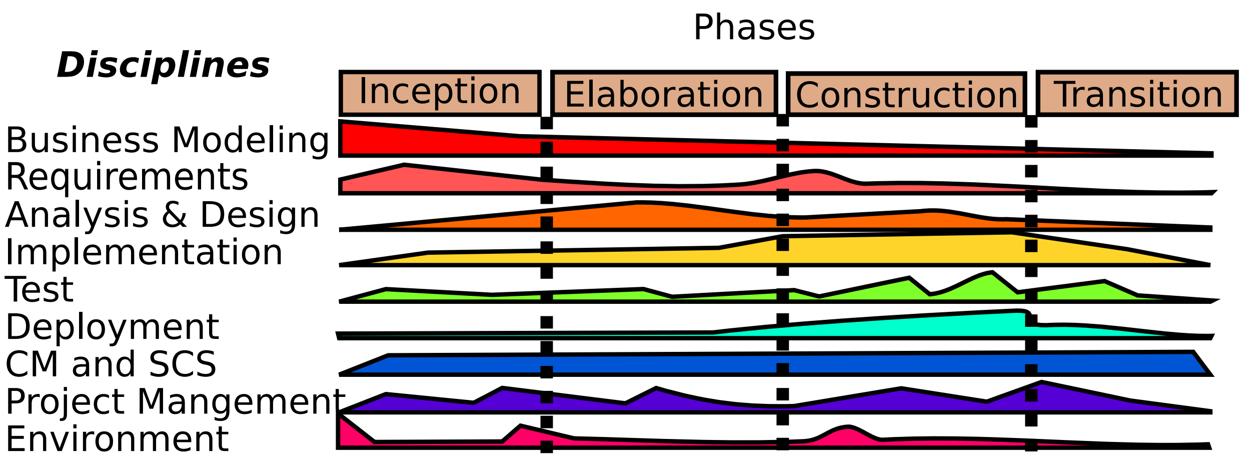

Fig. 1. Unified Process diagram: this is often used to explain the division of labour within the the Unified Process [4]

methodology that mines manually created tags from software in order to present views of the software history and processes.

All of this work is relevant to our attempts to produce Recovered Unified Process Views (RUPV) where we need to track changes, topics of changes, and discussions over time.

III. Methodology

Our purpose is to produce reports on activity within repositories and to produce diagrams like the UP diagram. Our recovered version of a UP diagram is called Recovered Unified Process Views (RUPV). RUPVs are views in that there are many possible perspectives from which to analyze and view the software development history and processes of a software project.

Our overall methodology can be broken down into seven steps as illustrated in Figure 2: acquisition, extraction, unsupervised analysis, annotation, supervised analysis, signal mapping and reporting. Our methodology flows from acquisition and extraction to supervised analysis and feeds back into unsupervised analysis or transitions into signal mapping and reporting.

Acquisition is the discovering and mirroring of relevant repositories of data and software artifacts. Often these repositories need to be mirrored in order to avoid affecting performance of the development environment or to ensure that the repositories are archived.

Extraction extracts data from the artifacts collected during the acquisition step. The type of data (and its meta-data) depends on the repository being extracted. In Section III-B we will discuss each kind of repository that we extracted for this paper, but the most important information we are looking for is creation and change events (revisions) of software artifacts. We can get this kind of information from source control, mailing-lists and bug trackers. Depending on the kind of analysis used, we might need partial or entire artifacts: source code analysis might require the full source code but natural language processing-style (NLP-style) analysis might only need word counts of textual data.

Unsupervised analysis is the analysis of the extracted data and events, generally without the help of the end user, without annotation. Automatic methods are used in this step. Unsupervised analysis ranges from summary statistics and modelling, to NLP-like analysis and word-bag analysis, to topic analysis.

Annotation is used to enhance the unsupervised analysis by clarifying information such as classification decisions, modifying stop words, or modifying word-bag dictionaries.

Supervised analysis includes methods that require some form of human intervention such as tuning training sets or labelling commits. We use supervised analysis to label topics and classify revisions by their maintenance categories. Supervised techniques often employ machine learning based classifiers that require annotated training sets.

Signal Mapping and reporting takes previous analyses and presents them as consumable Recovered Unified Process Views. Signals are combined to produce process related measures that summarize the underlying observable processes.

The rest of this section will detail each of these steps and demonstrate how they fit together.

A. Source Acquisition

We are interested in development events that we can extract from a wide variety of data-sources, including source code changes and bug tickets. The UP diagram illustrates how a software process might consist of parallel efforts, such as requirements and business modelling, that are related to multiple sources of data and multiple kinds of artifacts.

Depending on the project, such information might not be available. Even when artifacts, such as requirements documents, are unavailable, there still may be data that can act as its proxy. Sources are particularly important when they relate time to creation events or change events. A source can be useful even if it is difficult to analyze or assess, the simple count of an event occurring can suggest that some effort was taken with respect to a certain kind of task. These counts of events of a task could be further aggregated into larger disciplines like those Figure 1.

A series of events can be abstracted as a signal. Signals can be extracted from meetings, design document revisions,

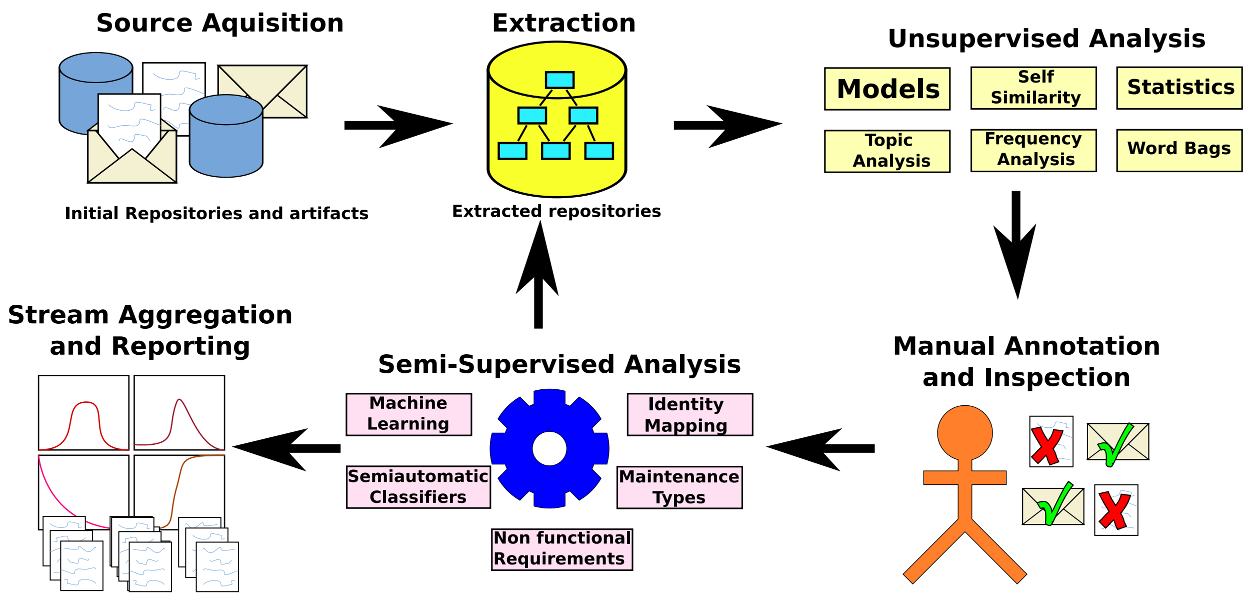

Fig. 2. Methodology flow chart, describes how artifacts are extracted, analyzed, annotated and reported about.

revisions to artifacts, requirements revisions, releases, milestones, deliverables, story cards and story card completions, even diagram revisions. Our two case studies in this paper rely on public data, so we are limited to available data sources such as version control systems, mailing-lists, and bug-trackers.

Ideally during acquisition we could also mine developer documents for information about mirroring or cloning the repositories of a project, such as mailing-list archives or version control systems. Tools like CVSup, cvssuck, svnadmin and Git can be used to mirror version control repositories.

The acquisition step is meant to ensure that these data sources are acquirable, thus it is useful to automate this process to allow for analysis. Once acquisition of sources is complete the data must be abstracted by the extraction step before they can be analyzed further.

B. Extraction

The extraction step attempts to abstract the data from those sources collected during the acquisition step. In this step we extract events, their data, and information about when they occur. This means we have to be able to convert the raw data into a format that is usable by our various tools. This extracted data will need further analysis in order to produce usable signals. Extractors that could be useful include CREX [23], CVSAnalY [24], SoftChange [25], MLStats [24], as well as our own extraction tools. We have many kinds of repositories that we can extract various kinds of development artifacts from:

1) Version Control Repositories: For CVS data we use tools such as CVSup to mirror the data, and we use SoftChange and CVSanalY to extract it into a database format that allows us to easily query authors, files, revisions, and commit messages. In the case of CVS, commits are not recorded and need to be rebuilt from file revisions. To extract FOSSIL repositories, used by SQLite, we wrote our own extractor.

2) Mailing-lists: While tools for analyzing mailing-lists such as MLStats exist, we wrote our own mailing-list analysis tool called MBoxTractor[2]. The data we extract from mailing-lists includes: authors, direct receivers, people mentioned within the body of the message, message content, and the content without quotes. Email address normalization should also be employed to help consolidate identities.

A particular project might have multiple mailing-lists dedicated to different issues such as user support, or developer discussions. These mailing-lists can be combined or separated per project. A developer mailing-list might be more relevant to development while traffic on a user mailing-list might be more relevant to end-users of the software.

3) Bug trackers: The bug tracker is a convenient source of process related information because interaction with the bug tracker is relatively formal, well recorded and annotated. Some bug trackers such as *gnats* or *FOSSIL* often are missing data of interest such as identity of the bug reporter or dates of certain events. Bug trackers are especially valuable because their bug identifiers are referenced in other repositories like the version control system. For example, the bug identifiers assigned to a ticket or a bug are often referenced on mailing-lists and in repository commit messages.

Bug trackers also may serve as a source of requirements data. For example, some bug reports are marked as feature requests and are sometimes used to discuss and flesh out requirements. Bug reports are often rich with many timestamped events. We used MBoxTractor, our own bug extractor, on the reported bugs of FreeBSD and SQLite.

4) Traceability Links: Traceability links are meta-artifacts, that is they are references between artifacts. An example traceability link would be a bug ticket number embedded

[2]Bug and email and FOSSIL extractor: http://softwareprocess.es/MBoxTractor/

in a commit log message. Traceability links highlight cross-repository aspects of the development process as information flows between repositories.

5) People: People are the ultimate traceable artifacts within a repository as they are creators of artifacts and changes. They are referenced by artifacts across repositories as they show up in most of the repository meta-data, in the source code copyright statements, in comments, and mentions in mailing-lists. If a person interacts with multiple repositories it could indicate what kind of role they play in a project. For example, someone who participates on a user mailing-list may be only a user. If they are referenced within the version control system they might be a developer or they might have contributed a patch.

Unfortunately people are not generally represented as entities within a repository so they need to be identified. Even once identified many contributors use more than one email so their identities must be consolidated.

These repositories and entities discussed in this section will be used as input for a more thorough analysis in the next step: unsupervised analysis.

C. Unsupervised Analysis

The Unsupervised analysis step occurs after extraction and is done automatically without the user's help. Unsupervised analysis often consists of partitioning data, decision making, and classification. This goes beyond extracting entities. The following subsections deal with the various kinds of supervised analysis we used in this paper.

1) STBD revisions: In order to break revisions down by their purpose, we track four main file-type revisions, which we refer to as STBD [26]: Source code changes, Test code changes, Build system changes, and Documentation changes. These kinds of revisions are relatively simple to track and extract with reasonable accuracy. To classify files into any of the four types we compare the filename against sets of regular expressions associated with each STBD type. STBD partitioning is useful as it allows us to break down a stream of revision events into semantically distinct streams.

2) Word-bags: Word-bag analysis is the matching of dictionaries of related terms to lexical tokens extracted from sources of natural language text. An example of a word-bag is the *requirements* word-bag which includes words such as requirements, specification, feature, and use case. If a document, such as a change, uses one of these words it might be labelled as a document related to requirements.

3) Topic Analysis: Topic analysis [27] tries to find independent development topics in commit log comments. Topic analysis utilizes tools such as latent Dirichlet allocation (LDA) or latent semantic indexing (LSI) to automatically find development topics. The topics produced are much like word-bags, but these are automatically extracted from the commit log messages and often represent development issues, such as bug fixing, that are addressed during development.

Unsupervised analysis can be enhanced by annotation of word-bags and stop words. The next step, annotation, can employ unsupervised analysis to create training sets for supervised analysis.

D. Annotation

The annotation step occurs after the unsupervised analysis step and serves to enhance the results of the unsupervised analysis. As well annotation serves as a method to prepare training sets for supervised learners.

Stop words, such as "the" and "with", are used to eliminate common words that obscure automatic results. Sometimes common words will dominate results and it might be necessary to remove them from the data in order to enhance the results. In the annotation step one should investigate the topics generated by topic analysis and remove stop words that are not useful in distinguishing topics from each other. For instance if a version control system was automatically updated by another service, such as a vendor's version control system, the words used in the automatic update might show up in most of the topics. The removal of these words might enable more meaningful topics to be extracted.

The lexicon of a project is often unique to itself [28]. Word-Bags should be tuned appropriately. For instance, if the word *optimize* or *optimizer* refers to a module rather than the performance of a project it might be best to avoid false classifications by removing those terms from the performance word-bags [18], [17].

Supervised analysis such as topic labelling or maintenance classification needs training sets. During the annotation step training sets can be fashioned out of previous unsupervised analyses like word-bag analysis. These training sets will be fed into the next step, supervised analysis.

E. Supervised analysis

Supervised analysis requires some human intervention, provided during the annotation step. These supervised methods can often produce better, more project relevant results than unsupervised methods. Two kinds of supervised analysis we use are topic labelling and maintenance classification.

1) Topics Labelled by NFRs: Topic labelling [17] attempts to label topics extracted during the unsupervised analysis step with non-functional requirements (NFRs) such as: efficiency, functionality, reliability, usability, maintainability and portability. These topics are labelled based upon the words they are composed of. We use a Bayesian classifier in order to label topics by their NFRs, thus a training set of labelled topics is needed to train the learner. By labelling topics by NFRs we can attempt to characterize some of the quality-oriented processes being followed within a software project.

2) Maintenance Classification: Maintenance classification tries to classify commits to the version control system by their maintenance categories based on their commit log messages [29]. The purpose is to characterize the maintenance based aspects of the underlying development process.

The maintenance categories that we use are based upon an extended version of Swanson's maintenance categories: corrective changes that fix bugs, adaptive changes dealing with the run-time environment and portability, perfective changes meant to improve maintainability or efficiency, feature additions, and non-source-code changes such as copyright statements. This is all done with a machine learning based classifier that requires a training set of commits labelled by their maintenance categories.

After our supervised analysis is complete we can either feedback into unsupervised analysis and annotation again or we can move on to aggregating and reporting our results.

F. Signals and reports

After completing the unsupervised analysis, the annotation and supervised analysis, we are ready to report the results. Results can be summarized textually, but our focus is to produce Recovered Unified Process Views (RUPVs) that are similar to the UP diagram in Figure 1.

For each set of events, such as commits, bug report messages, and mailing-list messages, we have signals of their events over time. We may choose to partition these signals by author interaction, by file type, or by their description.

For each set of events — commits, bug reports, and mailing-list messages — we extracted word-bag derived signals for NFRs and software engineering terms. The word-bags included efficiency, maintainability, reliability, functionality, portability, usability, requirements, analysis, deployment, and project management.

We combined signals into new signals meant to proxy or simulate UP diagram disciplines such as business modelling, requirements, analysis, implementation, testing, deployment, configuration, project management and support environment. These signals are explained in more detail in Section V.

Other signals we have access to are tags from version control systems and releases. Releases are an especially valuable signal because they indicate a process driven event (the release) has occurred.

These signals and results can then be presented to the end user as described in Section V. Next, in Section IV, we describe methods of visualizing and analyzing some of these signals.

IV. Signals

We can produce signals (measures over time) from events by counting them or measuring them over time. We can deal with signals in many ways, ranging from bucketing to windowing, to moving averages, to visualization and comparison, as we will now explain.

A. Signal Visualization

Generally to produce RUPVs we want to create parallel plots of signals across time, i.e., multiple signals plotted in a stacked fashion with a common time-line. Sometimes a signal may be too noisy to be useful so moving averages, windowing, or bucketing can be used to clean up the signal. In the cleaned up signal, we may be able to see trends in the data that would otherwise be obscured.

B. Signal Similarity

If we are analyzing process-oriented signals, we might want to see when a signal correlates with another signal or when a signal abruptly changes its behaviour. We can look for these patterns by comparing a signal to itself using self-similarity. Self-similarity is when we take periods (or regions) of the signal and compare it to other periods of the signal. We can use the same techniques to compare two signals as well.

We use two methods of signal similarity comparison, global and local. Global comparison correlates the local bins to all bins across time. The local comparison correlates the local bins to bins nearby within a certain threshold, such as an iteration or a three month window.

The comparison metric we use here is the Euclidean distance between feature vectors consisting of summary statistics. Euclidean distance was chosen as it accommodates any kind of extracted feature vector. We could use statistical measures like X^2 or Kolmogorov Smirnov or frequency-based comparisons such as auto-correlation or Fourier transform comparison [30].

To create a similarity signal, we produce a signal of the rank of the most similar periods to the current period, which in a stable system is expected to be almost constant or linear. This kind of presentation of a signal is useful because if the similarity abruptly changes it indicates a kind of discontinuity in behaviour.

We can combine all these different signal analysis methods to produce different views for the RUPVs described in the next section.

V. Recovered Unified Process Views

Recovered Unified Process Views are views of repository data that provide an overview that is similar to the UP diagram in Figure 1. These views can be created by plotting signals of development, extracted from the repositories. Whether the signal consists of counts of commits, mailing-list messages, bug reports, or relevant subsets of such artifacts, the default view is to display signals as parallel time-lines of activity.

RUPVs are intended to be analogous to the original UP diagram but instead of proposing a process they display data that is observable and extracted from the repositories used.

A. Mapping signals back to the Unified Process

Depending on the level of overview that one wants to provide to a stakeholder, a number of related signals will be combined to produce a summary signal. For instance signals about feature requests, signals of feature discussions and signals about story-card creation could be combined into one overview signal about *requirements*. It is these kinds of mapped signals that should be presented to the end user. Before we extract and plot Unified Process signals we have to discuss the matter of observability of UP disciplines based on the data we have.

1) Observability: We realize that not all signals are observable based on the data we might have available. In terms of our case study we lack concrete requirements, business modelling, design and analysis, and project management signals.

2) UP Signals: When synthesizing the UP signals we found that much of the data was derived from tool use and the processes being followed. Nonetheless we tried to map what we could back to original UP disciplines. Figure 3 demonstrates the UP signals that we extracted from FreeBSD and SQLite. We will describe the signals displayed in Figure 3:

UP Business Modelling is meant to include requirements, discussion of new features, and client interaction with new features [4]. Based on our data-sets we felt the signals that related to this discipline were commits that mention requirements and commits that mention usability. Unfortunately this misses a lot of important work that might go into a project in order to determine what are its goals. In this sense we feel that this signal does not map well to observable data from FLOSS projects unless one has access to the initial discussions in which the project was fleshed out.

UP Requirements signals are built up of changes, bugs, and mailing-list messages that mention requirements, usability issues, or new functionality. That said all of these signals require lexical approaches as one has to determine if requirements activities are taking place. This produces a signal that is less trustworthy than a signal extracted from a repository of documents that are dedicated to requirements.

The UP analysis and design signal might not be easily observable in the projects we were evaluating. Analysis and design will be discussions of new features, drastic changes to architecture and new use cases. During the case study, SQLite did show some design work in terms of designing and implementing the new database file format for SQLite version 3.

The UP implementation signal is relatively simple; we include revisions that change source code. Of course this could be expanded but we did not want to include all revisions because revisions to build files or documentation are not necessarily implementation.

The UP testing signal is easily extracted from our data sources. Changes to test files, which are easily matched by filename or via lookup table, are a reliable signal to use, but other signals could be added. UP suggests improvements to qualities such as reliability, functionality, and performance relate to testing.

The UP deployment signal is about packaging, portability, distribution and building the software. Our UP deployment signal is a combination of commits that mention deployment, release and tag occurrences, portability related commits, and changes to the build system.

The UP configuration and change management signals consist of commits, and build changes. This signal is a combination of revisions and build system related commits.

The UP project management signal deals with project plans. We did not determine much in our case studies, FreeBSD and SQLite, that concerned project management beyond version control changes that might mention project management. We used a word-bag related to project management in order to find relevant commits, bugs, and discussions.

The UP Environment signal refers to process management and tools. With the data we have from FreeBSD and SQLite the closest signals are the revisions themselves and the build revisions. Commits relating to portability might also be relevant. Some projects have their own tools for cleaning up source code or for generating source code. Perhaps the addition and use of automated tools should be included.

We observe that the quality of the RUPVs depends of the available data sources. The raw signals that compose higher level signals, such as the UP signals, are more trustworthy and accurate than these aggregate mappings.

3) Alternative signals: We found that the signals extracted from the two FLOSS projects we analyzed were not a good conceptual fit for some UP disciplines, so we explored other signals that appeared promising in illustrating the development processes. Alternative process-heavy signals that could also be used are build revision signals and NFR signals.

Build revisions are a signal of changes to the build files of a software project. Changes to these build files are often indicative of changes in portability, architecture, and modularity. Portability is often related to build revision changes because supporting new platforms often causes new configuration options or checks to be employed. Architecture is often changed or modified at the same time as build revisions because the build system needs to be informed of new files or the changed files.

Non functional requirements (NFRs) signals are less accurate than say build revisions but commits, mailing-list messages and bugs that deal with non functional requirements indicate the kind of software quality related topics occurring within a repository. The focus of programmers at certain times on certain qualities are potentially good indicators of process.

We will now discuss how we took all of these signals and produced RUPVs for FreeBSD and SQLite.

VI. FreeBSD Case Study

FreeBSD[3] is an popular open source operating system. FreeBSD was based on the original Berkley Software Distribution (BSD) and BSD386. FreeBSD differs from Linux in that the FreeBSD kernel and userland (UI) are an inseparable package. FreeBSD installations are expected to have at least a bare minimum of programs and tools installed.

Figure 3 shows plots (on the left) of 9 disciplines for FreeBSD over the roughly 16 year history of the project. Figure 3 shows a large peak protruding across the FreeBSD UP signals in 2001. In 2001 FreeBSD was ported over to GCC 2.95 as its main compiler in the 4.3 branch of FreeBSD. This meant that much code had to be semi-automatically changed to conform. Mostly the function definitions had to be changed. Requirements-related changes are related to definitions and thus these definition related changes made the peak in 2001 even larger. There were also other requirements-related changes made as the project was trying to ensure

[3]FreeBSD: http://www.freebsd.org/

SUSV conformance, especially SUSV2, the Single UNIX Specification version 2.

In Figure 3, the bump in 2002 testing, deployment, and support environment signals appears to have been due primarily to changes in C coding style. Much of the FreeBSD project switched from K&R C style declarations to C89 style declarations. This is notable because many of the changes that use the term *conformance* are related to function definition style. This was most likely spurred on by the adoption of GCC3 over GCC2.95. In 2002 there were many portability-related changes as well but there were also double the releases and CVS tags than the previous year. When GCC3 was introduced a new version of binutils was introduced as well which meant that the project actively had to seek testers to ensure the new GCC and binutils still worked with FreeBSD and the ports software. OpenPAM, a pluggable authentication module was a popular topic and release tag as well.

We investigated the peak in late 2009. At this time, FreeBSD 7.2.0 was released and version 8 of FreeBSD was being prepared. Many requirements-related changes had to do with the merge of Open Source Basic Module (OpenBSM). The OpenBSM is designed for security auditing and it was merged into the FreeBSD 8 branch. Since definitions used in the OpenBSM API changed, this merge was flagged as a requirements and analysis related change.

We conclude for these examples that these RUPVs have allowed us to find interesting requirements- and analysis-related behaviours in FreeBSD.

VII. SQLite Case Study

SQLite is a library-oriented database system. It provides an SQL interface to a common SQLite database file format. It is meant for use in a wide variety of contexts, including embedded devices, such as the iPhone, and web-browsers like Mozilla Firefox. SQLite allows a program to have the power of a SQL driven relational database without the need of a separate DBMS process.

SQLite uses its own version control system called FOSSIL[4]. FOSSIL is a version control system made by the SQLite authors that integrates distributed version control, distributed bug tracking, and a distributed documentation wiki. All of these are combined together into one SQLite file.

SQLite is unusual among FLOSS projects in that the development team explicitly documents, lists, and numbers their requirements[5]. Unfortunately for this study, there seems to be little traceability between the commits and the numbered requirements.

To determine if our requirements and analysis signals were at all relevant we decided to investigate SQLite's requirements peak around 2004 (see the right side of Figure 3). In our case we did not have any tags in 2004 because when SQLite's CVS repository was converted to a FOSSIL repository, the tags were not imported. The graph, in Figure 3 around 2004 has a peak across business analysis, requirements, analysis, implementation, testing, project management, and support environment signals. We can see that a lot of work happened at that time, but what portion was related to requirements? SQLite3 was being developed and there was a need to discuss and implement an appropriate file format. Developers and users were trading implementation notes and referencing literature on database implementations in an attempt to define the SQLite3 file format. One message discussed SQL 92, while a few others discussed literature. Many source control changes were flagged as requirements because they mentioned formats and formatting, and in this case it was actually about the file format itself. Thus this visible peak was attributable to new features and new requirements.

From 2007 to 2008 the requirements, business analysis, and the UP deployment signals peaked. We investigated related events and found that in 2007 that the developers were considering converting their requirements documentation in their code into formal requirements documents. That is the software, SQLite, already worked and met most of the requirements, as well the developers had already tracked the requirements, but this was an effort to improve requirements documentation within the project.

This trend continued into 2009 where requirements were discussed further and API documentation for the C-API of SQLite was being moved out of the code itself and into separate documentation files. This was noticeable in 2009 because requirements-related signals, business analysis, requirements, and analysis, peaked up but other signals like deployment and testing peaks lagged behind requirements.

Thus for SQLite the behaviour of the requirements-related signals of UP Business Analysis, UP Requirements and UP Analysis actually did indicate that efforts were being made to define a new file format, to formalize requirements, and to move requirements from code into formal documentation.

VIII. Discussion

RUPV's multiple time-line views highlights parts of the development activity allowing us to notice behaviours that might not have been obvious from plain aggregate views such as commits per month. Mostly notably with SQLite we saw the value of the UP requirements and UP analysis signals in that we were able to find important events that would not have shown up in a commits per month signal.

A. Threats to validity

There are many threats to validity with this research. We rely on the output and artifacts left behind by programmers. This varies between projects in terms of process, amount of detail as well as consistency. Some projects are more consistent in their practices than others, while others are more lax. Certain activities might produce results which are of lower detail than others.

Some disciplines we want to observe simply are not available in the repositories that we have studied so we had to suggest proxy measures. The accuracy of these inferences and

[4]FOSSIL SCM http://fossil-scm.org/

[5]SQLite Requirements: http://www.sqlite.org/requirements.html

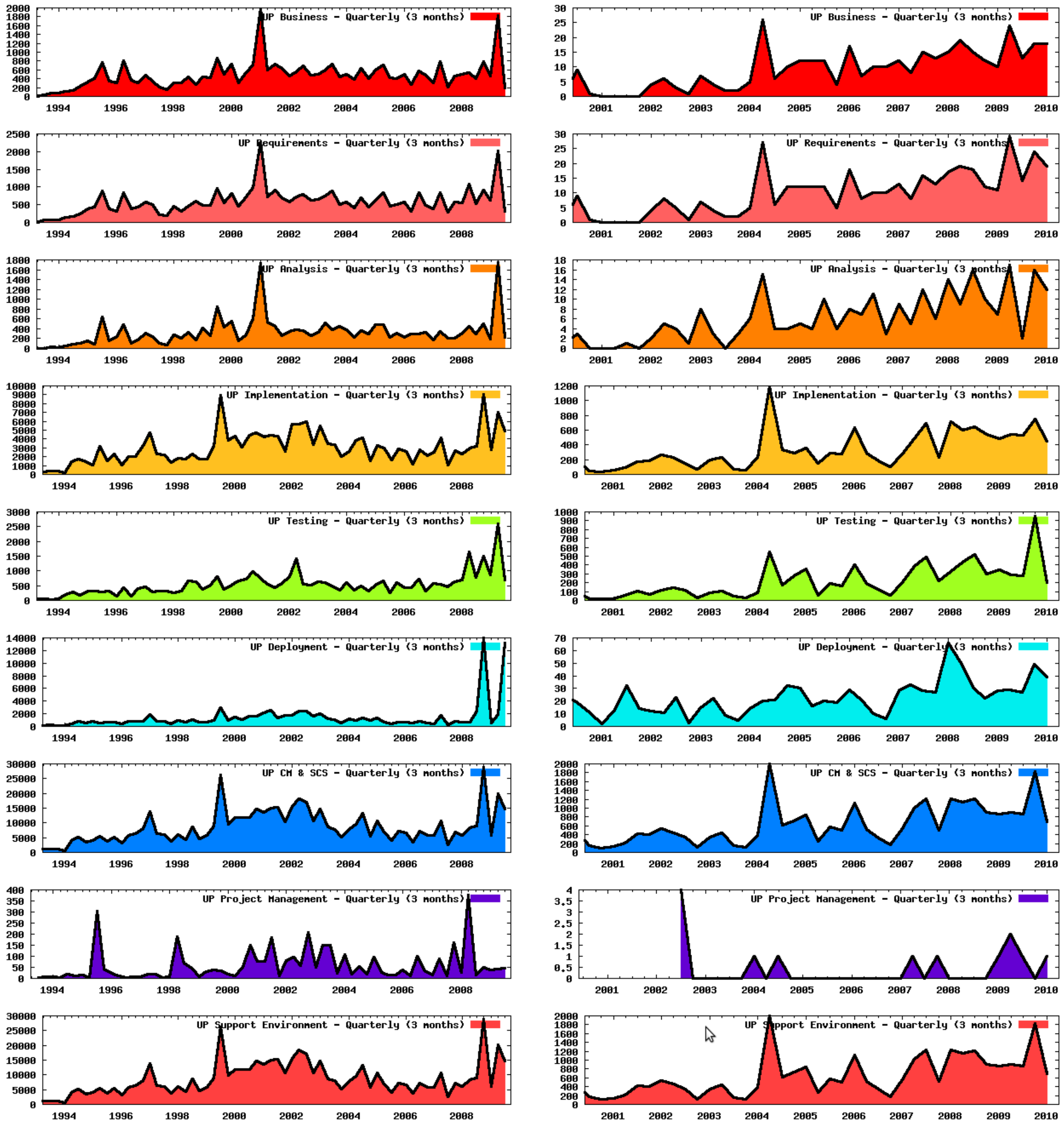

Fig. 3. Recovered Unified Process Views: These are the Recovered Unified Process Signals of FreeBSD and SQLite: business modelling, requirements, analysis and design, implementation, testing, deployment, configuration management, project management and environment. These signals have been extracted from the entire lifetime of FreeBSD and SQLite, they are composed of counts of relevant development events such as commits.

proxies can affect our results greatly. Different projects will not necessarily exhibit the same behaviour and might need largely different mappings.

The accuracy of the unsupervised and supervised analysis is a concern as it can produce spurious or biased results. For instance some of the requirements commits that were identified might have been more strongly related to design than requirements.

IX. Conclusions

We have shown that through the integration of many mining software repositories technologies that one can build high-level views about the software processes of a project. The repositories that we mined were mailing-list archives, version control systems and bug-tracker systems. Inspired by the UP diagram (see Figure 1) we wanted to integrate these data sources to produce an applied version of that diagram: Recovered Unified Process Views (RUPV). We detailed how to build RUPVs and applied our tools to two open source projects: FreeBSD and SQLite.

In order to attribute behaviour and events to various parallel disciplines we used signal mappings where we combined event and process signals to form proxy signals that represented the events related to a discipline or process. The signals proved useful when analyzing FreeBSD and SQLite as they highlighted behaviours, such as requirements and design, that would have been obscured without this kind of analysis.

A. Future Work

Our future work consists of further validating how well the observations presented relate to actual behaviour and processes that take place. We want to interview stakeholders, who are involved in development, in order to see if RUPVs relate well to their perception of the events that occurred and the processes followed. We also want to further study correlations between signals: we want to see if we can find projects with signals that are not in other projects, such as requirements, and find appropriate proxy signals that might be useful across projects.

References

[1] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proceedings of the 9th International Conference on Software Engineering*. IEEE Computer Society Press, 1987, pp. 328–339.

[2] B. Boehm, "A spiral model of software development and enhancement," *SIGSOFT Softw. Eng. Notes*, vol. 11, no. 4, pp. 14–24, 1986.

[3] I. I. Software and R. Singh, "International standard," in *Software Lifecycle Process Standards, in CrossTalk*, 1989, pp. 6–8.

[4] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maint. Evol.*, vol. 19, no. 2, pp. 77–131, 2007.

[6] M. M. Lehman, "Programs, life cycles and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[7] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in *Proceedings of International Conference on Software Maintenance*, 2000, pp. 131–142.

[8] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles, "Towards a theoretical model for software growth," in *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, p. 21.

[9] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters, "Workflow mining: A survey of issues and approaches," *Data & Knowledge Engineering*, vol. 47, no. 2, pp. 237 – 267, 2003.

[10] J. E. Cook and A. L. Wolf, "Automating process discovery through event-data analysis," in *ICSE '95: Proceedings of the 17th international conference on Software engineering*. New York, NY, USA: ACM Press, 1995, pp. 73–82.

[11] H.-J. Kung, "Quantitative method to determine software maintenance life cycle." in *ICSM*. IEEE Computer Society, 2004, pp. 232–241.

[12] K. Beck, *Extreme Programming Explained: Embrace Change*, 1st ed. Addison-Wesley Professional, October 1999.

[13] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[14] J. Highsmith and M. Fowler, "The agile manifesto," *Software Development Magazine*, vol. 9, no. 8, pp. 29–30, 2001.

[15] "Capability maturity model for software," Tech. Rep. CMU/SEI-91-TR-24 ADA240603, 1991.

[16] D. Stelzer, W. Mellis, and G. Herzwurm, "A critical look at iso 9000 for software quality management," *Software Quality Control*, vol. 6, no. 2, pp. 65–79, 1997.

[17] A. Hindle, N. Ernst, M. W. Godfrey, R. C. Holt, and J. Mylopoulos, "What's in a name? on the automated topic naming of software maintenance activities," 2010, submited to FSE 2010 http://softwareprocess.es/whats-in-a-name.

[18] N. A. Ernst and J. Mylopoulos, "On the perception of software quality requirements during the project lifecycle," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, Essen, Germany, June 2010, in press.

[19] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, "The Detection and Classification of Non-Functional Requirements with Application to Early Aspects," in *International Requirements Engineering Conference*, Minneapolis, Minnesota, 2006, pp. 39–48.

[20] L. Chung, B. A. Nixon, E. S. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, ser. International Series in Software Engineering. Boston: Kluwer Academic Publishers, October 1999, vol. 5.

[21] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *International Conference on Software Maintenance*, San Jose, CA, 2000, pp. 120–130.

[22] C. Treude and M.-A. Storey, "ConcernLines: A timeline view of co-occurring concerns," in *International Conference on Software Engineering*, Vancouver, May 2009, pp. 575–578.

[23] A. E. Hassan and R. C. Holt, "C-REX: An Evolutionary Code Extractor for C - (PDF)," University of Waterloo, Tech. Rep., http://plg.uwaterloo.ca/ aeehassa/home/pubs/crex.pdf.

[24] G. Robles, J. M. Gonzalez-Barahona, D. Izquierdo-Cortazar, and I. Herraiz, "Tools for the study of the usual data sources found in libre software projects," *International Journal of Open Source Software and Processes*, vol. 1, no. 1, pp. 24–45, Jan-March 2009.

[25] D. M. German, A. Hindle, and N. Jordan, "Visualizing the evolution of software using softchange," in *Proceedings SEKE 2004 The 16th Internation Conference on Software Engineering and Knowledge Engineering*. 3420 Main St. Skokie IL 60076, USA: Knowledge Systems Institute, June 2004, pp. 336–341.

[26] A. Hindle, M. Godfrey, and R. Holt, "Release pattern discovery: A case study of database systems," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, Oct. 2007, pp. 285–294.

[27] A. Hindle, M. W. Godfrey, and R. C. Holt, "What's hot and what's not: Windowed developer topic analysis," in *International Conference on Software Maintenance*, Edmonton, Alberta, Canada, September 2009, pp. 339–348.

[28] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, June 2008, pp. 123–132.

[29] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classification of large changes into maintenance categories," in *International Conference on Program Comprehension*, Vancouver, 2009, in press.

[30] A. Hindle, M. Godfrey, and R. Holt, "Mining recurrent activities: Fourier analysis of change events," in *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, May 2009, pp. 295–298.